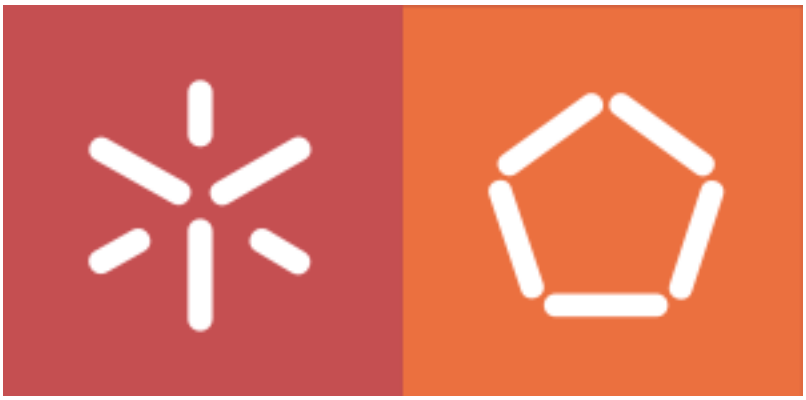

Universidade do Minho

Departamento de Informática

Computação Gráfica

Grupo nº 5

Pedro Veloso (A89557)

Carlos Preto (A89587)

Simão Monteiro (A85489)

Mafalda Costa (A83919)

Índice

Introdução

Nesta primeira fase do trabalho prático de Computação gráfica é pedido aos alunos que desenvolvam duas aplicações. A primeira, o *generator* (gerador), tem de ser capaz de gerar ficheiros com informação acerca dos modelos que se querem produzir sendo que nesta fase apenas serão gerados os vértices para esse mesmo modelo. A segunda aplicação, o *engine* (motor) tem, por sua vez, de interpretar o ficheiro recebido e mostrar ao utilizador o modelo.

O *generator* terá para já a capacidade de produzir ficheiros para as seguintes figuras geométricas: um quadrado no plano XZ, centrado na origem e desenhado a partir de 2 triângulos; uma caixa (pode ser um cubo ou não); uma esfera e finalmente um cone. Após a execução do *generator* deverá ficar criado um ficheiro guardado em memória com a informação do modelo para que depois o *engine* possa criar o mesmo a partir da sua leitura.

Generator

Na primeira parte do trabalho, pretende-se criar uma aplicação que calcula todos os pontos de uma determinada figura. As figuras implementadas pela aplicação são: plano, box, esfera e cone.

Nos tópicos seguintes encontra-se explicado o algoritmo implementado para cada uma das figuras.

Plano

De maneira a ser construído o plano, necessita-se dos valores de comprimento (x) e de largura (z), optando-se também por oferecer a opção ao utilizador de inserir o número de divisões (*partition*).

Cálculo dos Pontos

Por cada plano que for necessário desenhar, são considerados 4 pontos, onde o valor da coordenada Y em cada um deles será 0.

Uma vez que o plano está centrado na origem, ter-se-á que o comprimento e largura serão dividido por 2, de maneira que os pontos, nos extremos, variarem entre [-x/2, x/2] e [-z/2, z/2]. Conforme o número de divisões por aresta, de ponto para ponto irá haver um deslocamento nos eixos X e Z, dado por:

$$movimentoX = x / partition$$

$$movimentoZ = z / partition$$

Considerando inicialmente um plano com 1 divisões por aresta, considerou-se que o Ponto 1 teria de coordenadas (-x/2, 0, -z/2). A partir desse ponto, é possível chegar às coordenadas dos restantes pontos. O Ponto 2 resulta de uma translação nos eixos x e z de movimentoX e movimentoZ, o Ponto 3 resulta de uma translação no eixo x de movimentoX e o Ponto 4 resulta de uma translação no eixo z de movimentoZ.

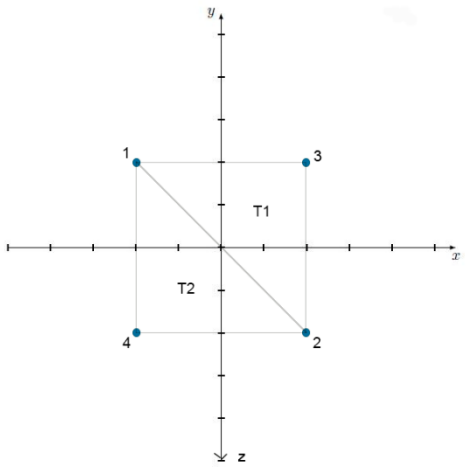

Figura 1: Exemplo Plano com 1 divisão

De maneira a obter o algoritmo do plano, optou-se por primeiro construir todos os planos em largura e ir avançado no comprimento. Também se teve em consideração a minimização de erros de vírgula flutuante, optando por isso em usar inteiros no corpo dos ciclos “for” e apenas no conteúdo destes é que se inicializavam os “floats”.

```
Para i = 0 até partition fazer i++ {
        myZ = -z/2 + (movimentoZ * i)
        Para j = 0 até partition fazer j++ {
                myX = -x/2 + (movimentoX * j)
                p1_x = myX;
                p1_z = myZ;
                p2_x = myX + movimentoX;
                p2_z = myZ + movimentoZ;
                p3_x = myX + movimentoX;
                p3_z = myZ;
                p4_x = myX;
                p4_z = myZ + movimentoZ;
        }
}
```

A razão pela qual se divide as coordenadas x e z por *partition* deve-se à possibilidade do utilizador pretender mais divisões por aresta. Por exemplo, se pretender 3 divisões, significa que o plano global passará a ser constituído por 9 planos mais pequenos, onde a variação nos eixos será menor.

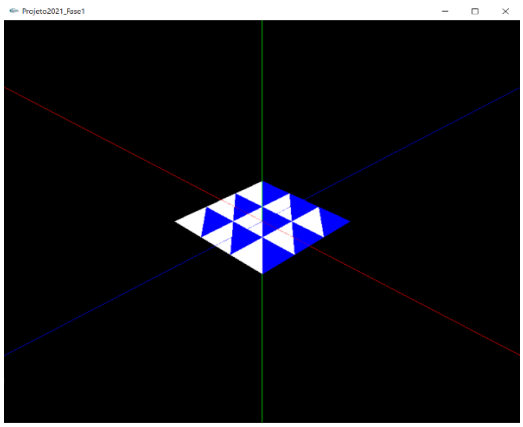

Figura 2: Exemplo Plano com 3 divisões

Box

Para se construir uma box, é necessário, tal como no plano, saber o valor do comprimento (x) e da largura (z), sendo agora também necessário saber o valor da altura (y). Tal como no plano, na box também se pode inserir quantas divisões se pretende(*partition*). Caso não se insira nenhuma *partition*, cada face da box é dividida em dois triângulos.

Cálculo dos Pontos

A box é composta por um conjunto de planos, logo apenas se teve necessidade de adaptar o raciocínio de construção do plano. Como a box tem altura, acrescentou-se qual seria o deslocamento no eixo Y, nas diferentes faces:

$$movimentoY = y / partition$$

Como a box terá de estar centrada na origem, ter-se-á que as diferentes 6 faces terão coordenadas que nunca irão variar:

- Face Frente: valor de Z nunca varia, sendo este Z/2;
- Face Trás: valor de Z nunca varia, sendo este -Z/2;
- Face Direita: valor de X nunca varia, sendo este X/2;
- Face Esquerda: valor de X nunca varia, sendo este -X/2;
- Face Cima: valor de Y nunca varia, sendo este Y/2;
- Face Baixo: valor de Y nunca varia, sendo este -Y/2.

Face da Frente

A face da frente estará assente sobre o eixo Z, tendo este o valor fixo de z/2, ou seja, apenas o valor das coordenadas nos eixos X e Y irá variar.

Considerando inicialmente um plano com 1 divisão por aresta, considerou-se que o Ponto 1 teria de coordenadas (-x/2, -y/2, z/2). A partir deste ponto, é possível chegar às coordenadas dos restantes pontos. O Ponto 2 resulta de uma translação no eixo x de movimentoX, o Ponto 3 resulta de uma translação no eixo y de movimentoY, e o Ponto 4 resulta de uma translação no eixo x e y de movimentoX e movimentoY.

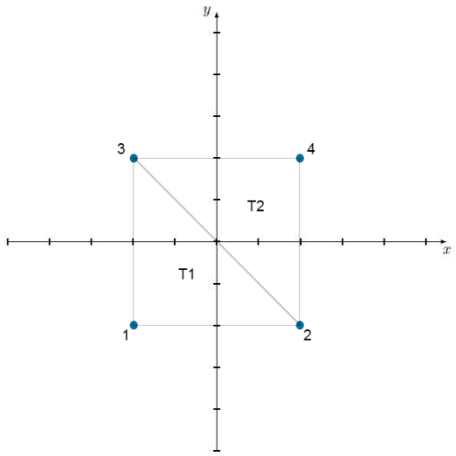

Figura 3: Exemplo Face da Frente com 1 divisão

De maneira a obter o algoritmo da face da frente, optou-se por primeiro construir todos os planos em comprimento e depois ir avançado na altura. Também se teve em consideração a minimização de erros de vírgula flutuante, optando por isso em usar inteiros no corpo dos ciclos “for” e apenas no conteúdo destes é que se inicializavam os “floats”.

```
Para i = 0 até partition fazer i++ {
    myY = -y/2 + (movimentoY * i);
    Para j = 0 até partition fazer j++ {
        myX = -x/2 + (movimentoX * i);
        p1_x = myX;
        p1_y = myY;
        p2_x = myX + movimentoX;
        p2_y = myY;
        p3_x = myX;
        p3_y = myY + movimentoY;
        p4_x = myX + movimentoX;
        p4_y =  myY + movimentoY;
    }
}
```

Face de Trás

A face de trás estará assente sobre o eixo Z, tendo este o valor fixo de -z/2, ou seja, apenas o valor das coordenadas nos eixos X e Y irá variar.

Considerando inicialmente um plano com 1 divisão por aresta, considerou-se que o Ponto 1 teria de coordenadas (-x/2, -y/2, -z/2). A partir deste ponto, é possível chegar às coordenadas dos restantes pontos. O Ponto 2 resulta de uma translação no eixo Y de movimentoY, o Ponto 3 resulta de uma translação no eixo X de movimentoX, e o Ponto 4 resulta de uma translação no eixo X e Y de movimentoX e movimentoY, respetivamente.

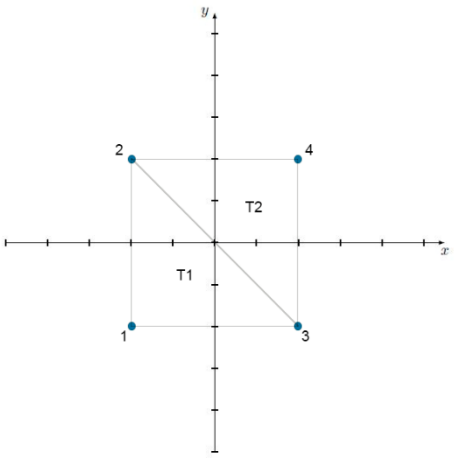

Figura 4: Exemplo Face de Trás com 1 divisão

De maneira a obter o algoritmo da face de trás, optou-se por primeiro construir todos os planos em comprimento e depois ir avançado na altura. Também se teve em consideração a minimização de erros de vírgula flutuante, optando por isso em usar inteiros no corpo dos ciclos “for” e apenas no conteúdo destes é que se inicializavam os “floats”.

```
Para i = 0 até partition fazer i++ {
        myY = -y/2 + (movimentoY * i);
        Para j = 0 até partition fazer j++ {
                myX = -x/2 + (movimentoX * j);
                p1_x = myX;
                p1_y = myY;
                p2_x = myX;
                p2_y = myY + movimentoY;
                p3_x = myX + movimentoX;
                p3_y = myY;
                p4_x = myX + movimentoX;
                p4_y = myY + movimentoY;
        }
}
```

Face da Direita

A face da direita estará assente sobre o eixo X, tendo este o valor fixo de x/2, ou seja, apenas o valor das coordenadas nos eixos Z e Y irá variar.

Considerando inicialmente um plano com 1 divisão por aresta, considerou-se que o Ponto 1 teria de coordenadas (x/2, -y/2, -z/2). A partir deste ponto, é possível chegar às coordenadas dos restantes pontos. O Ponto 2 resulta de uma translação no eixo Y e Z de movimentoY e movimentoZ. O Ponto 3 resulta de uma translação no eixo Z de movimentoZ e o Ponto 4 resulta de uma translação no eixo Y de movimentoY.

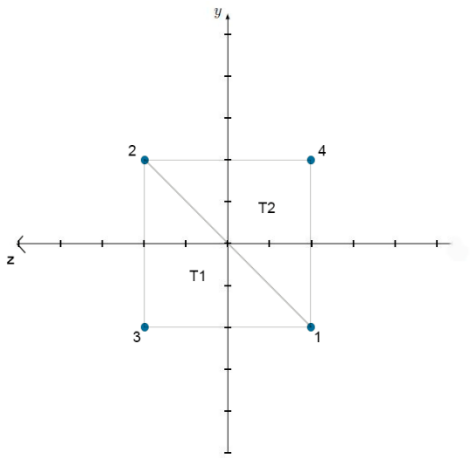

Figura 5: Exemplo Face da Direita com 1 divisão

De maneira a obter o algoritmo da face da direita, optou-se por primeiro construir todos os planos em largura e depois ir avançado na altura. Também se teve em consideração a minimização de erros de vírgula flutuante, optando por isso em usar inteiros no corpo dos ciclos “for” e apenas no conteúdo destes é que se inicializavam os “floats”.

```
Para i = 0 até partition fazer i++ {
        myY = -y/2 + (movimentoY * i);
        Para j = 0 até partition fazer j++ {
                myZ = -z/2 + (movimentoZ * j);
                p1_y = myY;
                p1_z = myZ;
                p2_y = myY + movimentoY;
                p2_z = myZ + movimentoZ;
                p3_y = myY;
                p3_z = myZ + movimentoZ;
                p4_y = myY + movimentoY;
                p4_z = myZ;
        }
}
```

Face da Esquerda

A face da esquerda estará assente sobre o eixo X, tendo este o valor fixo de -x/2, ou seja, apenas o valor das coordenadas nos eixos Z e Y irá variar.

Considerando inicialmente um plano com 1 divisão por aresta, considerou-se que o Ponto 1 teria de coordenadas (-x/2, -y/2, -z/2). A partir deste ponto, é possível chegar às coordenadas dos restantes pontos. O Ponto 2 resulta de uma translação no eixo Z de movimentoZ, o Ponto 3 resulta de uma translação no eixo Y e Z de movimentoY e movimentoZ e o Ponto 4 resulta de uma translação no eixo Y de movimentoY.

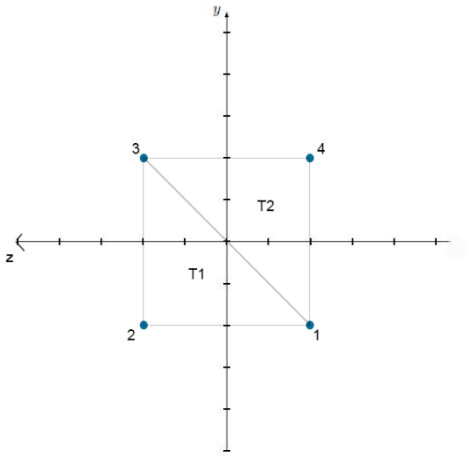

Figura 6: Exemplo Face da Esquerda com 1 divisão

De maneira a obter o algoritmo da face da esquerda, optou-se por primeiro construir todos os planos em largura e depois ir avançado na altura. Também se teve em consideração a minimização de erros de vírgula flutuante, optando por isso em usar inteiros no corpo dos ciclos “for” e apenas no conteúdo destes é que se inicializavam os “floats”.

```
Para i = 0 até partition fazer i++  {
        myY = -y/2 + (movimentoY * i);
        Para j = 0 até partition fazer j++ {
                myZ = -z/2 + (movimentoZ * j);
                p1_y = myY;
                p1_z = myZ;
                p2_y = myY;
                p2_z = myZ + movimentoZ;
                p3_y = myY + movimentoY;
                p3_z = myZ + movimentoZ;
                p4_y = myY + movimentoY;
```

```
            p4_z = myZ;
        }
    }
```

Face de Cima

A face de cima estará assente sobre o eixo Y, tendo este o valor fixo de y/2, ou seja, apenas o valor das coordenadas nos eixos X e Z irá variar.

Considerando inicialmente um plano com 1 divisão por aresta, considerou-se que o Ponto 1 teria de coordenadas (-x/2, y/2, -z/2). A partir deste ponto, é possível chegar às coordenadas dos restantes pontos. O Ponto 2 resulta de uma translação no eixo Z de movimentoZ, o Ponto 3 resulta de uma translação no eixo X e Z de movimentoX e movimentoZ, e o Ponto 4 resulta de uma translação no eixo X de movimentoX.

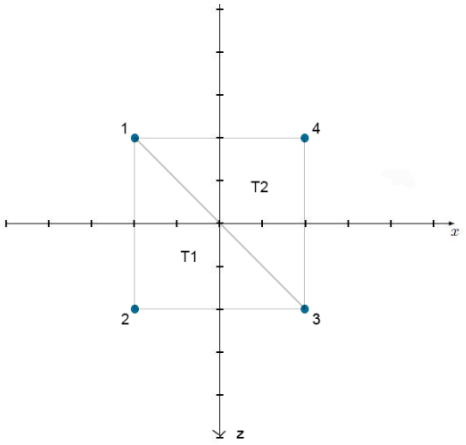

Figura 7: Exemplo Face de Cima com 1 divisão

De maneira a obter o algoritmo da face de cima, optou-se por primeiro construir todos os planos em comprimento e depois ir avançado na largura. Também se teve em consideração a minimização de erros de vírgula flutuante, optando por isso em usar inteiros no corpo dos ciclos “for” e apenas no conteúdo destes é que se inicializavam os “floats”.

```
Para i = 0 até partition fazer i++ {
    myZ = -z/2 + (movimentoZ * i);
    Para j = 0 até partition fazer j++ {
        myX = -x/2 + (movimentoX * j);
        p1_x = myX;
        p1_z = myZ;
        p2_x = myX;
```

```
                p2_z = myZ + movimentoZ;
                p3_x = myX + movimentoX;
                p3_z = myZ + movimentoZ;
                p4_x = myX + movimentoX;
                p4_z = myZ;
            }
        }
```

Face de Baixo

A face de baixo estará assente sobre o eixo Y, tendo este o valor fixo de -y/2, ou seja, apenas o valor das coordenadas nos eixos X e Z irá variar.

Considerando inicialmente um plano com 1 divisão por aresta, considerou-se que o Ponto 1 teria de coordenadas (-x/2, -y/2, -z/2). A partir deste ponto, é possível chegar às coordenadas dos restantes pontos. O Ponto 2 resulta de uma translação no eixo X e Z de movimentoX e movimentoZ, o Ponto 3 resulta de uma translação no eixo Z de movimentoZ, e o Ponto 4 resulta de uma translação no eixo X de movimentoX.

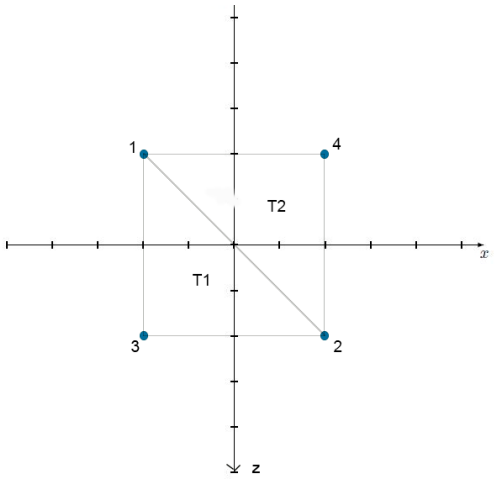

Figura 8: Exemplo Face de Baixo com 1 divisão

De maneira a obter o algoritmo da face de baixo, optou-se por primeiro construir todos os planos em comprimento e depois ir avançado na altura. Também se teve em consideração a minimização de erros de vírgula flutuante, optando por isso em usar inteiros no corpo dos ciclos “for” e apenas no conteúdo destes é que se inicializavam os “floats”.

```
Para i = 0 até partition fazer i++ {
        myZ = -z/2 + (movimentoZ * i);
        Para j = 0 até partition fazer j++ {
                myX = -x/2 + (movimentoX * j);
                p1_x = myX;
                p1_z = myZ;
                p2_x = myX + movimentoX;
                p2_z = myZ + movimentoZ;
                p3_x = myX;
                p3_z = myZ + movimentoZ;
                p4_x = myX + movimentoX;
                p4_z = myZ;
        }
}
```

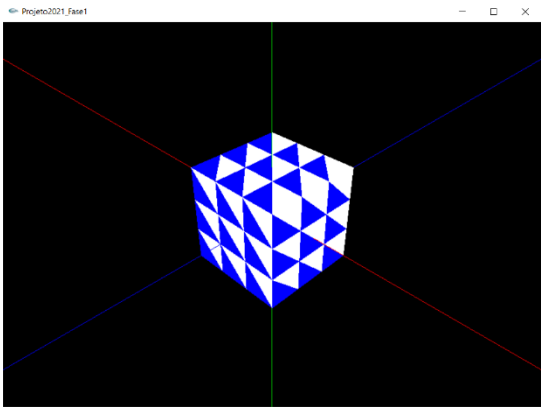

Figura 9: Exemplo Box com 3 divisões

Esfera

De maneira construir a esfera, necessita-se do raio, número de *slices* e número de *stacks*. Antes de passar para a construção do algoritmo, é necessário perceber o que são as *slices* e o que são as *stacks*. As *slices* correspondem ao número de camadas na vertical, enquanto as *stacks* correspondem ao número de camadas na horizontal que a esfera irá ter.

Cálculo dos Pontos

Dentro da esfera, os ângulos variam entre 0 e 2*PI. Ao dividir esse valor pelo número de camadas na vertical que se pretende, obtém-se o ângulo associado a cada uma das camadas na vertical:

*Alfa = 2*PI / slices*

Cada uma dessas camadas verticais irá ter um determinado número de camadas na horizontal. Para saber o ângulo associado a cada uma dessas camadas na horizontal, basta dividir PI por esse número de camadas na horizontal:

Beta = PI/ stacks

Quando se interseta uma camada na vertical com uma camada na horizontal, obtém-se 4 pontos. Cada um desses 4 pontos será definido à custa do valor dos ângulos Alfa e Beta referidos anteriormente, tal como a figura demonstra:

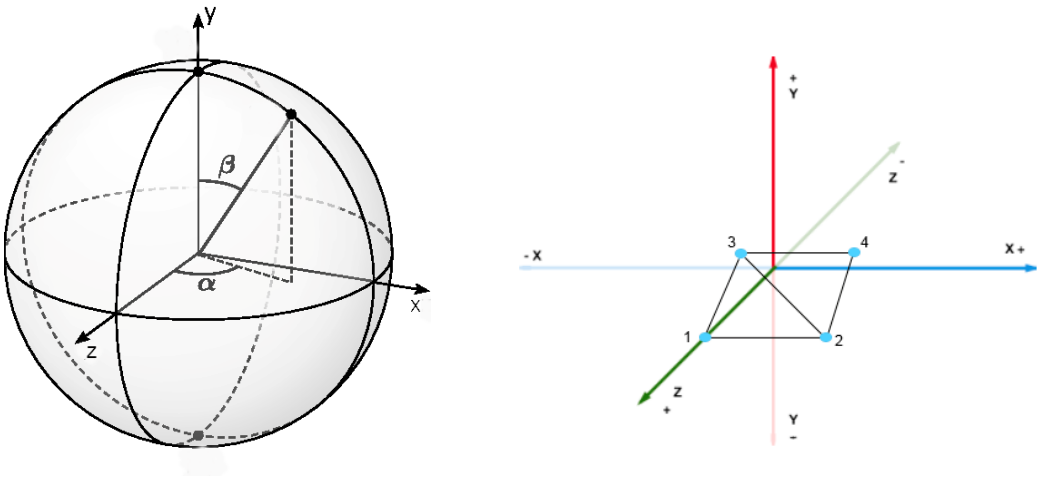

Figura 10: Exemplo face da esfera

Pela análise das Figuras, é possível tirar algumas conclusões:

- ✓ Do ponto 1 para o ponto 2 há uma variação de + Alfa
- ✓ Do ponto 1 para o ponto 3 há uma variação de + Beta
- ✓ Do ponto 1 para o ponto 4 há uma variação de + Alfa e de +Beta

Como se demonstrou, a partir de um ponto inicial 1, é possível chegar às coordenadas dos restantes pontos.

De uma *slice* para a seguinte, o ângulo Alfa vai aumentando Alfa, enquanto de uma *stack* para a seguinte, Beta vai aumentando Beta. Sempre que se avança para a *slice* seguinte, o Beta irá sempre começar em –(M_PI/2) e irá variar até M_PI/2, ou seja, obtém-se uma variação de PI, como se tinha referido no início da explicação.

Para i = 0 **até** slices **fazer** i += 1 **{**

newAlpha = alpha * i;

Para j = 0 **até** stacks **fazer** j += 1 **{**

newBeta = (-M_PI/2) + beta * j;

p1_x = radius * sin(newAlpha) * cos(newBeta);

p1_y = radius * sin(newBeta);

p1_z = radius * cos(newAlpha) * cos(newBeta);

p2_x = radius * sin(newAlpha + alpha) * cos(newBeta);

p2_y = radius * sin(newBeta);

p2_z = radius * cos(newAlpha + alpha) * cos(newBeta);

p3_x = radius * sin(newAlpha) * cos(newBeta + beta);

p3_y = radius * sin(newBeta + beta);

p3_z = radius * cos(newAlpha) * cos(newBeta + beta);

p4_x = radius * sin(newAlpha + alpha) * cos(newBeta + beta);

p4_y = radius * sin(newBeta + beta);

p4_z = radius * cos(newAlpha + alpha) * cos(newBeta + beta);

}

}

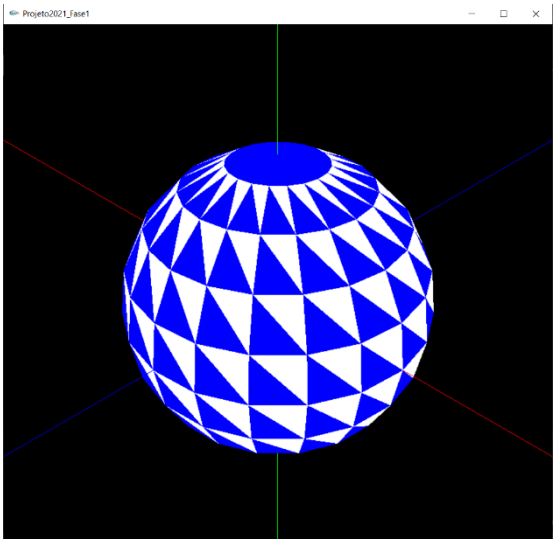

Figura 11: Exemplo Esfera

Cone

Um cone pode ser visto como a junção de uma circunferência na base e pirâmides à volta dessa mesma base. Para se construir um cone é necessário saber o valor do raio da circunferência da base (*radius*), altura do cone (*height*), *slices* e *stacks*.

Cálculo dos Pontos

Uma vez que o cone terá de estar centrado na origem, conclui-se que todos os pontos que constituem a base deste terão o mesmo valor de Y, que será -altura/2. À semelhança da esfera, tem-se que o ângulo ente cada camada na vertical será:

*Alfa = 2*PI / slices*

Também se sabe qual o espaçamento entre camadas na horizontal, sendo este:

horizontalSpace = height/stacks

Definidos estes valores, passa-se então à construção do cone, dividindo este em 2 fases, como dito anteriormente.

Base

A base é constituída por uma série de triângulos, onde em cada um destes o ângulo vai variando em +Alfa, sendo que com base nesse ângulo, é possível saber as coordenadas de X e Z.

Uma vez que o cone terá de estar centrado na origem, então a altura dos vértices da base do cone estarão na coordenada Y:

newHeight = -height/2;

Assim, o ponto 1 terá sempre coordenadas (0, newHeight, 0). O ponto 2 terá sempre um ângulo superior ao ponto 3, sendo o aumento do ângulo de 3 para 2 de +Alfa. A sucessiva variação do ângulo dos triângulos leva à criação de uma circunferência:

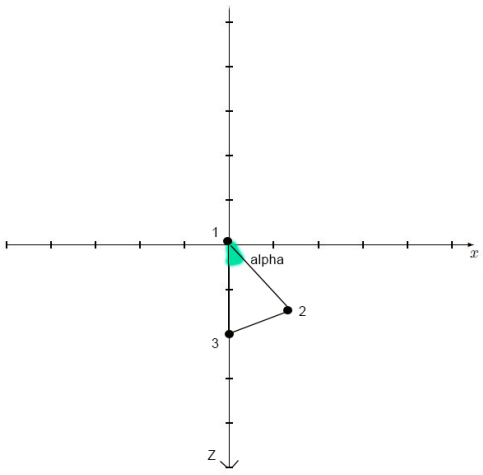

Figura 12: Exemplo de um dos triângulos da base

Para i = 0 **até** slices **fazer** i += 1 **{**

newAlpha = alpha * i;

p1_x = 0;

p1_y = newHeight;

p1_z = 0

p2_x = radius * sin(newAlpha + alpha);

p2_y = newHeight;

p2_z = radius * cos(newAlpha + alpha);

p3_x = radius * sin(newAlpha);

p3_y = newHeight;

p3_z = radius * cos(newAlpha);

}

Faces Laterais

As faces laterais podem ser divididas em 2 partes, uma correspondente à parte de baixo (fb) e outra à parte de cima (fc), onde cada uma vai variar conforme a iteração i (de 0 até *stacks*) em qual se encontrar:

$$fb = newHeight + (i * horizontalSpace)$$

$$fc = newHeight + ((i+1) * horizontalSpace)$$

Também se definiu um raio para a parte de baixo(rb) e um raio para a parte de cima(rc):

$$rb = radius – ((radius/stacks) * i)$$

$$rc = radius – ((radius/stacks) * (i+1))$$

A razão pela qual na parte de baixo se multiplica por i, e na parte de cima se multiplicar por i+1, deve-se ao facto de à medida que se avança nas *stacks*, o valor de X ir diminuindo e o valor de Y ir aumentando. Tal observação torna-se fácil através da análise lateral de uma face da pirâmide, onde como se pode ver, vai-se sempre removendo (raio/*stacks*) à medida que se avança nas *stacks* (parte verde) e o valor de Y vai sempre aumentando (horizontalSpace) até chegar ao topo da pirâmide (parte vermelha).

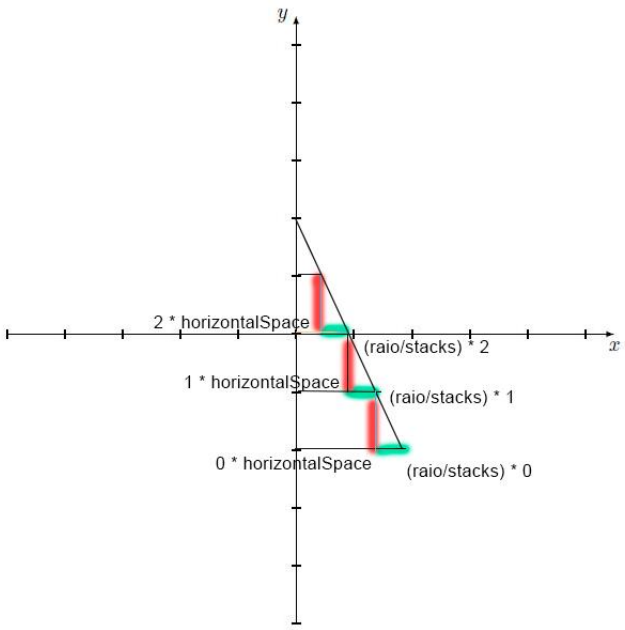

Figura 13: Exemplo Faces Laterais

Para i = 0 **até** stacks **fazer** i += 1 **{**

fb = newHeight + (i * horizontalSpace)

fc = newHeight + ((i+1) * horizontalSpace)

rb = radius – ((radius/stacks) * i)

rc = radius – ((radius/stacks) * (i+1))

Para j = 0 **até** slices **fazer** j += 1 **{**

height = alpha * j;

t1_p1_x = rb * sin(height);

t1_p1_y = fb;

t1_p1_z = rb * cos(height);

t1_p2_x = rc * sin(height + alpha);

t1_p2_y = fc;

t1_p2_z = rc * cos(height + alpha);

t1_p3_x = rc * sin(height);

t1_p3_y = fc;

t1_p3_z = rc * cos(height);

t2_p1_x = rb * sin(height);

t2_p1_y = fb;

t2_p1_z = rb * cos(height);

```
                t2_p2_x = rc * sin(height + alpha);
                t2_p2_y = fb;
                t2_p2_z = rc * cos(height + alpha);
                t2_p3_x = rc * sin(height + alpha);
                t2_p3_y = fc;
                t2_p3_z = rc * cos(height + alpha);

        }
}
```

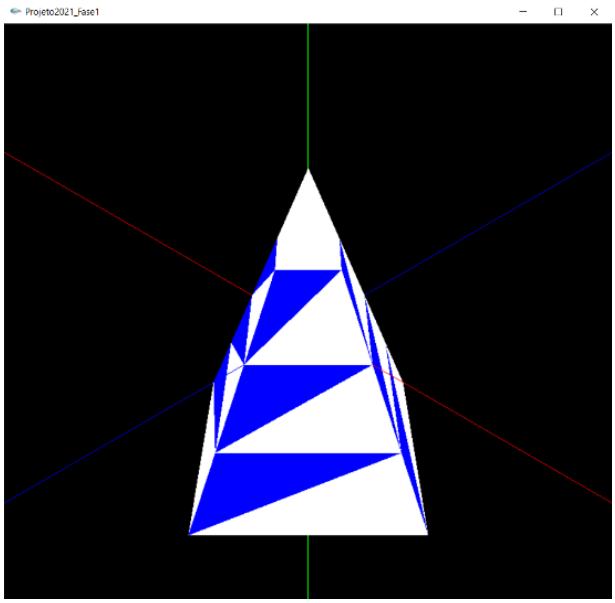

Figura 14: Exemplo Cone

Engine

Na segunda parte do trabalho, desenvolveu-se uma aplicação que lê um ficheiro com o formato XML e desenha os objetos encontrados. No ficheiro XML encontra-se registado o caminho para obter os modelos a desenhar. Uma vez que os modelos que a aplicação trata são gerados pelo *generator* desenvolvido na primeira parte do trabalho (formato 3d), realizou-se um algoritmo capaz de ler o ficheiro e registar, em memória, todos os pontos.

É possível, também, interagir com a aplicação, através do rato e do teclado, podendo observar o modelo de diferentes ângulos e formas.

Leitura de ficheiros

A primeira fase da *engine* passa por ler todos os pontos que mais tarde irão ser desenhados no ecrã. Contudo, o ficheiro lido pela *engine* é um ficheiro em formato XML, que apenas contém o caminho para os pontos. Sendo assim, necessário realizar um *parsing* no XML.

Para realizar o *parsing* do ficheiro XML utilizou-se o “TinyXML-2” o que permitiu navegar de uma forma simples pelos componentes do XML, e obter os ficheiros seguintes a ler.

```
void load (const char* pFilename) {
    this->modelos = list<Modelo>();

    XMLDocument doc;
    XMLElement *root;

    Cor c1, c2;
    c1.r = c1.g =0.0f;
    c1.b = c2.r = c2.g = c2.b = 1.0f;

    Cor c[2];
    c[0] = c1;
    c[1] = c2;

    if (!(doc.LoadFile(pFilename))) {
        root = doc.FirstChildElement();
        for (XMLElement *elem = root->FirstChildElement(); elem != NULL; elem = elem->NextSiblingElement()) {
            string ficheiro = elem->Attribute("file");

            map<Cor, list<Triangulo>> lista = load(ficheiro,c);

            if(!lista.empty()){
                Modelo m(lista);
                this->modelos.push_back(m);
            }
        }
    } else {
        printf("Ficheiro XML não encontrado\n");
    }
}
```

Figura 15: Algoritmo de parsing do ficheiro XML

Uma vez obtidos os nomes dos ficheiros com formato .3d, gerados previamente, é realizada a leitura dos pontos que constituem o modelo. Essa leitura, engloba ler a primeira linha do ficheiro que contém o número de pontos totais. Posteriormente é realizada a leitura dos pontos (representados pelas suas coordenadas x, y e z) até serem lidos todos os pontos.

Registar os pontos em memória

Após a leitura dos pontos, pretende-se desenhá-los no ecrã. Contudo, é necessário registá-los em memória aquando da leitura.

Para guardar os pontos criaram-se algumas classes, com o objetivo de tornar a organização da informação mais simples.

Uma vez que a informação de mais baixo nível obtida é o ponto criou-se a classe "Ponto" com as componentes x, y e z que representam as coordenadas de um ponto num referencial *xyz*.

```
class Ponto {
public:
    float x;
    float y;
    float z;
};
```

Figura 16: Classe Ponto

Com o acumular de pontos é interessante, no contexto de computação gráfica, representar um conjunto de três pontos por um triângulo. Assim criou-se a classe "Triângulo" caracterizada pela figura seguinte.

```
class Triangulo {
public:
    Ponto p1;
    Ponto p2;
    Ponto p3;
};
```

Figura 17: Classe Triângulo

Uma vez que um modelo é um conjunto de triângulos, torna-se lógico agrupar os triângulos numa única classe. Essa classe deve ser constituída por uma lista com todos os triângulos existentes no modelo. Contudo, optou-se por agrupar os triângulos em listas distintas, contendo em cada lista apenas os triângulos da mesma cor. Desta forma é possível desenhar no ecrã os triângulos todos de uma determinada cor, de forma sequencial. A classe modelo está representada na figura seguinte.

```
class Modelo{
    public:

    map<Cor, list<Triangulo>> triangulos;

    Modelo(){
        this->triangulos = map<Cor, list<Triangulo>>() ;
    }

    Modelo(map<Cor, list<Triangulo>> triangulos){
        this->triangulos = triangulos ;
    }
} ;
```

Figura 18: Classe Modelo

Por fim, uma vez que uma cena pode conter mais que um modelo, criou-se a classe “Cena”. Nesta classe é possível encontrar uma lista com todos os modelos que deverão ser desenhados no ecrã. Criou-se também nesta classe a função *load* responsável por ler um ficheiro em formato XML e povoar todas as estruturas de dados acima referidas.

Interação

Foram implementados comandos que permitem interagir com as renderizações obtidas. Assim a tabela seguinte representa as funcionalidades associadas a cada tecla.

Tabela 1: Associação de funcionalidade-tecla

Tecla	Funcionalidade
X	Define que a translação, rotação e escala será realizada considerando o eixo do X;
Y	Define que a translação, rotação e escala será realizada considerando o eixo do Y;
Z	Define que a translação, rotação e escala será realizada considerando o eixo do Z;
A	Diminui a coordenada selecionada. Como *default* utiliza a coordenada X;
D	Aumenta a coordenada selecionada. Como *default* utiliza a coordenada X;
I	Define a visualização como linhas;
O	Define a visualização como pontos;
P	Define a visualização como sólido;
UP	Aumenta a escala em função do eixo selecionado. Como *default* utiliza o eixo do X;
DOWN	Diminui a escala em função do eixo selecionado. Como *default* utiliza o eixo do X;
RIGHT	Aumenta o ângulo de rotação em torno do eixo selecionado. Como *default* utiliza o eixo do X;
LEFT	Diminui o ângulo de rotação em torno do eixo selecionado. Como *default* utiliza o eixo do X;
N	Define que o modo como os pontos são representados é “GL_CW”;
M	Define que o modo como os pontos são representados é “GL_CCW”.

Uma vez que é possível realizar escalamentos no modelo, foi necessário ponderar acerca de um escalonamento negativo, sobre 1,2 ou 3 eixos.

Realizar um escalonamento negativo leva a que o modelo sofra uma inversão sobre o eixo de escalonamento. Assim os pontos orientados para a frente encontram-se na face de trás do modelo. Contudo, como apenas são renderizados os pontos orientados para a frente, passa a ser visível a parte de trás do modelo, mantendo a perspetiva da frente. Para ultrapassar este problema, sempre que o escalonamento altera o sinal, é calculada a nova forma como se calcula a orientação dos pontos.

O algoritmo utilizado é o seguinte:

```
if (*escala > 0 && (*escala - 0.2 < 0)) {
    int num = vetorEscala.numCoordNegativas();

    if (num == 0) {
        glFrontFace(GL_CCW);
    }
    else {
        if (num == 2)
            glFrontFace(GL_CCW);
        else
            glFrontFace(GL_CW);
    }
}
```

```
if (*escala < 0 && (*escala + 0.2 > 0)) {
    int num = vetorEscala.numCoordNegativas();

    if (num == 0) {
        glFrontFace(GL_CCW);
    }
    else {
        if (num == 2)
            glFrontFace(GL_CCW);
        else
            glFrontFace(GL_CW);
    }
}
```

Figura 19

Além de ser possível de interagir diretamente com o modelo renderizado, através de translações, rotações e escalonamentos, também é possível de interagir com a câmara. Utilizando o rato é, então possível interagir com a câmara, através do modo de explorador. Assim temos:

Tabela 2: Movimentos do rato

Comando	Funcionalidade
Botão esquerdo do rato + movimento	Movimenta a câmara, mantendo o foco no ponto (0,0,0)
Botão direito do rato + movimento positivo no eixo do Y (cima)	Diminui a distância da câmara ao ponto (0,0,0)
Botão direito do rato + movimento negativo no eixo do Y (baixo)	Aumenta a distância da câmara ao ponto (0,0,0)

Conclusão

Terminada a primeira fase deste trabalho podemos afirmar que estamos contentes com os resultados obtidos dado que tanto o *generator* como o *engine* estão a funcionar como esperávamos.

É possível através do *generator* obter todas as figuras geométricas propostas no trabalho. A implementação do *generator* permitiu entender melhor como são representados os modelos 3D.

O desenvolvimento da *engine*, colocou em prática todos os conceitos aprendidos nas aulas, permitindo desenvolver capacidades e conciliar conceitos.

Assim com a *engine* é possível realizar translações, rotações e escalonamentos ao modelo, previamente gerado, bem como realizar uma exploração do modelo através do modo de expetador. Esta exploração só é possível devido ao movimento da câmara em torno de um ponto.

Consideramos também, que progressão faseada do trabalho, que até ao momento vai acompanhando a matéria lecionada nas aulas, é também uma mais-valia, dado que funciona como uma ferramenta de estudo para pôr em prática o que aprendemos nas aulas.

Bibliografia

TinyXML-2: *www.github.com/leethomason/tinyxml2/releases/tag/8.0.0*