Universidade do Minho
Escola de Engenharia

COMPUTAÇÃO GRÁFICA

PHASE 1 - GRAPHICAL PRIMITIVES

MARÇO 2021

Luís Miguel Pinto A89506

Ana Luísa Carneiro A89533

Pedro Almeida Fernandes A89574

Ana Rita Peixoto A89612

ÍNDICE

INTRODUÇÃO

O objetivo da 1ª fase do trabalho consistiu no desenvolvimento de um pequeno cenário gráfico 3D através da utilização de algumas primitivas gráficas tais como o plano, a caixa, a esfera e o cone.

Para esta fase foi necessário implementar duas aplicações: o *Generator* - capaz de criar ficheiros 3d com informação dos modelos (os vértices constituintes de cada primitiva); e o *Engine* - lê o ficheiro de configuração XML, fornecido pelo *Generator*, guarda em estruturas adequadas e, em seguida, apresenta os modelos em 3D.

Por motivos de teste, decidimos implementar uma câmara capaz de observar a primitiva em todas as suas vertentes. Para além disso, decidimos acrescentar também a primitiva cilindro no projeto como forma de torná-lo mais completo.

No presente relatório apresentamos explicações detalhadas de todas as etapas que sustentaram esta fase do projeto, acompanhadas por equações, algoritmos em pseudocódigo, imagens e diagramas, com o nome das variáveis utilizadas, de forma a ilustrar o raciocínio usado e manter uma documentação exemplificativa do projeto.

ESTRUTURA DO PROJETO

Como forma de melhor organização, o projeto foi estruturado recorrendo a diferentes diretorias: uma com informação relativa ao *Generator*, outra em relação ao *Engine* e, por último, uma diretoria *Models* que armazena os ficheiros XML. De seguida, encontra-se uma imagem representativa da estrutura do projeto:

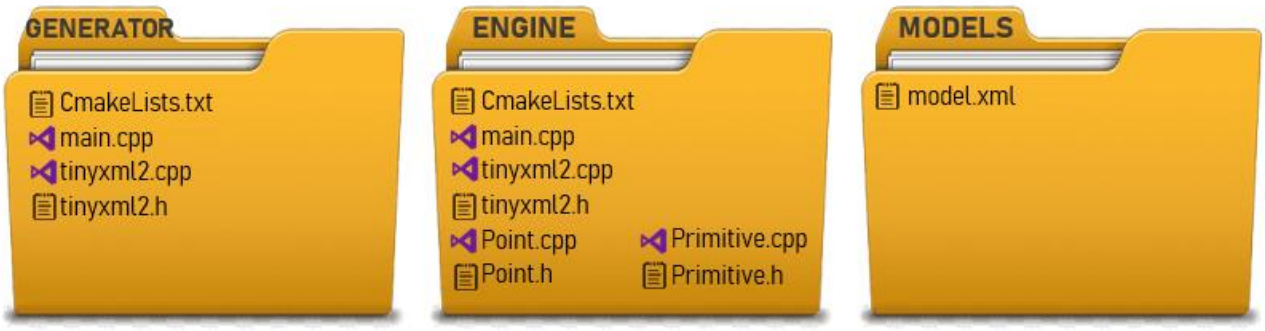


Figura 1: Estrutura do Projeto

Na diretoria *Generator* encontra-se todo o código associado à geração de coordenadas para a representação dos vários modelos. Assim, todos os ficheiros .3d onde estão armazenadas as várias coordenadas vão ser guardados na directoria Models juntamente com o ficheiro XML. Esta diretoria equivale ao demo scenes pedido no enunciado. Aquando da criação dos ficheiros .3d, estes são também armazenados no ficheiro XML, model.xml, para posteriormente ser utilizado pelo *Engine*.

No *Engine* encontra-se todo o código associado à implementação em 3D dos vários modelos. Através da leitura do ficheiro model.xml conseguimos aceder aos vários ficheiros .3d gerados que contém todos os pontos dos triângulos.

Durante a leitura do XML e consequente leitura dos ficheiros .3d por parte do *Engine*, os pontos vão ser armazenados numa estrutura. Para isso foram utilizadas as classes Point.cpp e Primitiva.cpp.

Nas diretorias *Generator* e *Engine* foi utilizada a classe tinyxml2.cpp para a configuração do ficheiro XML.

GENERATOR

Esta aplicação é responsável por gerar os vértices que irão dar origem aos triângulos das diferentes primitivas, através de algoritmos específicos. Após a geração dos vértices, estes serão guardados num ficheiro .3d que será referido num ficheiro XML.

Para a compilação foi utilizado um ficheiro *CMakeLists*, que irá permitir gerar o projeto através do *cmake.*

Para o tratamento e acesso aos ficheiros .3d e XML, foi necessário implementar um mecanismo que generalizasse os caminhos para os ficheiros, de modo a tornar o programa compatível com diferentes máquinas. Para concretizar este objetivo, foi necessário incluir diferentes bibliotecas tendo em conta o sistema operativo atual. Além disso, foi necessário também manipulação de *strings* para obter o caminho pretendido.

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 vírgula. A seguinte figura pretende ilustrar a estrutura do ficheiro:

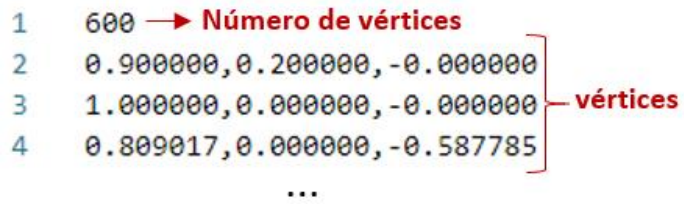


Figura 2: Ficheiro .3d

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). Caso geremos pontos no *Generator* com um nome de ficheiro já existente, este vai reescrito com a nova informação. Contudo, no ficheiro XML vai aparecer dois ficheiros .3d com o mesmo nome o que resulta na geração de duas primitivas idênticas sobrepostas. De seguida encontra-se um exemplo de um possível ficheiro XML gerado:

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

Figura 3: Exemplo de conteúdo do model.xml

PRIMITIVAS

Neste projeto foi requisitada a implementação de diferentes primitivas: plano, caixa, esfera e cone. Além dessas, foi também adicionada a primitiva cilindro.

Nas funções relativas a cada primitiva são gerados os vértices que lhes darão origem. Estas funções recebem parâmetros de acordo com a primitiva em questão, por exemplo o raio ou a altura do sólido. Assim, achamos conveniente garantir que os valores fossem válidos. Desta forma, cada uma das primitivas está encarregue de efetuar essa verificação. Valores nulos ou negativos não deverão ser aceites.

De modo a gerar as diferentes primitivas, é necessário fornecer diferentes comandos ao *Generator.* Os comandos possíveis estão apresentados de seguida:

- **sphere** radius slices stacks file
- **cone** radius height stacks slices file
- **box** x y z edges file
- **box** x y z file
- **plane** side file
- **cylinder** radius height slices file

SPHERE

Na construção de uma esfera, para além do habitual parâmetro de raio (radius), foi também necessário usar *stacks* e *slices*. *Stacks* correspondem às camadas da esfera (divisões horizontais) enquanto *slices* se referem às fatias (divisões verticais). Para dividir a esfera em *stacks* é preciso ter em atenção a altura, enquanto que para as *slices* a preocupação foca-se nas partições a partir do ângulo alpha. Ambas as divisões dividem a esfera em secções de iguais dimensões. Na figura abaixo conseguimos ver uma esfera dividida em 6 *stacks* e 4 *slices*.

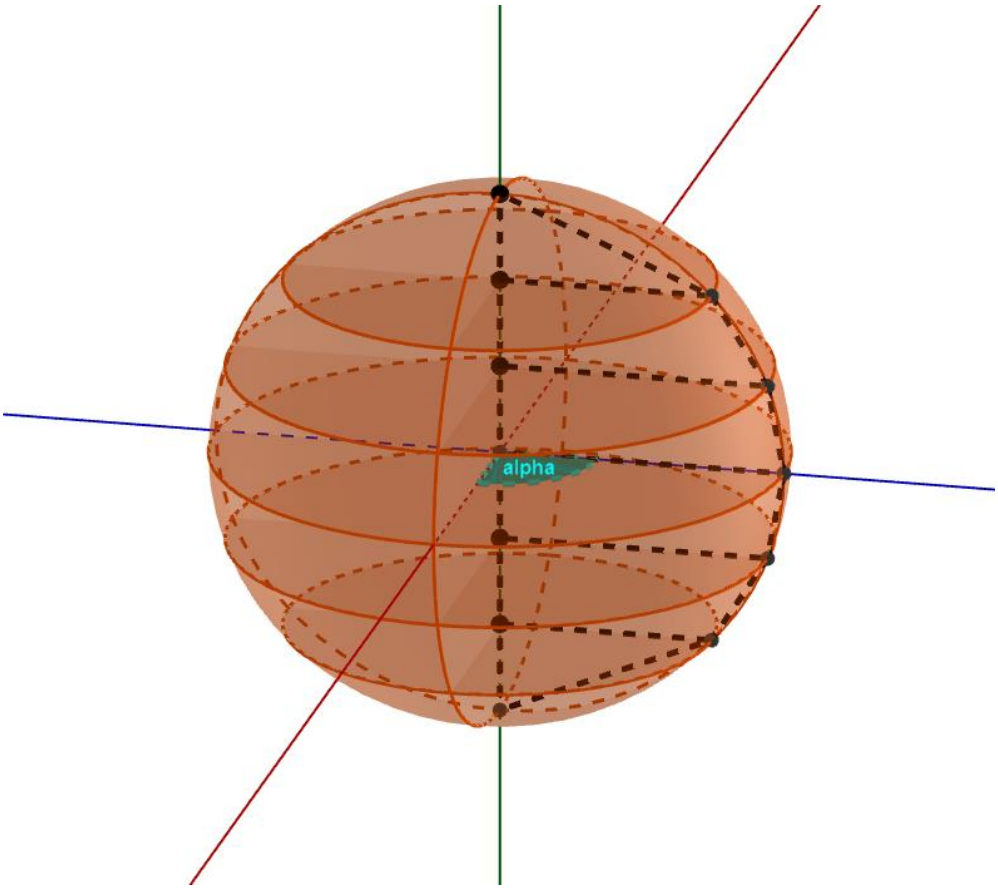


Figura 4: Esfera dividida em stacks e slices

Para representar a esfera, a abordagem baseou-se em representar para cada stack os vértices de cada slice. Desta forma, seria necessário saber em cada stack 3 fatores:

1) a sua altura (stackHeight);
2) o seu ângulo de rotação para cada slice (alpha);
3) o seu raio (stackRadius).

1) No que toca à **stackHeight** de cada *stack*, o cálculo é trivial:

$$stackHeight = \frac{(2 \times radius)}{stacks}$$

$$stackHeight\ (i) = radius - stackHeight \times i\ ,$$
$$para\ stack\ (i) \wedge 0 \leq i < stacks \wedge i \in N$$

2) Relativamente ao ângulo **alpha** ilustrado na esfera da figura anterior, apenas necessitamos de saber o número de *slices*, assim:

$$alpha = \frac{(2 \times \pi)}{slices}$$

3) Para o cálculo do **stackRadius** é necessário ter em conta o teorema de Pitágoras - "Em qualquer triângulo retângulo, o quadrado do comprimento da hipotenusa é igual à soma dos quadrados dos comprimentos dos catetos.". No nosso caso, este exemplo aplica-se entre o triângulo formado pelo **radius**, a **stackHeight** da iteração atual e a nossa incógnita, o **stackRadius.** Matematicamente e com o auxílio da seguinte figura isto traduz-se em:

$$stackRadius = \sqrt{radius^2 - i^2}\ , \qquad 0 < i \leq radius$$

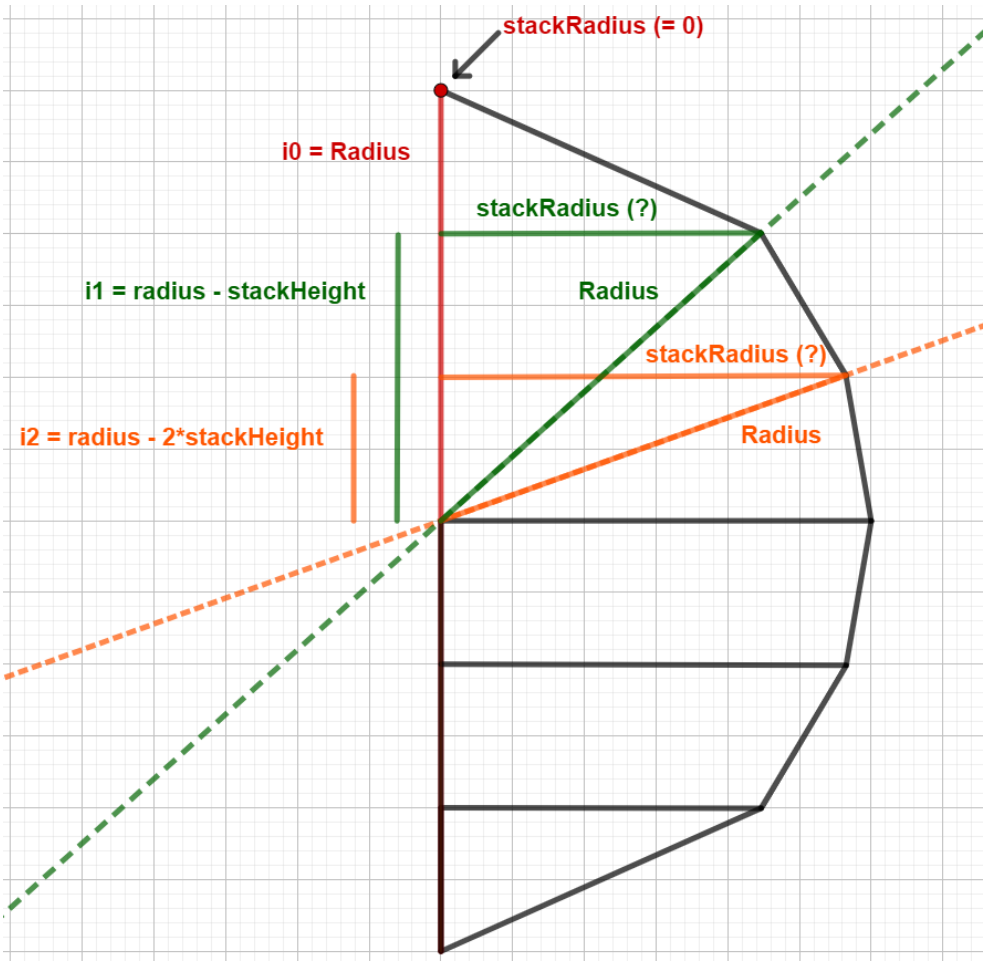


Figura 5: Cálculo do stackRadius

Estando agora munidos dos parâmetros necessários, podemos dar início à construção da esfera.

O algoritmo é simples, para cada *stack* e *slice* e suas correspondentes sucessoras, iremos calcular os 4 pontos do plano por elas formado. Para desenhar a esfera, o algoritmo começa no topo da esfera, a cada iteração desenha uma stack da semiesfera positiva, replica para a negativa e desce uma stackHeight. Nos casos em que o número de stacks é ímpar existe uma condição que evita o espelhamento da última stack de modo a não haver repetição de triângulos já existentes. Desta forma apenas bastará, encontrados os pontos, ordená-los segundo a regra da mão direita para obter triângulos capazes de representar uma esfera. De seguida encontra-se o pseudocódigo do algoritmo referido:

```
stackHeight geral = (2 * radius) / stacks
enquanto height i > 0 {
        stackRadius = sqrt(pow(radius, 2) - pow(i, 2))
        stackRadius2 = sqrt(pow(radius, 2) - pow(i-stackHeight, 2))
        Para cada slice j {
                alpha = ((2 *  π)/ slices) * j
                alpha2 = ((2 * π) / slices )* j+1

                p1x = cos(alpha) * stackRadius
                p1y = i
                p1z = -sin(alpha) * stackRadius

                p2x = cos(alpha2) * stackRadius
                p2y = i
                p2z = -sin(alpha2) * stackRadius

                p3x = cos(alpha) * stackRadius2
                p3y = i - stackHeight
                p3z = -sin(alpha) * stackRadius2

                p4x = cos(alpha2) * stackRadius2
                p4y = i - stackHeight
                p4z = -sin(alpha2) * stackRadius2
                //caso ímpar
                //repetição hemisfera negativa
        }
}
```

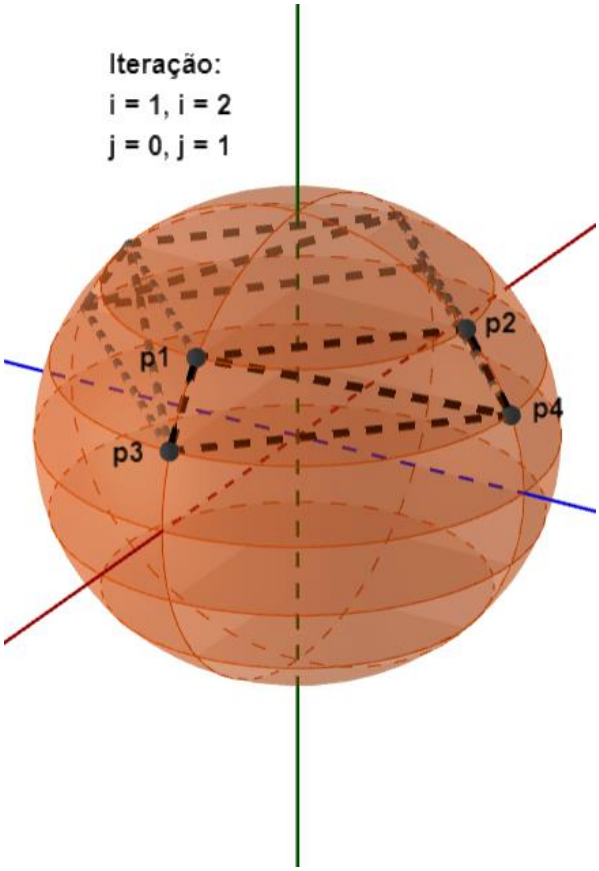


Figura 6: Pontos da interseção das *slices* por *stacks*

Para gerar os triângulos do exemplo acima poderíamos usar a ordem p1 p3 p4 e p1 p4 p2, respetivamente.

Antes de guardar os vértices do ficheiro .3d, indicamos o número de vértices presentes na esfera. Para isso podemos usar a seguinte expressão matemática:

$$Nº\ Vértices = 6 \times slices \times (stacks - 1)$$

A expressão para construir uma esfera é **sphere radius slices stacks file**. Por fim o resultado obtido foi o seguinte:

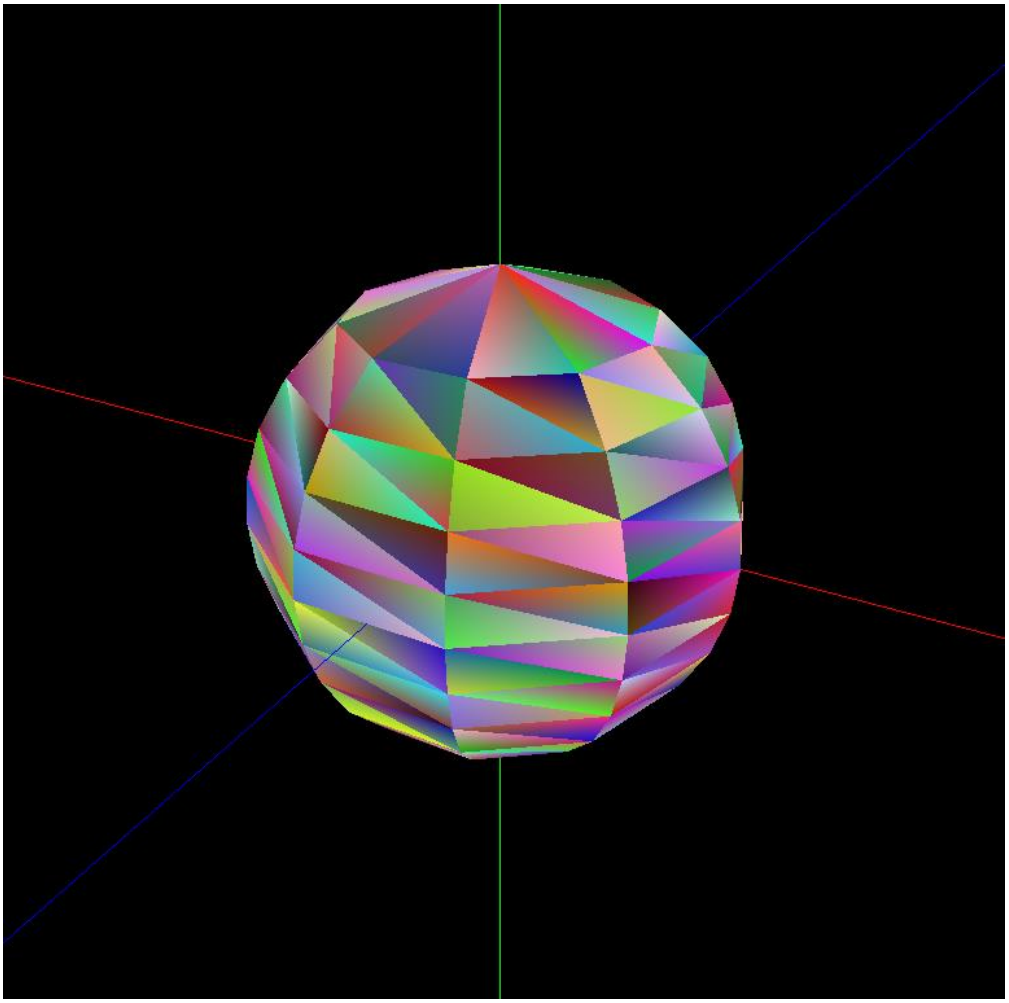


Figura 7: sphere 2 10 10 sphere.3d

CONE

Na construção de um cone, para além dos habituais parâmetros de raio (radius) e altura (height), foi necessário usar *stacks* e *slices*. *Stacks* correspondem às camadas do cone (divisões horizontais) enquanto *slices* se referem às fatias (divisões verticais). Para desenhar a esfera, o algoritmo começa na base do cone e a cada iteração sobe uma **stackHeight**. Para dividir o cone em *stacks* é preciso ter em atenção a altura, enquanto para as *slices* a preocupação foca-se nas partições a partir do ângulo alpha. Ambas as divisões dividem o cone em secções de iguais dimensões.Na figura abaixo conseguimos ver um cone dividido em 3 *stacks* e 4 *slices*.

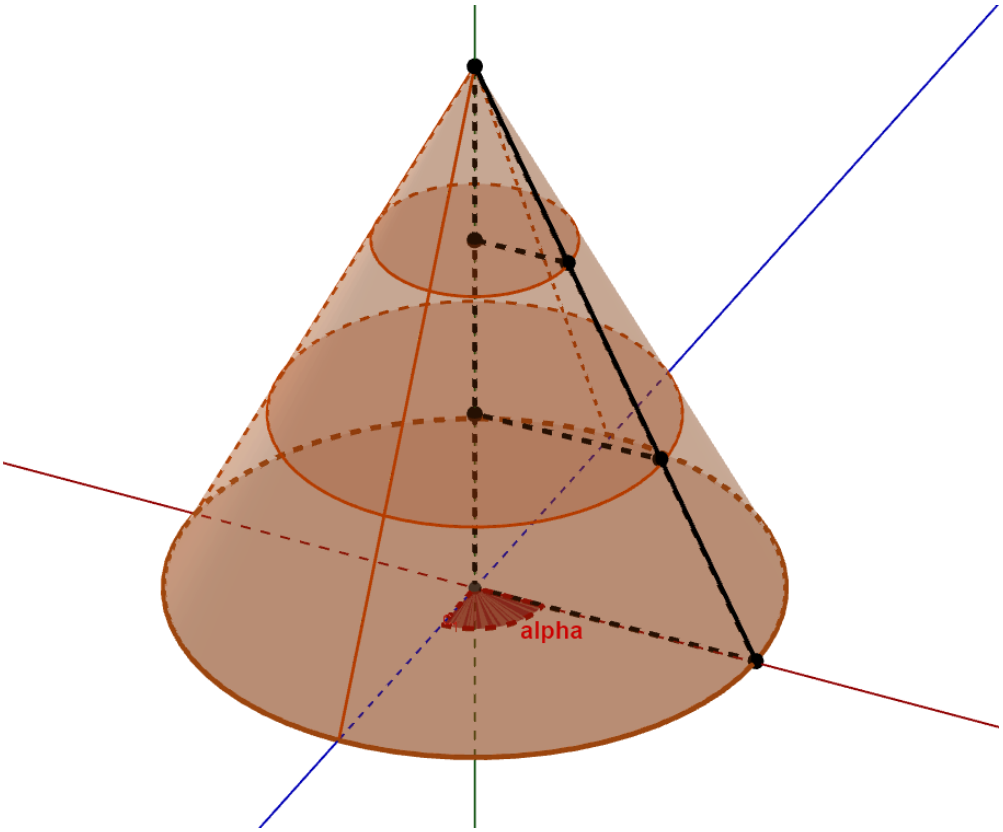


Figura 8: Cone dividido em *stacks* e *slices*

Para representar o cone, a abordagem baseou-se em para cada *stack* representar os vértices de cada *slice*. Desta forma, seria necessário saber em cada *stack* três fatores:

1) a sua altura (stackHeight) ;
2) o seu ângulo de rotação para cada slice (alpha);
3) o seu raio (stackRadius).

1) No que toca a **stackHeight** de cada *stack*, o cálculo é trivial:

$$stackHeight = \frac{height}{stacks}$$

$$stackHeight\ (i) = stackHeight \times i, \qquad para\ stack\ (i) \wedge 1 \leq i \leq stacks \wedge i \in N$$

2) Relativamente ao ângulo **alpha** ilustrado no cone da figura anterior, apenas necessitamos de saber o número de *slices*, assim:

$$alpha = \frac{2 \times \pi}{slices}$$

3) Para o cálculo do **stackRadius** é necessário ter em conta o princípio matemático da semelhança de triângulos - "um tipo de relação que é estabelecida entre triângulos quando eles possuem os lados proporcionais e os ângulos congruentes.". No nosso caso, este exemplo aplica-se entre o triângulo formado pelo **radius** e **height** do cone**,** com os triângulos formados em cada *stack* a partir da **stackHeight** ponderada (desde a base da *stack* até ao vértice do cone) com a nossa incógnita, o **stackRadius.** Matematicamente e com o auxílio da seguinte figura isto traduz-se em:

$$stackRadius = \frac{radius * \big(height - (stackHeight * i)\big)}{height},$$

$$0 \leq i < stacks \wedge i \in N$$

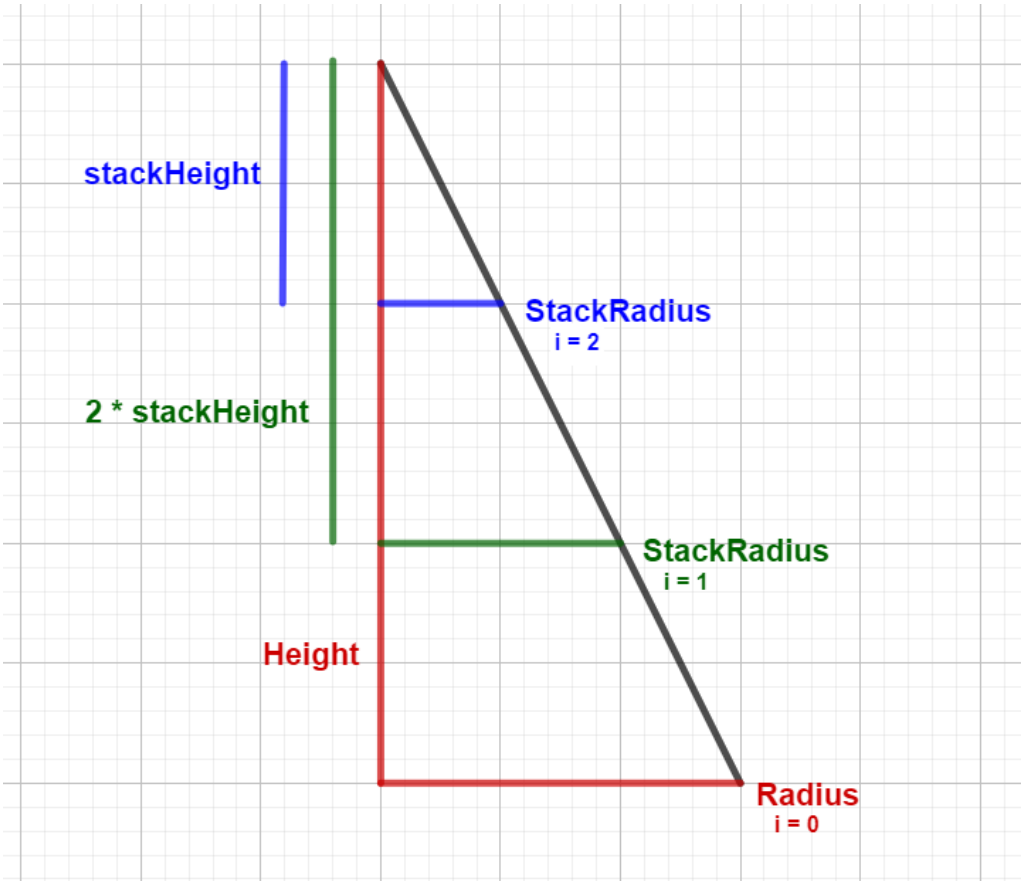


Figura 9: Cálculo da semelhança de triângulos

Estando agora munidos dos parâmetros necessários, podemos dar início à construção do cone.

O algoritmo é simples, para cada *stack* e *slice* e suas correspondentes sucessoras, iremos calcular os 4 pontos do plano por elas formado. Desta forma apenas bastará, encontrados os pontos, ordená-los segundo a regra da mão direita para obter triângulos capazes de representar um cone. De seguida encontra-se o pseudocódigo do algoritmo referido:

```
stackHeight = height / stacks
Para cada stack i {
    stackRadius = (radius * ( height - ( stackHeight * i ))) / height
    stackRadius2 = (radius * ( height - ( stackHeight * ( i+1 ))) / height

    Para cada slice j {
        alpha = ((2 * π) / slices) * j
        alpha2 = ((2 * π) / slices )* j+1

        p1x = cos(alpha) * stackRadius
        p1y = stackHeight * i
        p1z = -sin(alpha) * stackRadius

        p2x = cos(alpha2) * stackRadius
        p2y = stackHeight * i
        p2z = -sin(alpha2) * stackRadius

        p3x = cos(alpha) * stackRadius2
        p3y = stackHeight * (i + 1)
        p3z = -sin(alpha) * stackRadius2

        p4x = cos(alpha2) * stackRadius2
        p4y = stackHeight * (i + 1)
        p4z = -sin(alpha2) * stackRadius2
    }
}
```

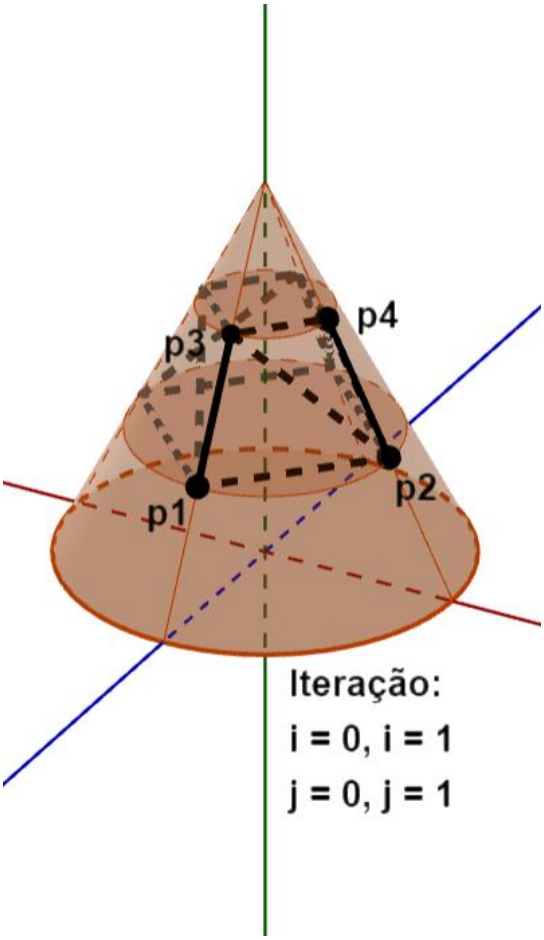


Figura 10: Pontos de uma slice de um cone

Para gerar os triângulos do exemplo acima poderíamos usar a ordem p3 p1 p2 e p3 p2 p4, respetivamente.

Tal como foi indicado anteriormente, antes de guardar os vértices do ficheiro .3d, indicamos o número de vértices presentes no cone. Para isso podemos usar a seguinte expressão matemática:

$$Nº\,Vértices = (2 * slices * stacks) * 3$$

A expressão para construir um cone é **cone radius height stacks slices file**. Por fim o resultado obtido foi o seguinte:

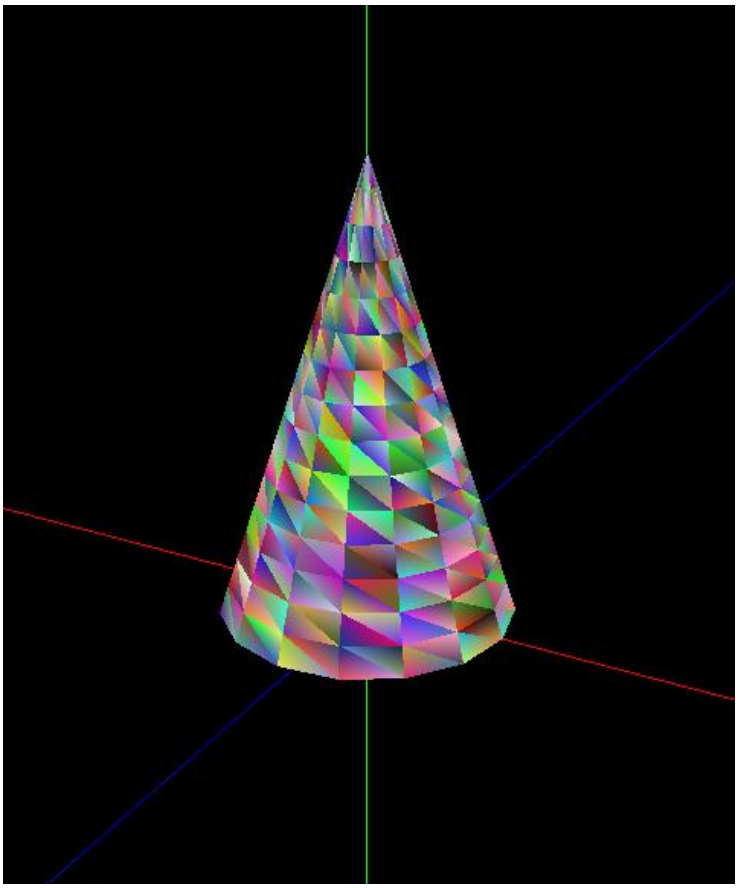


Figura 11: cone 1 3 15 15 cone.3d

BOX

Para a construção da caixa criou-se a função **creatBox** que recebe como parâmetros a **largura**, **altura** e **profundidade** da caixa, isto é, o comprimento no eixo x, y e z respetivamente. Para além disso a função também recebe opcionalmente um parâmetro **edge** que vai dividir cada uma das medidas da caixa em n partes. Caso não exista nenhum parâmetro edge então a função assume que a caixa não será dividida, ou seja, edge igual a 1. Finalmente, o último parâmetro diz respeito ao ficheiro onde vão ser colocados os pontos que formam a caixa.

1ª Parte – Inicialização de variáveis

A construção da caixa inicia-se com o cálculo do número total de vértices necessários, através da expressão apresentada:

$$nrVertices = edge^2 * 2 * 3 * 6 = edge^2 * 36$$

nº de triângulos por divisão; nº de faces; nº de divisões por face; nº de pontos por triangulo

De seguida, calcula-se quais os valores das coordenas x, y e z que se encontram no extremo positivo a partir das medidas da caixa, de acordo com a seguinte expressão:

$$coordenada = \frac{medida}{2}, medida \in \{largura, altura, profundidade\}$$

Finalmente, calcula-se a distância a cada ponto nos eixos respetivos através do parâmetro medida e do número de edge especificado, obtendo a mesma estrutura em todos os eixos do referencial 3D de acordo com o exemplo da figura 12 (edge = 2).

$$distancia = \frac{medida}{edge}, medida \in \{largura, altura, profundidade\}$$

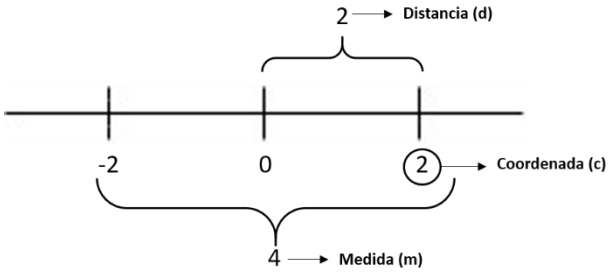


Figura 12: Configuração dos Eixos

2ª Parte – Criação dos planos

Na segunda parte da criação da caixa, criam-se os vários planos segundo as variáveis inicializadas na primeira parte e segundo a posição das faces no referencial. Na figura 13 conseguimos ver uma caixa em que as faces do plano XY estão a vermelho. Vemos que todos os pontos da face 1 tem a mesma coordenada positiva de z ($\text{profundidade}/2$) e que os pontos da face 2 tem a mesma coordenada negativa de z ($-\text{profundidade}/2$). Este formato é repetido para todas as faces do cubo segundo se encontra na tabela 1.

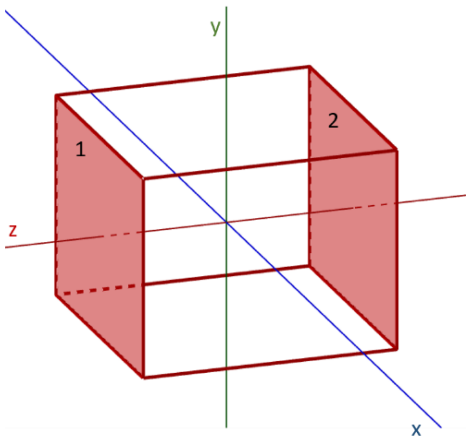


Figura 13: Caixa

BOX	PLANO XY	PLANO XZ	PLANO YZ
FACE 1	Z = profundidade/2	Y = $altura/2$	X = $largura/2$
FACE 2	Z = −profundidade/2	Y = $-altura/2$	X = $-largura/2$

Tabela 1: Coordenadas nos planos

A partir da semântica expressa em cima usou-se a função **buildPlanes** que cria todos os pontos necessários à construção dos vários triângulos que formam as faces da caixa. Os valores passados como parâmetros para a função representam: as coordenadas de x, y e z do ponto inicial ; as distâncias entre os pontos nos vários eixos, as edges e um identificador que identifica qual o plano a construir.

3ª Parte – Algoritmo para a construção das faces

O algoritmo da criação das faces inicia-se com o cálculo do número de pontos necessários para a construção de uma face, identificada pela expressão abaixo.

$$npoints = edge^2 * 2 * 3 = edge^2 * 6$$

De seguida, inicializa-se as variáveis usadas de acordo com o identificador recebido que vai determinar o plano da face a criar. Além disso, determina-se a mudança inicial entre pontos, isto é, qual será eixo que vai variar entre o ponto inicial e o segundo ponto. Na tabela 2 encontra-se associação entre o identificador, o plano e a mudança inicial.

IDENTIFICADOR	1	2	3
PLANO	PLANO XY	PLANO XZ	PLANO YZ
MUDANÇA INICIAL	Sobre o eixo do X	Sobre o eixo do Z	Sobre o eixo do Y

Tabela 2: Associação entre os planos e os valores

O algoritmo vai ser inicializado no ponto positivo mais extremo, isto é, o ponto em que as duas coordenadas que vão variar são positivas e iguais ao valor ***coordenada*** determinada na 1ª parte. No exemplo de uma face no plano XY (figura 14) o ponto inicial seria o A. Para todas as faces em que a coordenada que se mantém constante é positiva (coordenada z no plano XY, por exemplo), os pontos que formam os triângulos têm de ser criados no sentido anti-horário de forma a cumprir com a regra da mão direita (figura 14). Para as faces cuja coordenada constante é negativa os pontos têm de girar no sentido horário (figura 15), uma vez que estas faces não se podem ver na perspetiva em que são desenhadas. Como forma de determinar no algoritmo se estamos a aplicar o método da figura 14 ou o da 15 adicionamos uma variável **order** que será igual a -1 caso seja no sentido anti-horário e igual a 1 caso seja no sentido horário.

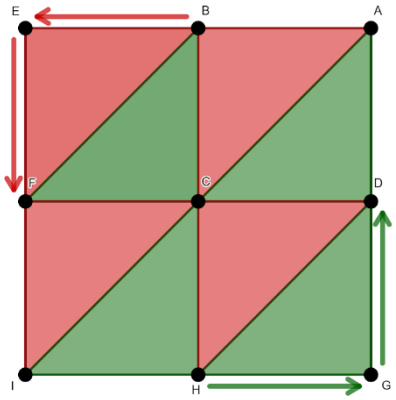


Figura 14: Rotação no sentido anti-horário (Plano XY)

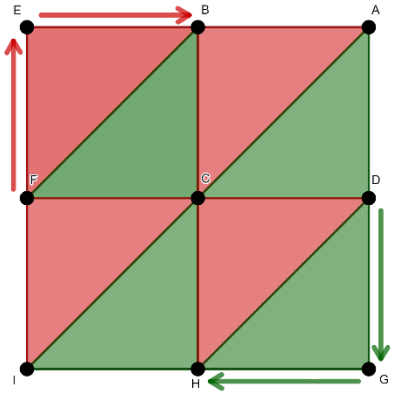


Figura 15: Rotação no sentido horário (Plano XY)

Nestes moldes, para um plano que é estruturado de acordo com a figura 14, os pontos vão ser criados segundo a ordem: A-B-C-C-D-A-B-E-F-etc. Na ordem apresentada podemos ver que o vértice C terá de ser duplicado de modo a iniciar o próximo triângulo. Na figura 16 encontra-se a ordem de criação para faces estruturadas como a da figura 14.

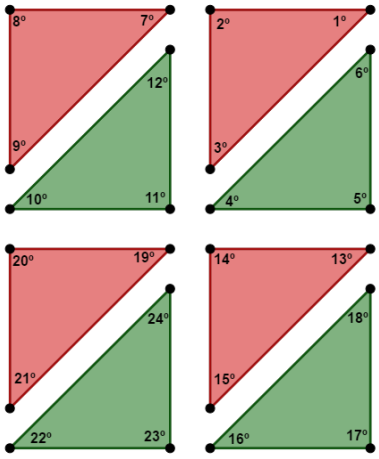


Figura 16: Ordem de criação dos pontos em sentido anti-horário

Tal como podemos observar, o modo de criação dos pontos dos triângulos vai variando entre horizontal (A-B na figura 14) e vertical (B-C na figura 14). Devido a este facto, usou-se a variável **signal** que a cada iteração vai alterar o modo como são formados os pontos – se percorremos os pontos horizontal (signal = 1) então na próxima iteração os pontos vão ser percorridos na vertical (signal = -1), e vice versa. Após percorrermos um triângulo (exemplo o triângulo vermelho ABC) o próximo triângulo a ser formado (o triângulo verde CDA) terá valores simétricos ao triângulo anterior, tal como está representado na tabela 3, sendo dx e dy a distância entre os pontos no eixo x e y, respetivamente.

Triangulo	**Formação dos pontos**	
	Horizontal (exemplo x para o plano XY)	**Vertical** (exemplo y para o plano XY)
Vermelho	1: x = x - dx	2: y = y - dy
Verde	3: x = x + dx	4: y = y + dy

Tabela 3: Formação dos Pontos

Para faces criadas segundo a ordem da figura 14 a mudança inicial de eixo (tabela 2) equivale à formação de pontos segundo a horizontal (tabela 3). Para faces criadas segundo a ordem da figura 15, a mudança inicial equivale à formação de pontos segundo a vertical.

Quando forem formados todos os triângulos numa fila da face, o algoritmo vai passar para o ponto mais extremo positivo da próxima linha, no caso da figura 14 seria o vértice D. Para isso vai-se contar quantas vezes chegamos ao fim da linha. Quando essa contagem atingir um certo valor passamos para a próxima linha.

Todos os pontos formados em cada iteração vão ser colocados numa matriz de *double* através da função **addPoints**, seguido de uma função que passa a matriz de *double* para uma *string* - **buildTriangles**. Este algoritmo é análogo tanto para faces cujos triângulos sejam formados no sentido horário como no anti-horário. Abaixo está apresentado um pseudocódigo do algoritmo de construção de cada face da caixa.

```
Identificação da face
Se (coordenada constante positiva) então
        signal = 1
        order = 1                       //determina quando é que é fim de linha
Senão então
        signal = -1
        order = -1                      //determina quando é que é fim de linha
n = 0;                                  //conta o número de pontos do triangulo
triangle = 1;
count = 0;

Para i = 0 até i < npoints faz-se i++  {  // para cada ponto i
        n++
        Se (atingirmos o fim da linha) então
                count++
        Se (count == 3) então
                mudamos para a próxima linha
        Adicionamos o ponto á matriz
        Se (n==3) então                 //um triangulo esta construído
                Se (ponto de mudança entre triângulos) então   //exemplo vértice C fig. 14
                        Adicionamos o ponto á matriz
                        n = 1
                        triangle = 2
                Senão então
                         n = 0
                        triangle = 1
        Se (está construído um quadrado completo) então
                signal=signal*(-1)
        Se (triangle == 1) então
                Se (signal==1) então
                        Muda a coordenada num eixo        //exemplo tabela 3 célula 1
                Senão então
                        Muda a coordenada noutro eixo    //exemplo tabela 3 célula 2
        Senão então
                Se (signal==1) então
                        Muda a coordenada num eixo     //exemplo tabela 3 célula 3
                Senão então
                        Muda a coordenada noutro eixo //exemplo tabela 3 célula 4
        signal = signal*(-1)

}
```

No final conseguimos obter a seguinte representação de uma caixa, gerada através do input **box x y z edges file**:

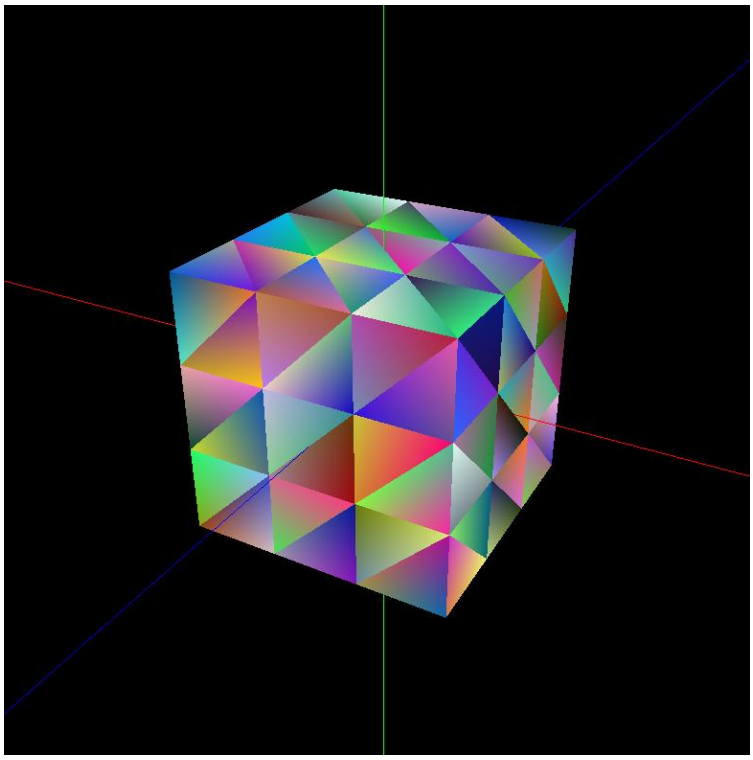


Figura 17: box 3 3 3 3 box.3d

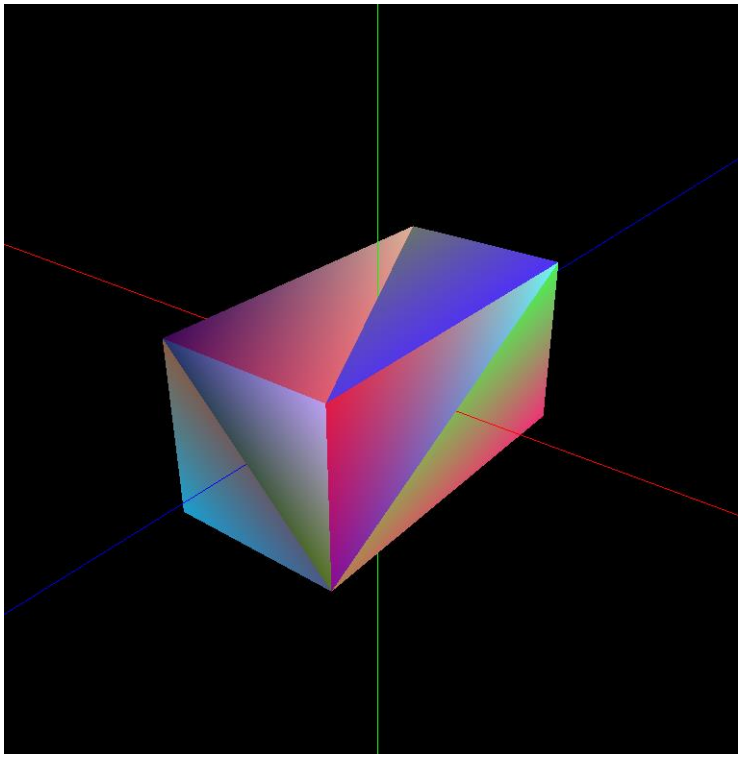


Figura 18: box 2 2 4 box.3d

PLANE

Para a construção do plano foi usada a função **creatPlane** que ao receber como parâmetros o lado (*l*) do plano e o ficheiro .3d, vai criar um plano (quadrado de lado *l*) no plano XZ, armazenando os pontos gerados no ficheiro recebido.

Numa primeira fase, determina-se o número de pontos necessários à construção do plano, que neste caso é sempre 6. De seguida, calcula-se quais os valores das coordenadas x e z que se encontram no extremo positivo a partir do lado recebido de acordo com a seguinte expressão:

$$coordenada\ x = \frac{lado}{2}, \qquad coordenada\ y = 0, \qquad coordenada\ z = \frac{lado}{2}$$

Como construímos um quadrado centrado na origem num plano XZ então a coordenada de y será, em todos os pontos 0. Para a geração dos pontos utilizou-se a função **buildPlanes** que recebe como parâmetro os valores das coordenadas iniciais (coordenada x, 0, coordenada y), a distância que vai entre cada ponto (neste caso é o valor do lado), o número de edge (para o plano só é necessário construir 2 triangulos logo edge = 1) e finalmente qual o plano a criar (plano XZ equivale a id = 2, segundo a tabela 2). O algoritmo de criação dos pontos está explicito na construção da Box.

No final conseguimos obter a seguinte representação de um plano, gerada através do input **plane side file**:

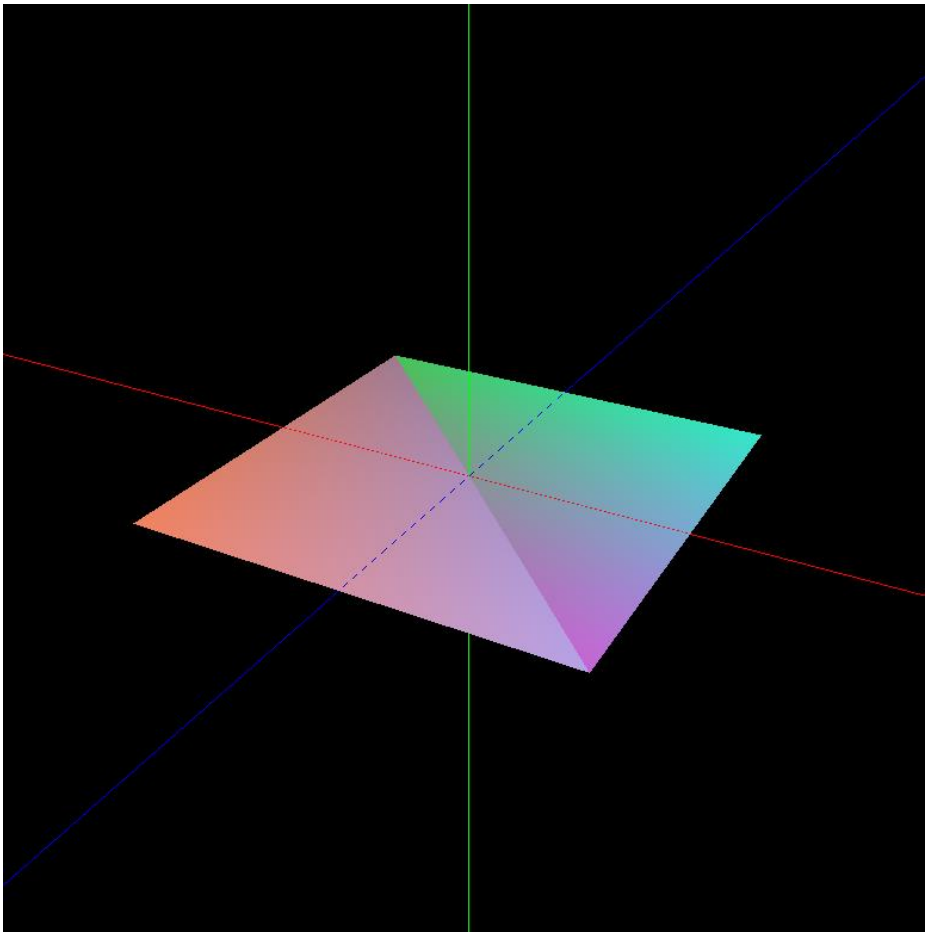


Figura 19: plane 4 plane.3d

CYLINDER - EXTRA

De modo a complementar o trabalho elaborado nesta fase, consideramos conveniente adicionar uma outra primitiva: o cilindro.

A primitiva cilindro implementada permite construir um cilindro de dimensões variadas, a partir do fornecimento ao programa das variáveis raio, altura e *slices*. O algoritmo começa por efetuar o cálculo do número total de vértices dos triângulos constituintes do cilindro. De seguida, é efetuado o cálculo do ângulo de cada *slice.* Estas operações estão apresentadas de seguida:

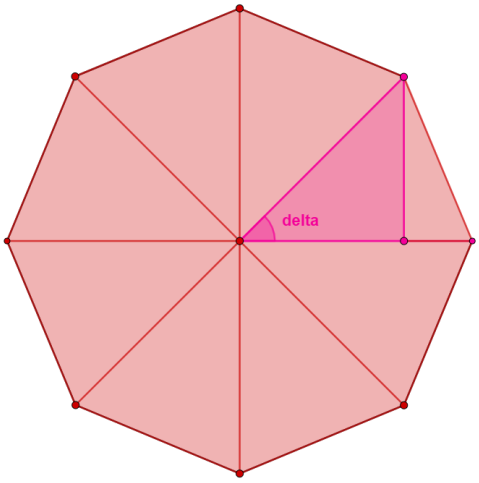
Nr de faces

Lados do triângulo

$$total\ pontos = slices * 2 * 3$$

$$ângulo = \frac{2 \times \pi}{slices}$$

De forma a gerar os diferentes vértices e, consequentemente, os triângulos do cilindro, é efetuado um ciclo *for* que, através de fórmulas trigonométricas, gera as diferentes coordenadas. As componentes y podem ser obtidas de forma direta através da altura do cilindro. Seguem abaixo os cálculos efetuados:



$$x = raio * sen(delta)$$

$$z = raio * cos(delta)$$

$$y = altura/2$$

Figura 20: Cálculo das coordenadas

Cada base do cilindro vai ser dividida em *slices* que irão dar origem a diferentes triângulos. Ao unir os vértices do topo e da base conseguimos obter os triângulos que formam as laterais do cilindro.

O algoritmo explicitado anteriormente pode ser traduzido no pseudocódigo da seguinte página:

delta = (2 * π) / slices;

Para cada slice i {

alfa = i * delta;

nextalfa = delta + alfa;

p1x = radius * sin(alfa);

p1z = radius * cos(alfa);

p2x = radius * sin(nextAlfa);

p2z = radius * cos(nextAlfa);

variáveis top, bottom, side1, side2;

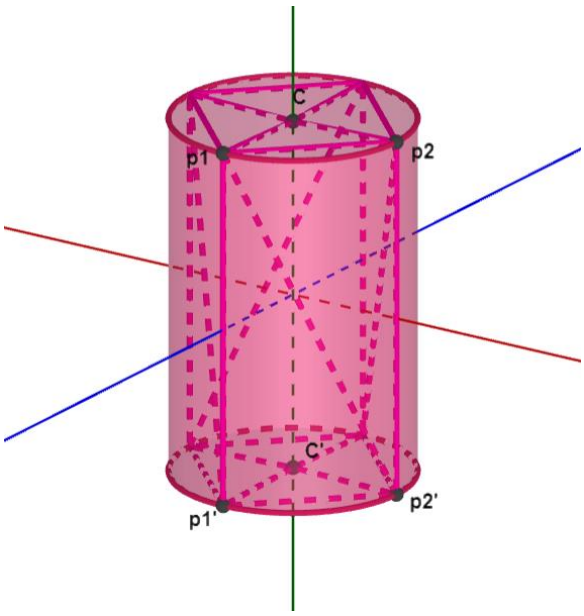


Figura 21: Divisões do cilindro

agrupar os vértices de modo a formar triângulos para a top, bottom, side1, side2

}

Para cada 2 pontos, p1 e p2, pertencentes à mesma *slice*, calculamos o respetivos x e z. Para desenhar os triângulos, basta ter em consideração os valores de y e a regra da mão direita. Desta forma, seguindo a figura 21, poderíamos criar o triângulo do topo (C -> p1 -> p2), assim como o da base (C' -> p2' -> p3'), e também os das laterais (p1' -> p2 -> p1 e p1' -> p2' -> p2).

Através da invocação do comando **cylinder radius height slices file**, são gerados os vértices que irão dar origem à seguinte figura em OpenGL.

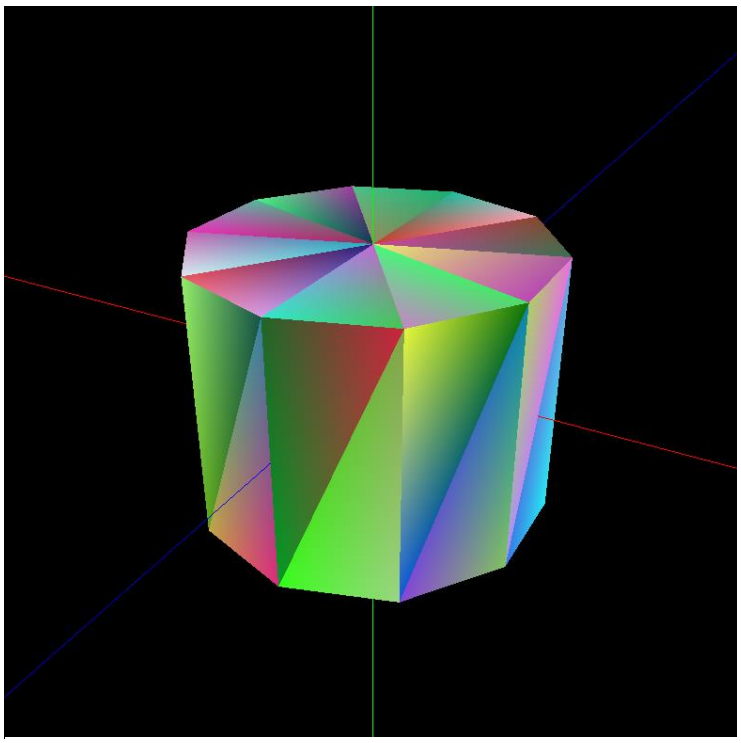


Figura 22: cylinder 2 3 10 cylinder.3d

ENGINE

ESTRUTURA DA APLICAÇÃO

Além da aplicação *Generator*, o programa requer também uma outra: o *Engine*, ou seja, um motor capaz de ler e desenhar triângulos através da informação contida num ficheiro XML.

Esta aplicação começa por aceder a um ficheiro XML com recurso ao tinyXML, que será lido apenas uma vez. De seguida, verifica quais as primitivas lá contidas e lê os ficheiros .3d respetivos. O conteúdo desses ficheiros será guardado em memória para posteriormente ser desenhado.

De forma a armazenar diferentes primitivas em memória, foi necessário recorrer ao uso das classes *Primitive* e *Point*, que possuem diferentes estruturas internas capazes de armazenar os dados adequados. Assim, a classe *Point* possui variáveis de instância que denotam um ponto, ou seja, consegue armazenar as coordenadas x, y e z de cada vértice. No seguimento da criação desta classe, foi também criada a classe *Primitive* que possui uma variável que constitui um vetor de elementos da classe *Point*. Assim, cada elemento da classe *Primitive* será uma das primitivas anteriormente explicitadas (plano, caixa, esfera, cone ou cilindro) e as coordenadas dos triângulos que as formam, estarão armazenadas num vetor. Na seguinte figura estão representadas esquematicamente as diferentes estruturas de dados:

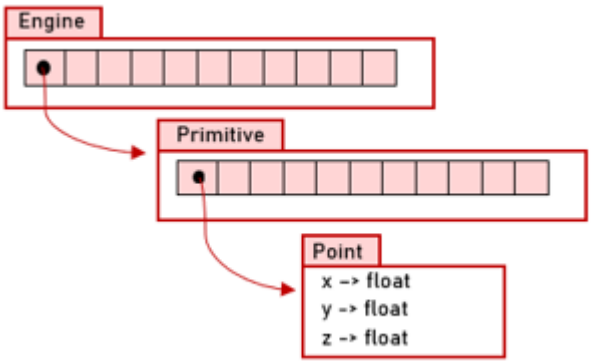


Figura 23: Estrutura de dados

Por último, e analogamente ao *Generator*, também foi utilizada uma *CMakeFile* que permite compilar o projeto e também incluir os diferentes ficheiros utilizados (os ficheiros das classes *Primitive, Point e tinyXML2*).

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. Em baixo, apresentamos dois exemplos de primitivas sobrepostas desenhadas pelo *Engine.*

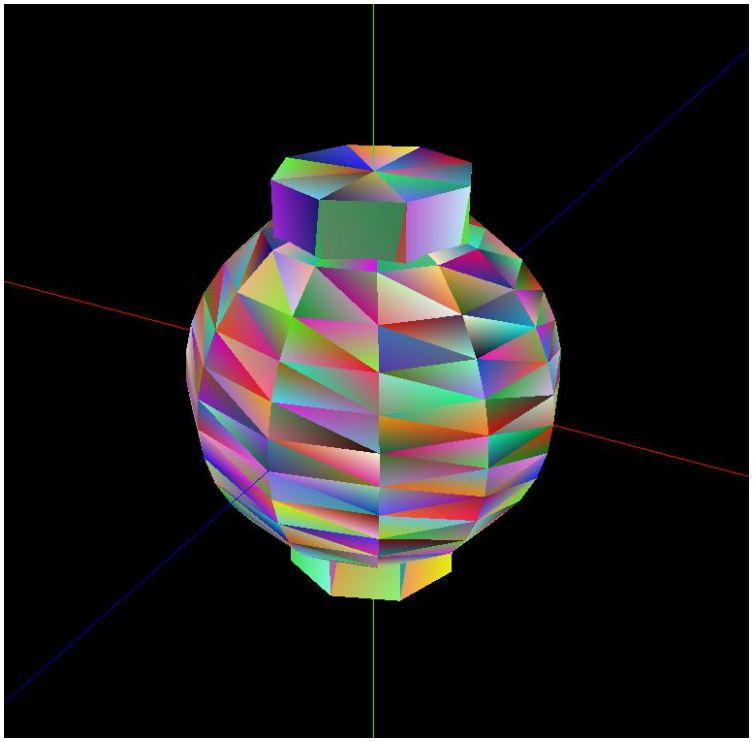


Figura 24: sphere 1 12 12 e cylinder 1 4.5 8

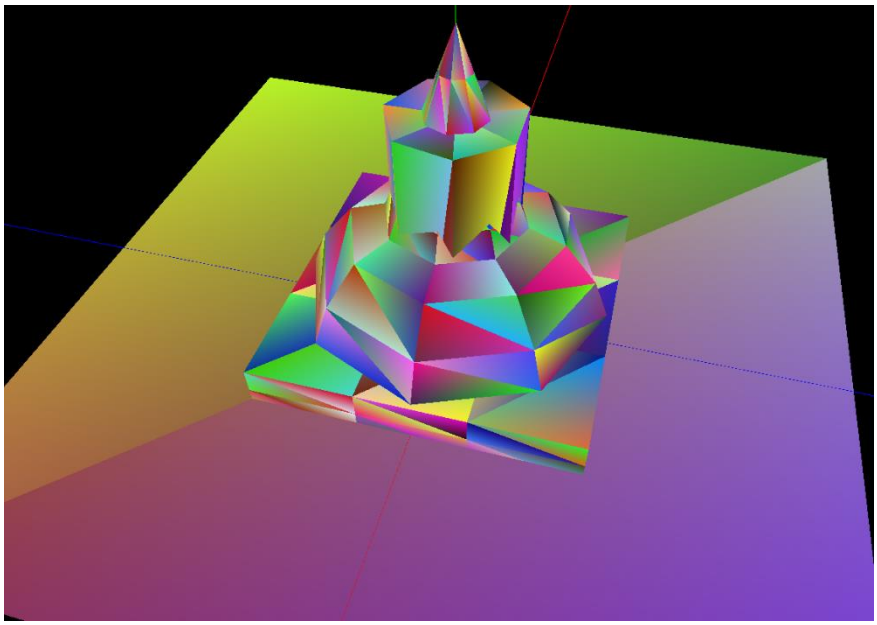


Figura 25: Demo scene

EXTRAS

Na aplicação *Engine* foram implementadas funcionalidades extra, para além daquelas que eram referidas no enunciado.

Consideramos que implementar o movimento da câmara seria um acréscimo positivo ao trabalho realizado. Além disso, também foi relevante incluir cores aleatórias para colorir os diferentes triângulos, de modo a melhor visualizá-los.

CONCLUSÃO

Dada por concluída a 1ª fase do projeto, consideramos relevante efetuar uma análise crítica do trabalho realizado.

Concluido o desenvolvimento do trabalho, notamos que existiram aspetos positivos a realçar, entre eles a preocupação estética relativamente às cores, a criação de uma primitiva extra – o cilindro, movimento da câmara e também código fiável que inclui tratamento de erros.

Por outro lado, também existiram algumas dificuldades sentidas, tais como colocar o programa funcional em qualquer diretoria e sistema operativo e também na construção da caixa com *edges* devido à complexidade no cálculo dos pontos. Apesar de terem requirido atenção extra, os problemas encontrados foram 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.