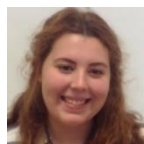


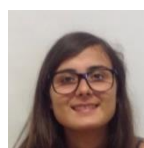
**Universidade do Minho**  
Escola de Engenharia

Computação Gráfica  
**Relatório do projeto prático**  
**Fase I**

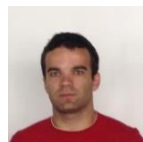
**Grupo de Trabalho**



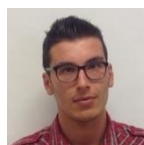
Ana Esmeralda Fernandes  
A74321



Bárbara Nadine Oliveira  
A75614



João Paulo Ribeiro Alves  
A73542



Miguel Dias Miranda  
A74726

Mestrado Integrado em Engenharia Informática  
março de 17



## Conteúdo

1. Introdução.....	3
2. Projetos Desenvolvidos.....	4
2.1 Aplicação Gerador .....	4
2.1.1 Algoritmo para gerar um plano .....	5
2.1.2 Algoritmo para gerar uma caixa.....	6
2.1.3 Algoritmo para gerar um cilindro .....	8
2.1.3 Algoritmo para gerar uma esfera.....	10
2.1.4 Algoritmo para gerar um cone.....	12
2.2 Aplicação Motor3D.....	15
3. Instruções de funcionamento dos Projetos .....	16
3.1 Gerador.....	16
3.2 Motor3D .....	16
3.2.1 Interação com os modelos desenhados .....	17
4. Conclusão.....	18
5. Referências.....	19
6. Anexos.....	20
6.1 Código fonte para gerar vértices de um cone .....	20
6.2 Código fonte para gerar vértices de uma esfera .....	21
6.3 Código fonte para gerar vértices de um cilindro .....	21
6.4 Código fonte para gerar vértices de uma caixa .....	22
6.5 Código fonte para gerar vértices de um plano .....	24



## 1. Introdução

Este relatório visa apresentar as decisões tomadas na realização da primeira fase do trabalho prático da Unidade Curricular de Computação Gráfica. Procuramos assim justificar todas as considerações feitas na formulação do problema, na elaboração dos algoritmos de geração de vértices para os determinados modelos e o processo de construção e desenho das figuras através do input de um ficheiro previamente criado.

Toda a modulação do problema foi feita com recurso à ferramenta de programação Visual Studio e à linguagem C++, abordada nas aulas práticas da UC. Numa primeira fase será descrito o processo de geração do ficheiro de vértices e posteriormente explicado o modo de desenvolvimento do dito motor3D que irá desenhar as figuras.

Para esta primeira fase, foram implementados os algoritmos para gerar os vértices de um:

- Plano  
Quadrado em XZ centrado na origem
- Caixa  
Modelo tridimensional com largura, comprimento, altura, número de fatias e stacks
- Cilindro  
Modelo de um cilindro, com base no seu raio, altura, número de fatias e stacks
- Esfera  
Esfera centrada na origem, com raio, número de fatias e stacks
- Cone  
Modelo de um cone, identificado pelo raio da base, altura, número de fatias e stacks.

Os algoritmos criados para a geração dos vértices dos referidos modelos serão esclarecidos textualmente e com esquemas no próximo capítulo deste relatório. O código fonte dos ficheiros .h e .cpp destas classes necessárias para o projeto estão disponibilizadas na seção Anexos do presente relatório.

## 2. Projetos Desenvolvidos

### 2.1 Aplicação Gerador

A aplicação responsável por gerar um ficheiro com a informação sobre os vértices de um determinado modelo, caracteriza-se por conter o ficheiro com a função *main* e mais três pares de ficheiros, que estruturam as classes *Modelo*, *Vértice* e *Triangulo*, nos seus ficheiros .h e .cpp, respetivamente.

A função *main*, limita-se por passar os seus argumentos de entrada a uma função responsável por fazer o parsing destes mesmos. Esta função, designada por *parseParametros*, vai simplesmente determinar a partir do elemento dado na posição 1 (“plane”, “box”, etc.), qual o conjunto de parâmetros relevantes introduzidos e converte-los para o tipo apropriado. Tendo estes dados validados, ligeiramente diferentes para cada tipo de figura, irá simplesmente chamar funções da classe *Modelo*, que irão construir o modelo desejado. Este modelo é gerado tendo o seu ponto de gravidade na origem do referencial XYZ.

Na classe *Modelo*, com recurso a uma variável do tipo *vector<Triangulo>* são guardados o conjunto de *Triangulos* que formarão a figura desejada. Esta variável é escrita e preenchida pelas funções que geram as figuras, denominadas por *plane*, *box*, *cilindro*, *sphere* e *cone*, que conforme o modelo que criam, geram os vértices necessários para compor os triângulos que dão origem a esse modelo. Gerados os vértices, são criados então os triângulos e guardados na variável da classe, privada, que posteriormente será passada à função *escreveModelo*.

Por fim, tudo o que a função *escreveModelo* executa é para cada triângulo da variável da classe *Modelo*, ir a cada um dos seus vértices e escrever as suas coordenadas no ficheiro.

Para a nossa implementação, cada linha do ficheiro será conceptualmente um triângulo, e as coordenadas de cada vértice, serão separadas por o caracter virgula ‘,’. Os triângulos são criados tendo em consideração a regra da mão direita, para garantir posteriormente a sua correta visualização no motor3D que irá desenhar o modelo.

A classe *Triangulo* contém apenas como variáveis privadas os três vértices que o formam, e funções essenciais como o construtor da classe e métodos *get*. De igual modo, também a classe *Vertice* apresenta apenas como variáveis privadas as coordenadas (doubles) que serão o trio de pontos que no espaço 3D dão origem ao vértice, e como funções o seu construtor e métodos *get*.

Os algoritmos para a criação dos modelos serão explicados nas secções futuras do relatório e o código fonte destas classes disponibilizado na secção Anexos.

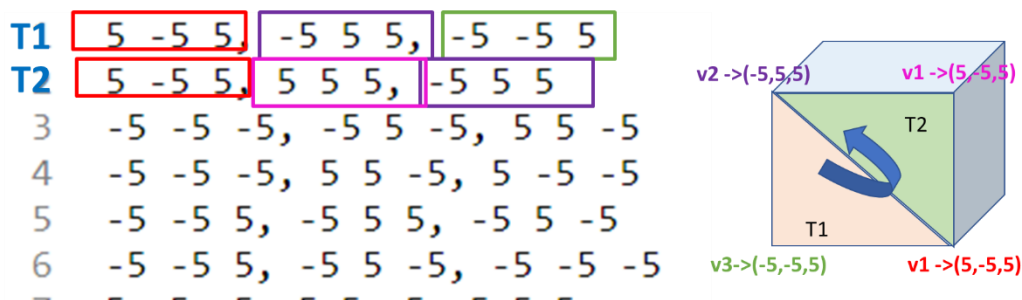


Figura 1 - Exemplo do output escrito para um ficheiro, com as informações necessárias para criar um cubo com lado = 10. Cada linha contém as informações para um triângulo, sendo que no exemplo da imagem cada face da caixa só é definida com dois triângulos.

### 2.1.1 Algoritmo para gerar um plano

**Cabeçalho da função:** `void plane(double ladoSqr, char* nameFile3d);`

**Parâmetros de entrada:** ladoSqr: valor do lado do plano; char\* nameFile3d: nome do ficheiro que será criado com os vértices do plano.

A função que gera as informações dos vértices de um plano, designada por *plane*, limita-se a determinar os 4 vértices necessários para a criação do plano e posteriormente produzir dois triângulos que o irão formar. De referir que sendo um plano, centrado na origem do referencial XYZ e com a variável Y constante (Y=0), os vértices apresentam simetria face ao eixo dos X.

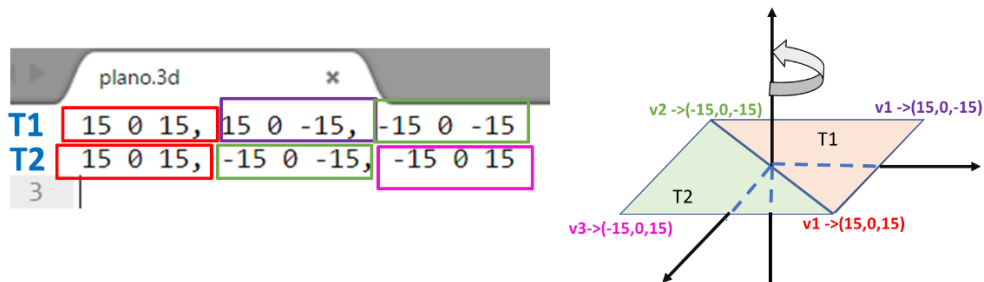


Figura 2 - Exemplo do output escrito para um ficheiro plano.3d, com as informações necessárias para criar um plano com lado = 30. De notar que o plano é desenhado com as faces dos triângulos voltadas para cima, segundo a regra da mão direita.

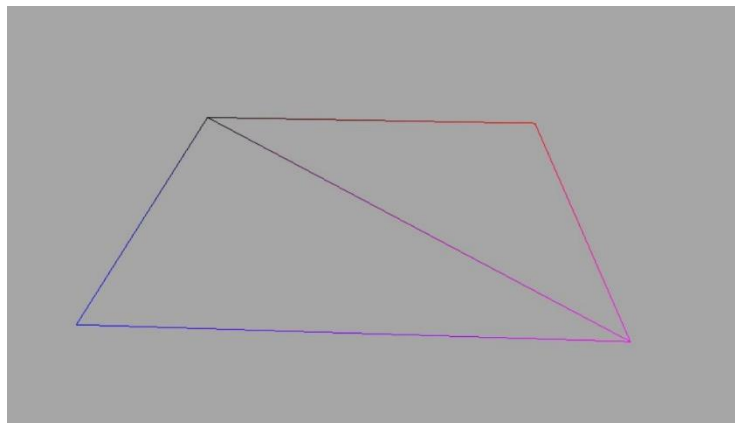


Figura 3 – Janela criada com o input escrito gerado para a geração do plano de lado 10. De notar que os ângulos de visualização foram alterados para ser possível ver o plano deste modo.

### 2.1.2 Algoritmo para gerar uma caixa

**Cabeçalho da função:** `void Modelo::box(double largura, double compri, double altura, double nDiv, char* nameFile3d);`

**Parâmetros de entrada:** double largura, compri e altura que representam os valores de largura, comprimento e altura do cubo, respetivamente; double nDiv: representa o número de divisões pretendidos para criar um face; char\* nameFile3d: nome do ficheiro que será criado com os vértices da caixa.

Para determinar os vértices necessários para formar um cubo em três dimensões implementou-se o seguinte algoritmo:

É usado um ciclo que irá percorrer o eixo dos Y, fazendo as diversas stacks. Dentro deste, existem dois ciclos, um para iterar o eixo dos X, determinando os vértices dos triângulos das faces frontal e traseira do cubo a cada stack, e o outro ciclo para iterar o eixo dos Z, calculando os vértices das faces laterais esquerda e direita.

Terminada esta fase, temos então um cubo onde apenas faltarão as faces superiores e inferiores. Novamente com recurso a dois ciclos aninhados, um para percorrer o eixo do X e outro para percorrer o eixo dos Z (a ordem podia ser arbitrária), e fixando a variável Y correspondente à coordenada constante nas faces superiores e inferiores (são planos em XZ), são gerados os vértices e respetivos triângulos que irão cobrir estas duas faces do cubo.

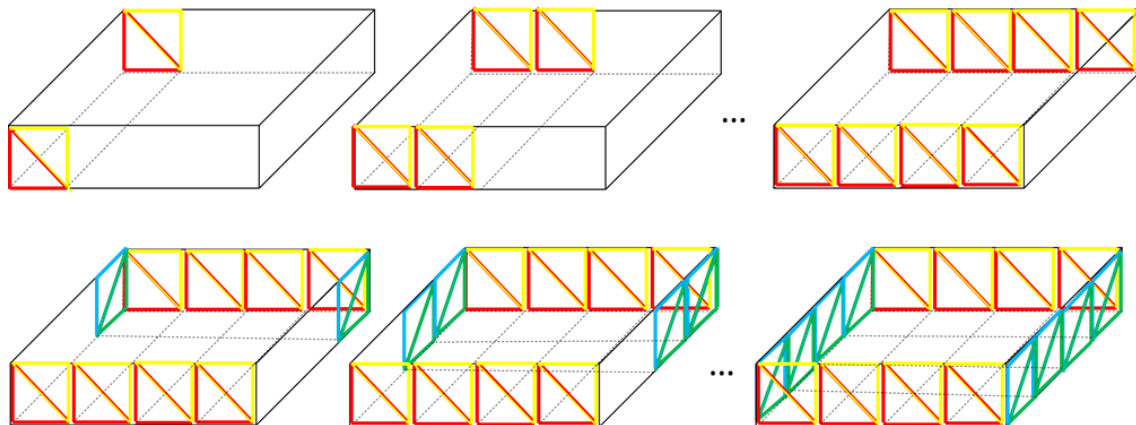


Figura 4 - Representação esquemática do algoritmo de criação de um cubo com 4 divisões na primeira iteração do ciclo principal – stack 0, percorrendo primeiro o eixo dos X e posteriormente o eixo do Z.

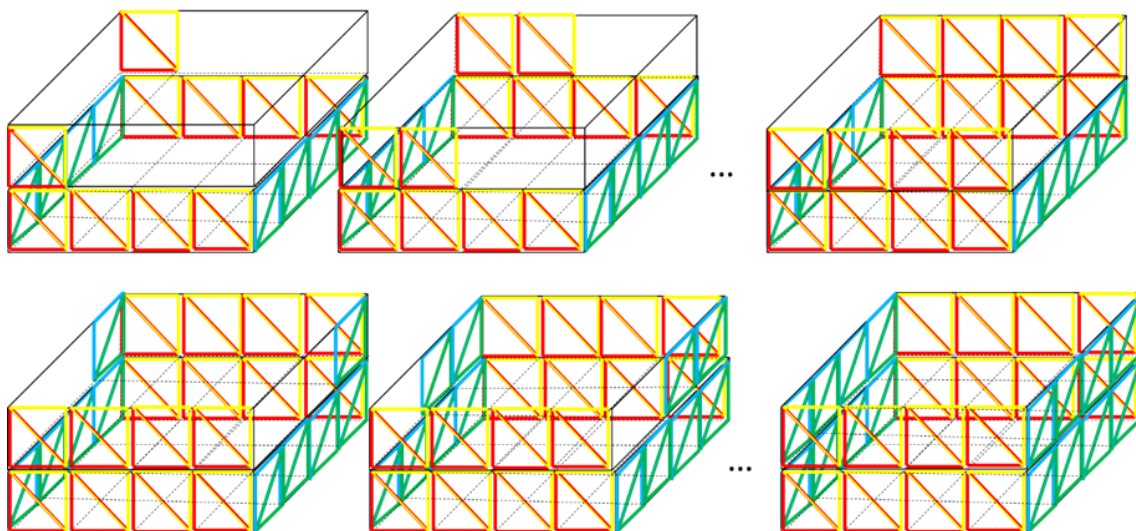


Figura 5 - Representação esquemática do algoritmo de criação de um cubo com 4 divisões na segunda iteração do ciclo principal – stack 1, percorrendo primeiro o eixo dos X e posteriormente o eixo do Z.

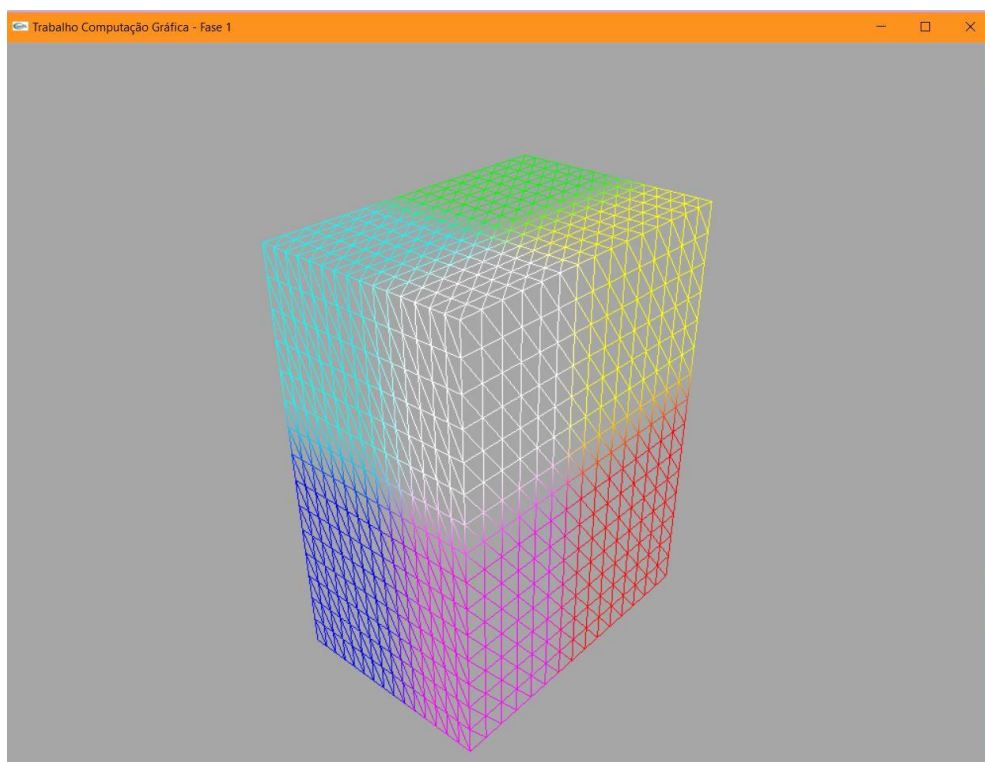


Figura 6 – Resultado do modelo, no motor desenvolvido, de uma caixa previamente construída com parâmetro de divisão igual a 15, comprimento 4, largura 6 e altura 8.

### 2.1.3 Algoritmo para gerar um cilindro

**Cabeçalho da função:** `void Modelo::cilindro(double raio, double altura, int fatias, int stacks, char* nameFile3d);`

**Parâmetros de entrada:** double raio e altura, que indicam os valores do raio e altura do cilindro; int fatias e stacks, que indica o número de fatias e stacks a aplicar na criação do cilindro; char\* nameFile3d: nome do ficheiro que será criado com os vértices do cilindro.

Desenvolvidos os algoritmos para a geração dos vértices de um plano e de um cubo, foi tomada a iniciativa de dar a possibilidade de também gerar um cilindro, recebendo parâmetros como o raio, altura, número de fatias e número de stacks do modelo.

Para o construir, foi apenas necessário implementar dois ciclos for aninhados, uma para iterar cada uma das stacks, vistas como “anéis” e o outro ciclo para iterar o ângulo de 0 a  $2\pi$  em cada nível, que vai aproximar a circunferência desse anel em  $n$  lados, onde  $n$  é o número de fatias do parâmetro de entrada. Assim, tendo noção que temos anéis com um raio constante, por se tratar de um cilindro e que os vértices da camada inferior diferem apenas na coordenada Y dos vértices da camada superior, facilmente o algoritmo foi estruturado. Ainda dentro do ciclo interno, quando o valor de  $y$ , no ciclo externo, foi igual ao valor de  $y$  na base ou topo do cilindro, serão em acréscimo desenhados os triângulos da base e do topo, otimizando assim o código sem necessidade de um novo ciclo aninhado.

Apesar de poderem ter sido usadas coordenadas cilíndricas para representar esta figura, foi usada a notação cartesiana onde o valor de da coordenada X é  $\text{raio} \cdot \cos(\alpha)$  e Z toma o valor de  $\text{raio} \cdot \sin(\alpha)$ .

Considerando um cilindro com altura  $H$ , raio  $R$ , parâmetro de stacks  $S$  e de fatias  $F$ , o parâmetro  $n$ , que representa a altura de cada anel, será  $H/S$ , e o ângulo  $\alpha$ , usado na iteração que constrói parte da fatia  $i$ , será igual a  $i \cdot \theta$ , onde  $\theta$  é igual a  $2 \cdot \pi / F$ , ou seja, o ângulo de uma fatia. As seguintes figuras procuram ilustrar este raciocínio.

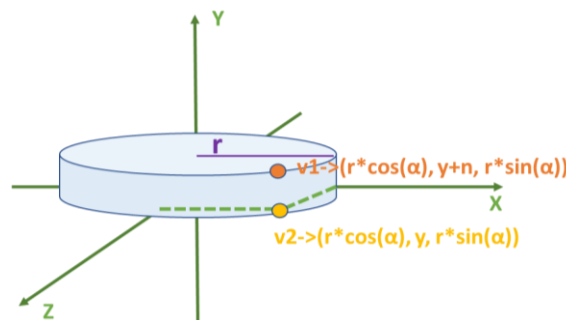


Figura 7 – Exemplo das coordenadas de dois pontos que se encontram sobre circunferências de igual raio, mas a alturas, no eixo do Y, diferentes.



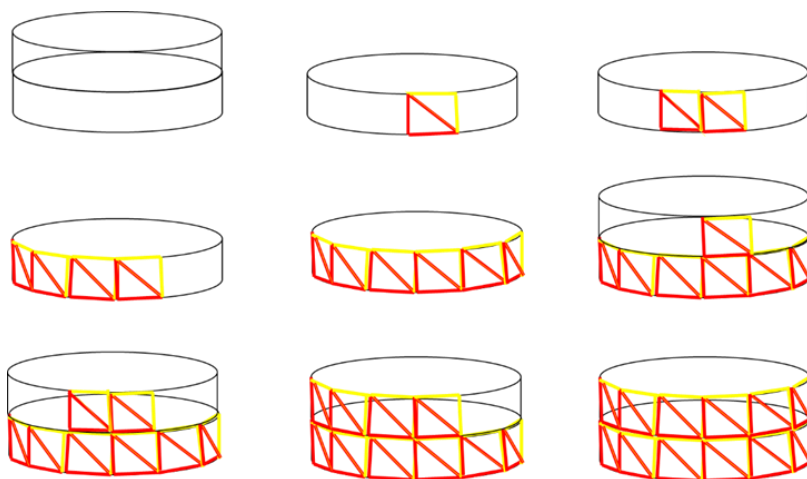


Figura 8 - Esboço do algoritmo de criação de um cilindro com duas stacks e 12 fatias. Primeiro é iterado o anel da primeira stack, compondo os triângulos de cada fatia, rodando entre  $0$  e  $2\pi$ , e posteriormente iteram-se os seguintes níveis, repetindo o processo. Nas imagens não são representados os triângulos da parte traseira, para não ficar muito confuso.

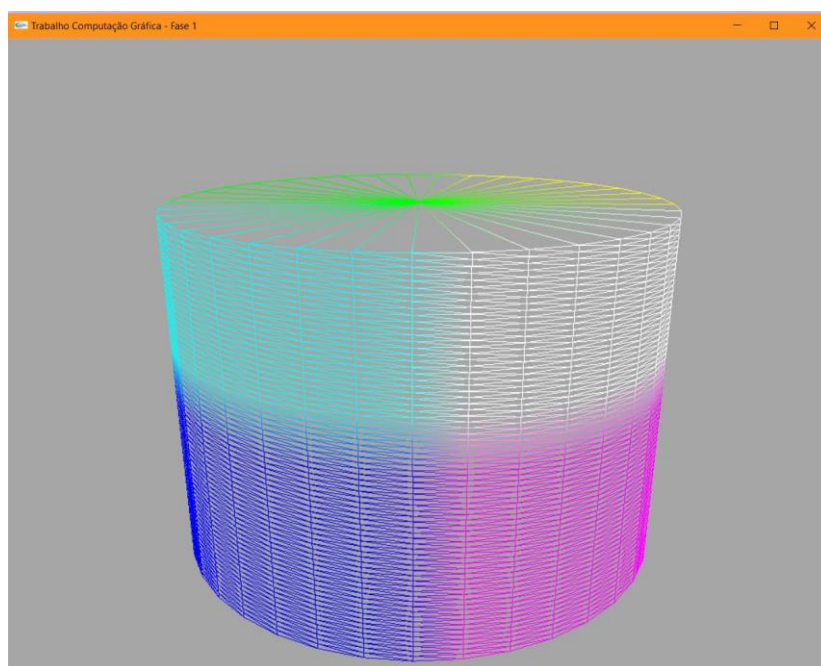


Figura 9 – Exemplo de um cilindro gerado pelo motor, com valor de raio 5, altura 7, número de fatias 37 e número de stacks 70

### 2.1.3 Algoritmo para gerar uma esfera

**Cabeçalho da função:** `Modelo::sphere(double raio, int fatias, int stacks, char* nameFile3d);`

**Parâmetros de entrada:** double raio: valor do raio da esfera, int fatias e stacks, que indica o número de fatias e stacks a aplicar na criação do cilindro; char\* nameFile3d: nome do ficheiro que será criado com os vértices da esfera.

Para implementar um algoritmo de construção de uma esfera, foi usada a noção de coordenadas esféricas, onde qualquer ponto que pertence a uma determinada superfície esférica, pode ser identificado, considerando o referencial do glut, por três parâmetros: o raio  $r$  da esfera, o ângulo  $\alpha$ , em função ao eixo dos X e do ângulo  $\theta$  em função do eixo dos Y.

Assim, utilizando dois ciclos aninhados, um para iterar o eixo dos Y (stacks), começando com o valor de  $\theta$  em  $\pi$  e terminando em  $-\pi$  e, dentro deste ciclo, o outro para iterar as fatias, começando com o ângulo  $\alpha$  em 0 e terminando em  $2\pi$ .

O ângulo de cada fatia será um múltiplo de  $(2\pi / \#Fatias)$  e o ângulo de cada stacks será múltiplo de  $(\pi / \#Stacks)$ .

Como foi utilizada a noção de coordenadas esféricas, os vértices não foram diretamente gerados com o construtor da classe Vertice, mas sim com uma função auxiliar desta classe, que converte as coordenadas esféricas em cartesianas, segundo as regras usadas na imagem seguinte.

Também nas imagens seguintes pode ser visualmente entendido como itera e funciona este algoritmo de criação de uma esfera.

```
Vertice Vertice::esfericas(double raio, double alpha, double beta) {  
    double x = raio * sin(alpha) * cos(beta);  
    double y = raio * sin(beta);  
    double z = raio * cos(alpha) * cos(beta);  
  
    return Vertice(x, y, z);  
}
```

Figura 10 – Função que cria um vértice em coordenadas cartesianas, recebendo como parâmetros as suas coordenadas esféricas, sendo **raio** o raio da esfera, **alpha** o ângulo de cada fatia (iterado de 0 a  $2\pi$  e referente ao eixo dos X) e por fim, o ângulo **beta** (que varia de  $\pi$  a  $-\pi$ , e corresponde ao ângulo de cada stacks).

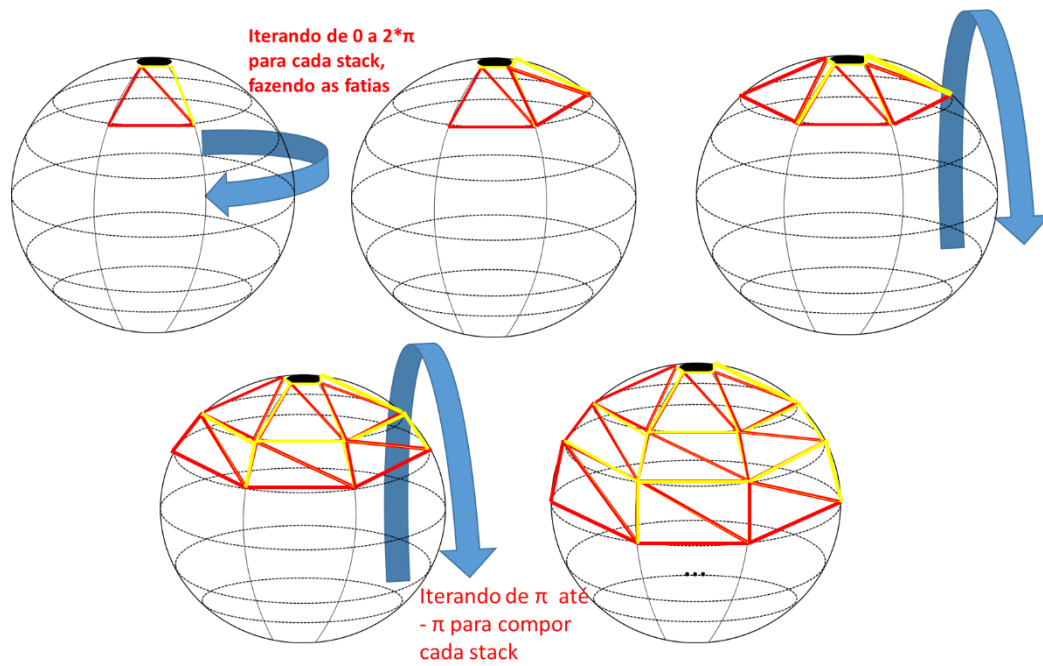


Figura 11 - Esboço da geração dos vértices de uma esfera com 6 fatias e 7 stacks, fazendo cada nível desde a stack superior para a stack inferior.

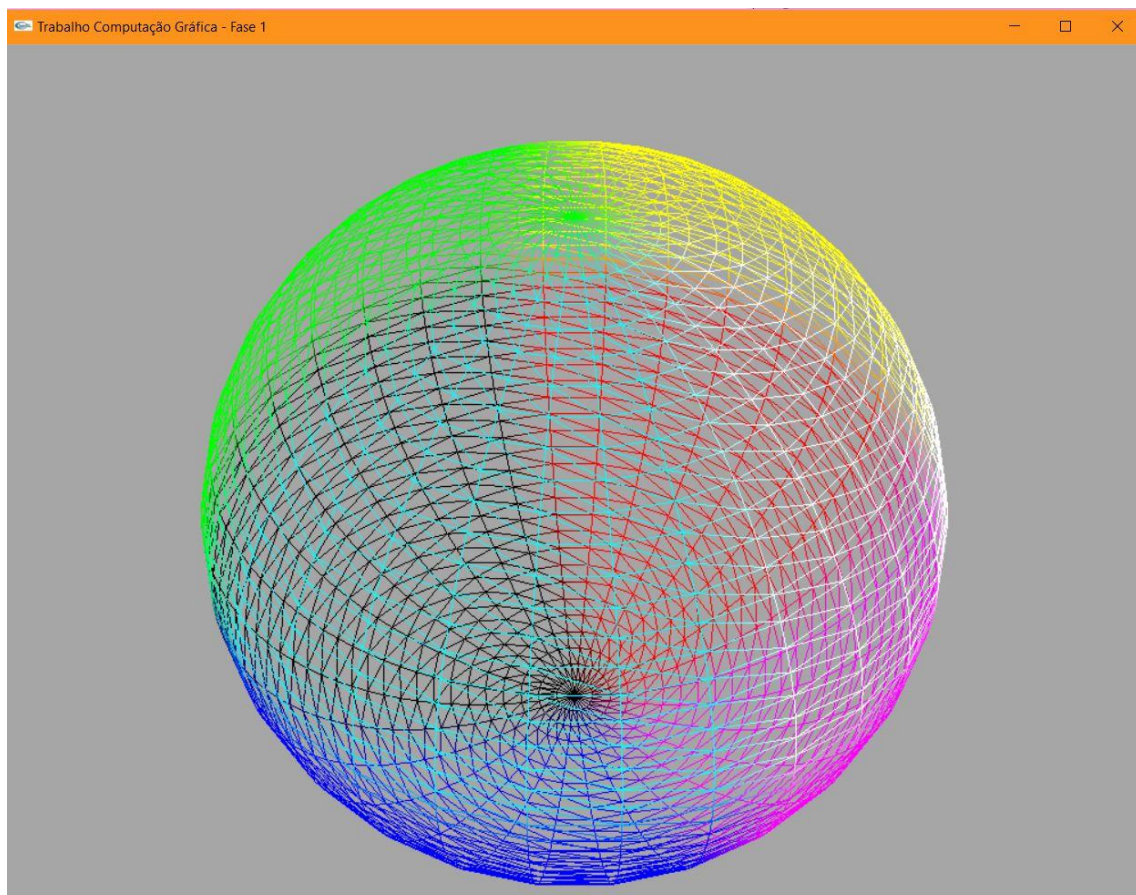


Figura 12 – Exemplo de uma esfera gerada, com raio = 5, número de fatias = 30 e número de stacks = 50.

## 2.1.4 Algoritmo para gerar um cone

**Cabeçalho da função:** `void Modelo::cone(double raio, double altura, int fatias, int stacks, char* nameFile3d);`

**Parâmetros da função:** double raio e altura, que indicam os valores do raio da base e altura do cone; int fatias e stacks, que indica o número de fatias e stacks a aplicar na criação do cone; char\* nameFile3d: nome do ficheiro que será criado com os vértices do cone.

Muito em semelhança ao modo de geração do cilindro, para formar o cone, apenas foi preciso criar um algoritmo com dois ciclos “for” aninhados. O de fora itera as stacks e o ciclo de dentro serve para iterar o ângulo que varia entre 0 e  $2\pi$  em cada nível da stack. Cada stack do cone (em diferentes iterações do ciclo externo) terá um raio diferente, mas este decresce de forma linear, e é calculado conforme as seguintes equações matemáticas:

$$\begin{aligned}\text{raioInf} &= \text{raio} - (\text{raio} * \text{iterY}) / \text{stacks}; \\ \text{raioSup} &= \text{raio} - (\text{raio} * (\text{iterY} + 1)) / \text{stacks};\end{aligned}$$

Sendo, “raio” o raio da base, passada como argumento de entrada, “iterY” o número da iteração em que nos encontramos, que está diretamente relacionada com o número de stacks.

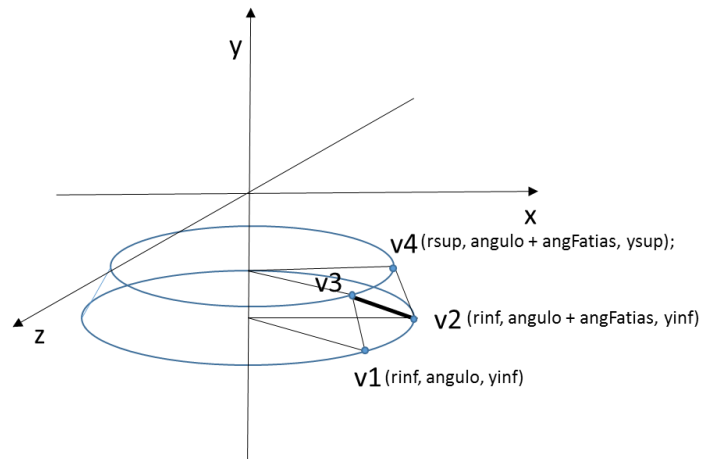
É dentro do ciclo interno, que se vão calcular estes raios e com isto gerar os vértices necessários à posterior criação dos triângulos. Após termos os vértices, geramos então os triângulos, que irão ser importantes na criação da parte lateral do cone. Para a criação da base de um cone, temos um “if” no ciclo interno, que só vai desenhar os triângulos da base quando o valor da variável y na iteração corresponder a (-altura/2), que é o valor de Y onde se inicia a forma e onde deve ficar desenhada a base.

Para gerar este modelo, foram usadas coordenadas cilíndricas, mais adequadas, onde cada ponto é representado pelas seguintes coordenadas  $(\rho, \varphi, y)$ . O  $\rho$  é a projeção do ponto sobre a base XZ, que em cada stack será o valor do raio nessa iteração, calculado com as formulas apresentadas anteriormente, o  $\varphi$  é o ângulo face ao eixo dos X, variando de 0 a  $2\pi$  e múltiplo de  $\alpha$  (ângulo de 1 fatia =  $(2\pi / \text{\#Fatias})$ ), e y a altura que vai variando conforme o número de stacks com um incremento constante. Tendo as coordenadas de cada vértice, em coordenadas cilíndricas, foi criada também uma função, na classe Vertice, que gera um vértice convertendo as suas coordenadas esféricas para cartesianas segundas as seguintes equações.

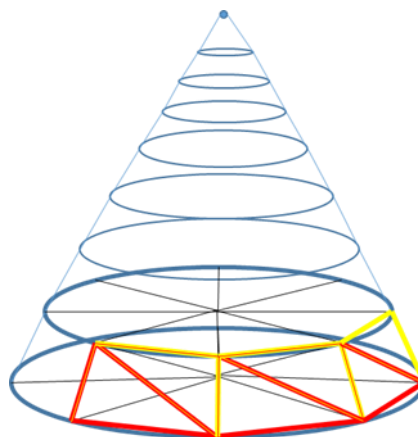
```
Vertice Vertice::cilindricas(double raio, double alpha, double y) {  
    double x = raio * sin(alpha);  
    double z = raio * cos(alpha);  
    return Vertice(x, y, z);  
}
```

Figura 13 - Função que cria um vértice em coordenadas cartesianas, recebendo como parâmetros as suas coordenadas cilíndricas, sendo **raio** o raio numa dada iteração, **alpha** o ângulo de cada fatia (iterado de 0 a  $2\pi$  e referente ao eixo dos X) e por fim, o **Y** representa a altura da stack numa dada iteração.

As seguintes figuras procuram ilustrar um pouco do processo de construção de um cone, assim como alguns pontos mais específicos e as suas coordenadas. Estes pontos e as suas coordenadas servem como um exemplo representativo de todos os pontos gerados no processo de criação do cone em si.



*Figura 14 - Criação de vértices com as seguintes coordenadas, e posterior construção dos triângulos que formam os quadrados da lateral do cone.*



*Figura 15 - Esboço do algoritmo de criação de um cone. Serão criadas 8 stacks e 8 fatias. Primeiro é iterado o anel da primeira stack, compondo os triângulos de cada fatia, e posteriormente iteram-se os seguintes níveis, repetindo o processo.*



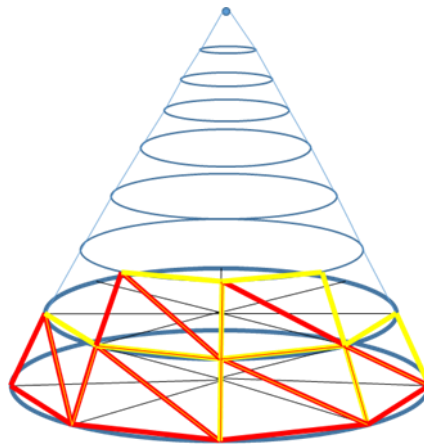


Figura 16 - Tendo já percorrido toda a stack de baixo, o algoritmo passa para a próxima iterando da mesma forma como para a stack anterior.

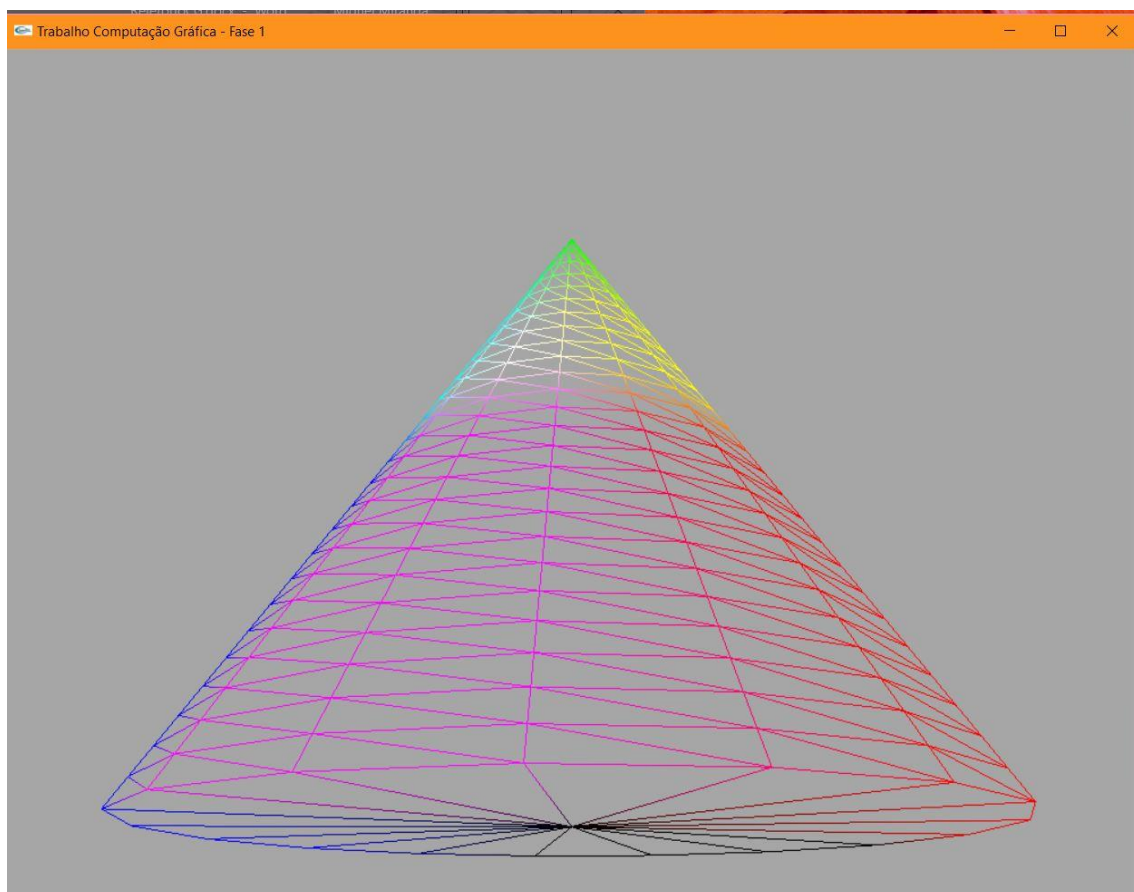


Figura 17 – Exemplo de um cone gerado, com raio da base = 5, altura = 8, fatias = 17 e 25 stacks



## 2.2 Aplicação Motor3D

A aplicação Motor3D tem como função, a partir do nome de um ficheiro xml, desenhar o modelo de uma figura previamente criada com a aplicação Gerador.

Ao abrir o ficheiro xml irá localizar todos os elementos com nomes de um ficheiro como atributo. Esta tarefa foi bastante simplificada depois de uma breve contextualização à semântica do xml e ao uso da biblioteca tinyxml importada. Tendo então lido quais os ficheiros que devem ser lidos, este módulo vai simplesmente usá-los como input para funções específicas, responsáveis pelo parsing das linhas e criação dos vértices, posteriormente renderizados nas funções glut.

### 3. Instruções de funcionamento dos Projetos

#### 3.1 Gerador

A execução da aplicação Gerador deve ser feita dando como parâmetros o nome da forma geométrica que se pretende gerar e os respetivos parâmetros necessários para compor os vértices do modelo.

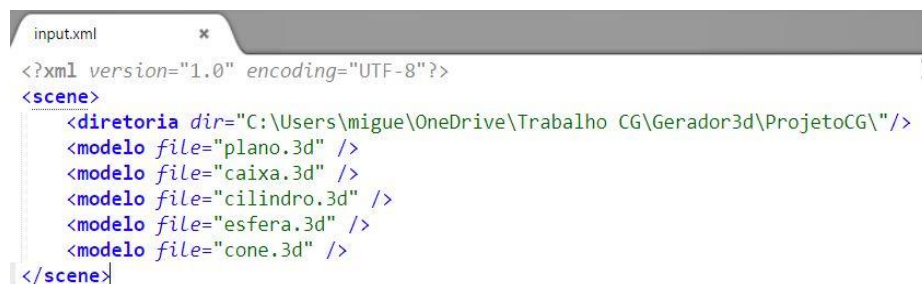
Para este aspeto do input necessário para arrancar o gerador, foi tida em conta hipótese de erros. Caso estes parâmetros não sejam corretamente inseridos ou estiverem em número insuficiente para a figura pedida, a geração da figura não se realiza e é visualizado no terminal um aviso de erro na inserção de parâmetros.

Na perspetiva de sucesso, o programa gerador é também interativo e após terminar com a geração dos vértices e respetiva escrita no ficheiro solicitado, responde com aviso de término e sucesso.

#### 3.2 Motor3D

A execução da aplicação Motor3D depende apenas da passagem de um único parâmetro, que será o nome do ficheiro xml utilizado para gerar os modelos.

Face ao formato do ficheiro xml, deve destacar-se que deve existir um campo **diretoria**, com o atributo **dir** onde deve ser especificada a diretoria onde se encontram os ficheiros que se vão seguir. Estes ficheiros, identificados pelo seu nome, devem estar contidos no elemento **modelo** e especificados no atributo **file**. A seguinte imagem exemplifica a forma do ficheiro xml de input utilizado e disponibilizado para o teste da aplicação Motor3D.



```
<?xml version="1.0" encoding="UTF-8"?>
<scene>
  <diretoria dir="C:\Users\migue\OneDrive\Trabalho CG\Gerador3d\ProjetoCG\"/>
  <modelo file="plano.3d" />
  <modelo file="caixa.3d" />
  <modelo file="cilindro.3d" />
  <modelo file="esfera.3d" />
  <modelo file="cone.3d" />
</scene>
```

Figura 18 – Exemplo dos campos de um ficheiro xml a ser usado como input da aplicação Motor3D

Na possibilidade de encontrar vários nomes de ficheiros no ficheiro xml, eles serão todos considerados e, se forem encontrados, desenhados na mesma janela sobrepondo os diversos modelos.

A nível da aplicação Motor3D foi também tido em conta o controlo de erros quando o nome dos ficheiros não existir ou não for encontrado na diretoria especifica.



Como aspeto negativo, existe o facto de para aceder ao ficheiro .xml, como é passado ao main apenas o seu nome, é necessário alterar no código o valor de uma variável global que indica a diretoria dos ficheiros .xml a utilizar. Tida esta diretoria para os .xml identificada, a aplicação Motor3D concatena o nome do ficheiro especificado com a diretoria que tem como variável global e chega assim até ele.

Como aspetos positivos, destacamos as diversas formas de interação e visualização dos modelos renderizados, através de diversas teclas. Estas funcionalidades serão descritas na seção seguinte.

### 3.2.1 Interação com os modelos desenhados

Para possibilitar uma melhor visualização das figuras geométricas realizadas foram criadas as seguintes opções, baseadas nos conhecimentos apreendidos nas aulas práticas, quer permitem deslocar e rodar a figura ou alterar o ponto de camara.

#### 1. Alteração do ponto visão da câmara:

- Tecla Up (desloca para cima)                      -Tecla Down (desloca para baixo)
- Tecla Right (desloca para a direita)           -Tecla Left (desloca para a esquerda)

#### 2. Movimentação do modelo:

- Tecla W - Mover no sentido positivo do eixo z.
- Tecla S - Mover no sentido negativo do eixo z.
- Tecla A - Mover no sentido positivo do eixo x.
- Tecla D - Mover no sentido negativo do eixo x.

#### 3. Rotação do modelo:

- Tecla Y - Rodar no sentido anti-horário eixo x.
- Tecla H - Rodar no sentido horário eixo x.
- Tecla U - Rodar no sentido anti-horário eixo y.
- Tecla J - Rodar no sentido horário eixo y.
- Tecla I - Rodar no sentido anti-horário eixo z.
- Tecla K - Rodar no sentido horário eixo z.

#### 4. Aumentar e diminuir o zoom da camara:

- Tecla + - Aumenta zoom da câmara.
- Tecla - - Diminui zoom da câmara.

#### 5. Opções de visualização do modelo:

- Mostrar figura só com linhas (Wire) - tecla "1".
- Mostrar a figura toda preenchida (Fill) - tecla "2".
- Mostrar figura só com pontos dos vértices (Point) - tecla "3".
- Tecla O - Restaura todas as definições de visualização iniciais.



## 4. Conclusão

Durante a realização deste trabalho fomos encontrando algumas dificuldades, mais concretamente na geração dos vértices da esfera e do cone. A transição para um sistema de coordenadas esféricas e cilíndricas trouxe ainda alguns problemas, que, a nosso ver, foram totalmente ultrapassados e compreendidos, simplificando a criação dos algoritmos destas figuras geométricas.

Deste modo cumprimos todas as exigências pretendidas para esta primeira fase do trabalho, construindo todas as figuras geométricas pedidas, com a possibilidade extra de gerar ainda um cilindro. Ainda relativos aos requisitos mínimos desta fase, o desenvolvimento do motor foi realizado com sucesso, dando consistência aos conhecimentos sobre as funções do glut e de parsing de ficheiros .xml.

No final desta implementação, consideramos que, no limite, até nem seria eventualmente necessária esta organização de classes em que um modelo é um conjunto de triângulos, um triângulo um conjunto de três vértices, e um vértice um conjunto de 3 coordenadas. Contudo, ponderando as fases seguintes do projeto e por uma organização mental e conceptual, optamos por organizar o código desse modo.



## 5. Referências

Lighthouse3d. (2017). GLUT Tutorial. [online] Available at:  
<http://www.lighthouse3d.com/tutorials/glut-tutorial/> [Accessed 27 Feb. 2017].

## 6. Anexos

### 6.1 Código fonte para gerar vértices de um cone

```
void Modelo::cone(double raio, double altura, int fatias, int stacks,
char* nameFile3d) {
    double angFatias = 2 * M_PI / fatias;
    double paramY = altura / stacks;

    std::vector<Triangulo> tris;
    double iterY, yinf, ysup, angulo;
    yinf = - altura / 2;

    double rinf, rsup, r = 0;
    for (iterY = 0; iterY < stacks; iterY++) {
        // iterar o eixo dos YY para as stacks
        ysup = yinf + paramY;
        for (angulo = 0; angulo < 2 * M_PI; angulo += angFatias) {
            // iteras as várias fatias
            rinf = raio - raio*iterY / stacks;
            rsup = raio - (raio*(iterY+1)) / stacks;
            Vertice v1 = Vertice::cilindricas(rinf, angulo, yinf);
            Vertice v2 = Vertice::cilindricas(rinf, angulo +
angFatias, yinf);
            Vertice v3 = Vertice::cilindricas(rsup, angulo, ysup);
            Vertice v4 = Vertice::cilindricas(rsup, angulo +
angFatias, ysup);

            tris.insert(tris.end(), { Triangulo(v2,v4,v3),
Triangulo(v2,v3,v1) });
            if (yinf == -altura / 2) {
                //Condição para criar os triangulos da base do
cone
                Vertice o = Vertice(0, -altura / 2, 0); //
Vertice do centro do circulo da base do cone
                tris.insert(tris.end(), { Triangulo(v1,o,v2)});
            }
            yinf = ysup;
        }
    }

    Modelo cone = Modelo(tris);
    escreveModelo(cone, nameFile3d);
    cout << "O modelo do cone foi criado com sucesso no ficheiro "
        << nameFile3d << "." << endl;
}
```

## 6.2 Código fonte para gerar vértices de uma esfera

```
void Modelo::sphere(double raio, int fatias, int stacks, char* nameFile3d)
{
    double angStacks = M_PI / stacks;
    double angSlices = 2 * M_PI / fatias;
    std::vector<Triangulo> tris;
    double alpha, beta;
    for (beta = M_PI; beta > -M_PI; beta -= angStacks) {
        //Iterar o eixo dos Y, de cima para baixo, conforme o numero
de stacks
        for (alpha = 0; alpha < 2 * M_PI; alpha+=angSlices) {
            //Iterar cada nivel stack, rodando de 0 a 2*pi para
criar os triangulos das fatias
            Vertice v1 = Vertice::esfericas(raio, alpha, beta);
            Vertice v2 = Vertice::esfericas(raio, alpha +
angSlices, beta);
            Vertice v3 = Vertice::esfericas(raio, alpha +
angSlices, beta + angStacks);
            Vertice v4 = Vertice::esfericas(raio, alpha, beta +
angStacks);
            tris.insert(tris.end(), { Triangulo(v2,v3,v4),
Triangulo(v2,v4,v1)});
        }

        Modelo esfera = Modelo(tris);
        escreveModelo(esfera, nameFile3d);
        cout << "O modelo da esfera foi criado com sucesso no ficheiro "
<< nameFile3d << "." << endl;
    }
}
```

## 6.3 Código fonte para gerar vértices de um cilindro

```
void Modelo::cilindro(double raio, double altura, int fatias, int stacks,
char* nameFile3d) {
    double angFatias = 2 * M_PI / fatias;
    double paramY = altura / stacks;
    double yinf, ysup, ang = 0, x1,x2,z1,z2;
    int iterY;
    std::vector<Triangulo> tris;

    double y = altura / 2;

    yinf = -altura / 2;
    for (iterY = 0; iterY < stacks; iterY++) {
        //iterar o eixo dos Y, conforme o numero de stacks
        ysup = yinf + paramY;
        for (ang = 0; ang < 2 * M_PI; ang += angFatias) {
            //rodar desde 0 a 2*pi cada nivel stack para compor os
triangulos
            x1 = raio*cos(ang);
            x2 = raio*cos(ang + angFatias);
            z1 = raio*sin(ang);
            z2 = raio* sin(ang + angFatias);
            //Considerando cada stack como um anel, v1 e v2 vão
estar na circunferencia inferior
            Vertice v1 = Vertice(x1, yinf, z1);
            Vertice v2 = Vertice(x2, yinf, z2);
```



```
//Considerando cada stack como um anel, v3 e v4 vão
estar na circunferencia superior
Vertice v3 = Vertice(x1, ysup, z1);
Vertice v4 = Vertice(x2, ysup, z2);
tris.insert(tris.end(), { Triangulo(v1,v4, v2),
Triangulo(v1,v3,v4)});
if (yinf == -y) {
    //Condição para gerar a face inferior do cilindro
    Vertice b1 = Vertice(0, yinf, 0); //Vertice do
centro do circulo da base do cilindro
    Vertice b2 = Vertice(x1, yinf, z1);
    Vertice b3 = Vertice(x2, yinf, z2);
    tris.insert(tris.end(), { Triangulo(b3,b1,b2) });

// Face inferior
}
if (iterY == stacks -1 ) {
    //Condição para gerar a face superior do cilindro
    Vertice v1 = Vertice(0, ysup, 0); //Vertice do
centro do circulo do teto do cilindro
    Vertice v2 = Vertice(x1, ysup, z1);
    Vertice v3 = Vertice(x2, ysup, z2);
    tris.insert(tris.end(), {Triangulo(v2,v1,v3)
}); // Face superior
}
    yinf = ysup;
}

Modelo cilindro = Modelo(tris);
escreveModelo(cilindro, nameFile3d);
cout << "O modelo do cilindro foi criado com sucesso no ficheiro "
    << nameFile3d << "." << endl;
}
```

## 6.4 Código fonte para gerar vértices de uma caixa

```
void Modelo::box(double largura, double compri, double altura, double
nDiv, char* nameFile3d) {

    double x = largura / 2, y = altura / 2, z = compri / 2;
    double paramY = (altura / nDiv);
    double paramX = largura / nDiv;
    double paramZ = compri / nDiv;

    std::vector<Triangulo> tris;
    int iterX, iterY, iterZ;
    double xinf, xsup, yinf, ysup, zinf, zsup;

    /*ciclo no eixo do y para criar as stacks*/
    yinf = -y;
    for (iterY = 0; iterY != nDiv; iterY++) {
        ysup = yinf + paramY;
        /*ciclo no eixo do x para criar as faces frontal e traseira*/
        xinf = -x;
        for (iterX = 0; iterX != nDiv; iterX++) {
            xsup = xinf + paramX;
            /*Vertices da base inferior*/
            Vertice b1 = Vertice(xinf, yinf, z);
            Vertice b2 = Vertice(xsup, yinf, z);
            Vertice b3 = Vertice(xsup, yinf, -z);
            Vertice b4 = Vertice(xinf, yinf, -z);
            /*Vertice da base superior*/
            Vertice s1 = Vertice(xinf, ysup, z);
```



```
Vertice s2 = Vertice(xsup, ysup, z);
Vertice s3 = Vertice(xsup, ysup, -z);
Vertice s4 = Vertice(xinf, ysup, -z);

tris.insert(tris.end(),
{ Triangulo(b2,s1,b1), Triangulo(b2,s2,s1), // Face
  Triangulo(b4,s4,s3), Triangulo(b4,s3,b3) }); //face

frontal
traseira

xinf = xsup;
}

/*ciclo no eixo do Z para criar as faces esquerda e direita*/
zinf = -z;
for (iterZ = 0; iterZ != nDiv; iterZ++) {
  zsup = zinf + paramZ;
  /*Vertices da base inferior*/
  Vertice b1 = Vertice(-x, yinf, zsup);
  Vertice b2 = Vertice(x, yinf, zsup);
  Vertice b3 = Vertice(x, yinf, zinf);
  Vertice b4 = Vertice(-x, yinf, zinf);
  /*Vertice da base superior*/
  Vertice s1 = Vertice(-x, ysup, zsup);
  Vertice s2 = Vertice(x, ysup, zsup);
  Vertice s3 = Vertice(x, ysup, zinf);
  Vertice s4 = Vertice(-x, ysup, zinf);

  tris.insert(tris.end(),
{ Triangulo(b1,s1,s4), Triangulo(b1,s4,b4), // Face
  Triangulo(b3,s3,s2), Triangulo(s2,b2,b3) }); //face

  esquerda
  direita

  zinf = zsup;
}

yinf = ysup; // subir um camada na stack
}

/*Ciclo nos eixos X e Z para criar as faces superior e inferior do
cubo*/
xinf = -x;
for (iterX = 0; iterX != nDiv; iterX++) {
  xsup = xinf + paramX;
  zinf = -z;
  for (iterZ = 0; iterZ != nDiv; iterZ++) {
    zsup = zinf + paramZ;
    /*Vertices da base inferior*/
    Vertice b1 = Vertice(xinf, -y, zsup);
    Vertice b2 = Vertice(xsup, -y, zsup);
    Vertice b3 = Vertice(xsup, -y, zinf);
    Vertice b4 = Vertice(xinf, -y, zinf);
    /*Vertice da base superior*/
    Vertice s1 = Vertice(xinf, y, zsup);
    Vertice s2 = Vertice(xsup, y, zsup);
    Vertice s3 = Vertice(xsup, y, zinf);
    Vertice s4 = Vertice(xinf, y, zinf);

    tris.insert(tris.end(),
{ Triangulo(s2,s3,s4), Triangulo(s2,s4,s1), // Face
  Triangulo(b1,b4,b3), Triangulo(b1,b3,b2) });

    superior

    //face inferior

    zinf = zsup;
  }
}
```



```
        xinf = xsup;
    }

    Modelo caixa = Modelo(tris);
    escreveModelo(caixa, nameFile3d);
    cout << "O modelo da caixa foi criado com sucesso no ficheiro "
          << nameFile3d << "." << endl;
}
```

## 6.5 Código fonte para gerar vértices de um plano

```
void Modelo::plane(double ladoSqr, char* nameFile3d){

    double x = ladoSqr / 2;

    Vertice v1 = Vertice(x, 0, x);
    Vertice v2 = Vertice(x, 0, -x);
    Vertice v3 = Vertice(-x, 0, -x);
    Vertice v4 = Vertice(-x, 0, x);

    vector<Triangulo> tris = { Triangulo(v1, v2, v3), Triangulo(v1, v3,
v4) };

    Modelo plano = Modelo( tris );

    escreveModelo(plano, nameFile3d);

    cout << "O modelo do plano foi criado com sucesso no ficheiro "
          << nameFile3d << "." << endl;
}
```