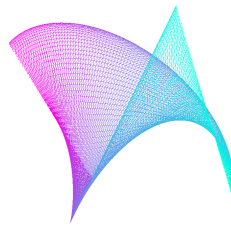


Universidade do Minho  
Departamento de Informática

**Solaris**



Computação Gráfica  
Grupo 24 - Fase 1

13 de março, 2022



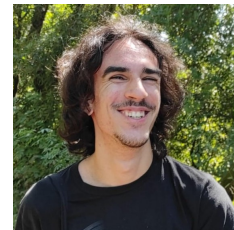
Beatriz  
Rodrigues  
(*a93230*)



Francisco Neves  
(*a93202*)



Guilherme  
Fernandes  
(*a93216*)



João Carvalho  
(*a93166*)

# Índice

<b>1</b>	<b>Introdução</b>	<b>3</b>
<b>2</b>	<b>Utilities</b>	<b>4</b>
2.1	Bibliotecas . . . . .	4
2.2	<i>Script</i> . . . . .	4
2.3	Classes Comuns . . . . .	4
<b>3</b>	<b>Generator</b>	<b>5</b>
3.1	Plano . . . . .	5
3.2	Caixa . . . . .	7
3.3	Cone . . . . .	11
3.4	Esfera . . . . .	14
3.5	Cilindro . . . . .	16
3.6	Torus . . . . .	18
<b>4</b>	<b>Engine</b>	<b>22</b>
4.1	Carregamento da <i>scene</i> . . . . .	22
4.1.1	Carregamento das primitivas . . . . .	23
4.1.2	Leitura do XML . . . . .	23
4.2	Câmara . . . . .	24
4.2.1	Orbital . . . . .	24
4.2.2	FPV . . . . .	26
<b>5</b>	<b>Conclusão e Trabalho Futuro</b>	<b>28</b>

# 1. Introdução

*Solaris* é um programa constituído por duas partes essenciais: um *generator* e um *engine*. O primeiro é responsável por gerar ficheiros com um conjunto de pontos que representam uma determinada primitiva. Posteriormente, estes pontos serão interpretados pelo *engine*, de forma a fazer a representação 3D correspondente.

As primitivas que foram desenvolvidas foram as seguintes:

- Plano
- Caixa
- Cone
- Esfera
- Cilindro
- Torus

De forma a tornar mais interessante a visualização das primitivas, foi ainda implementada a noção de movimento através da câmara.

## 2. Utilities

### 2.1 Bibliotecas

Para o seu funcionamento, é necessário recorrer às bibliotecas `{fmt}`, utilizada para facilitar a formatação de strings, `result` para facilitar o tratamento de erros e `tinyxml2` para fazer o *parsing* do ficheiro XML.

### 2.2 *Script*

Para que o processo de recompilar o programa se tornasse bastante mais simples, os executáveis correspondentes às duas componentes principais do programa podem ser gerados através de um *script* designado por *build.sh*.

### 2.3 Classes Comuns

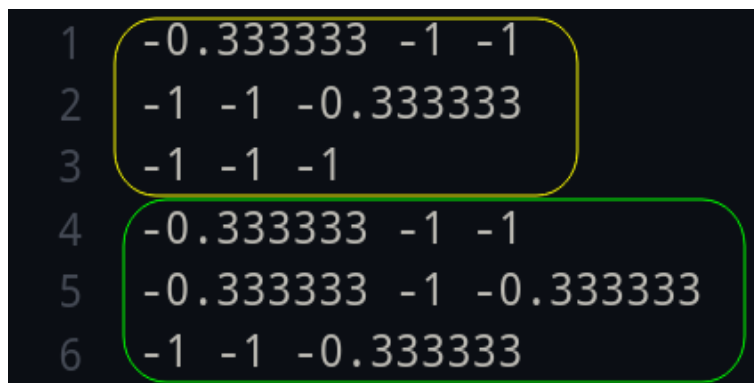
Para além disso, com o intuito de tornar mais clara a representação das primitivas, foi implementada uma classe *Point.cpp*, que instancia coordenadas cartesianas (constituídas por 3 valores - x, y e z) e esféricas (constituídas pelo raio, inclinação e azimuth). Esta classe é também capaz de efetuar conversões entre estes tipos de coordenadas.

De modo a facilitar o *debugging* e para disponibilizar informação ao utilizador acerca de um resultado não habitual do programa, existe ainda uma classe *Logger.cpp*, capaz de disponibilizar erros e *warnings*.

### 3. Generator

Como referido anteriormente, o *generator* origina um conjunto de pontos. Todas as primitivas desenvolvidas são desenhadas apenas a partir de triângulos, ou seja, conjuntos de três pontos. Desta forma, são escritos para o ficheiro output com o intuito de serem lidos pelo *engine* de forma agrupada.

O formato escolhido para os ficheiros *.3d* corresponde à indicação de um ponto por linha. Desta forma, grupos de três linhas correspondem a um triângulo a desenhar.



```
1 -0.333333 -1 -1
2 -1 -1 -0.333333
3 -1 -1 -1
4 -0.333333 -1 -1
5 -0.333333 -1 -0.333333
6 -1 -1 -0.333333
```

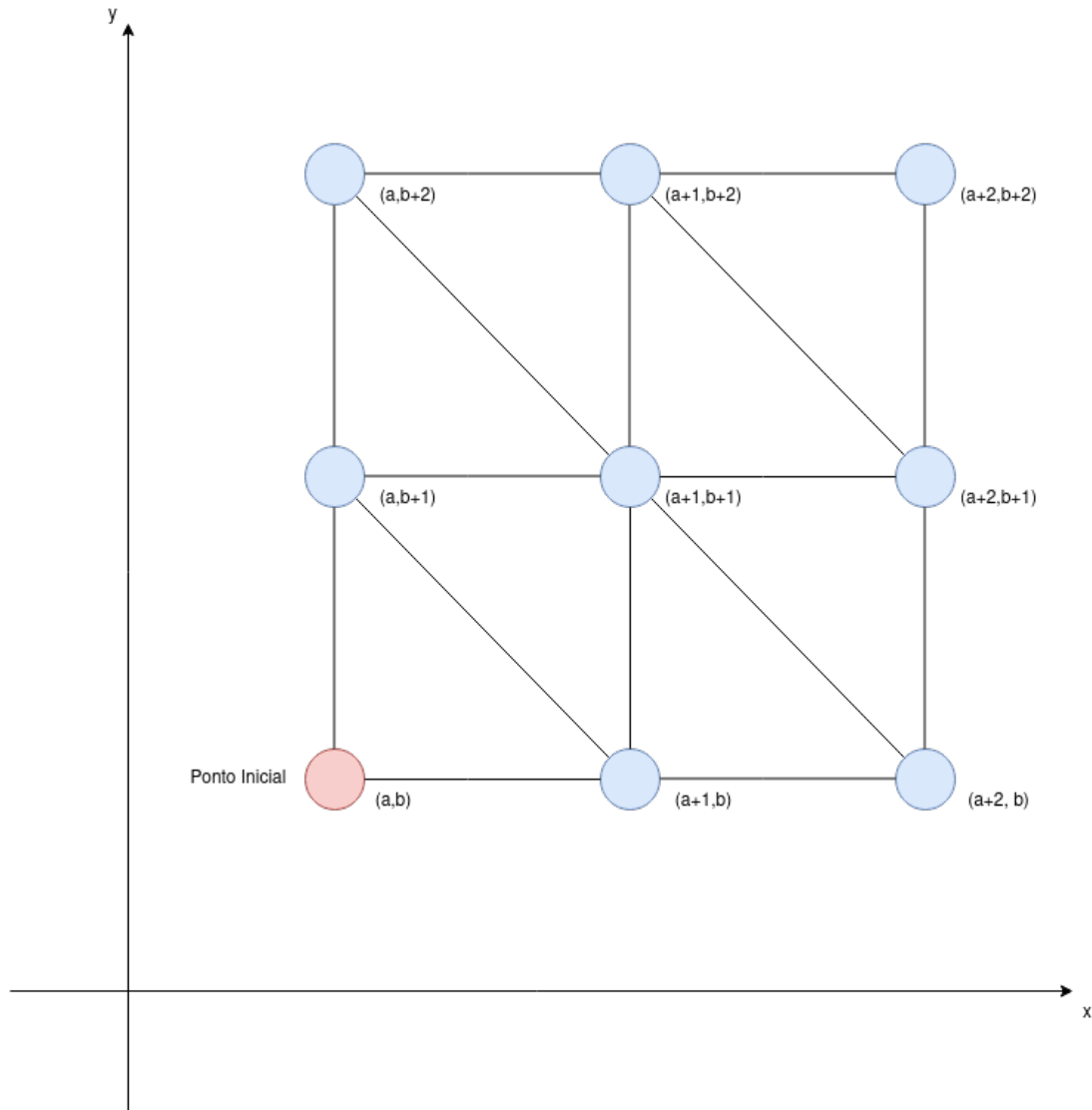
**Figura 1.** Ficheiro *.3d*

Seguidamente, explica-se a lógica por detrás dos algoritmos criados.

#### 3.1 Plano

Um plano é constituído a partir de um dado comprimento do lado e de um número de divisões, que devem ser efetuadas nas direção do eixo  $x$  e do eixo  $y$ .

Para implementar o pretendido, foi desenvolvido um algoritmo que gera 2 triângulos de cada vez. Este foi baseado na lógica seguinte, que demonstra que é possível obter o valor de cada ponto necessário para a formação do plano a partir de um ponto inicial.



**Figura 2.** Lógica do Plano

Em termos práticos, o algoritmo foi concretizado da seguinte forma:

```
for (size_t i = 0; i < n_divisions; ++i) {
    for (size_t j = 0; j < n_divisions; ++j) {
        Point p1 =
            starting_point + Point::cartesian(j * step, 0, i * step);
        Point p2 =
            starting_point + Point::cartesian(j * step, 0, (i+1) * step);
        Point p3 =
            starting_point + Point::cartesian((j+1) * step, 0, i * step);
        Point p4 =
            starting_point + Point::cartesian((j+1) * step, 0, (i+1) * step);

        coords.push_back(p1);
        coords.push_back(p2);
        coords.push_back(p3);

        coords.push_back(p2);
    }
}
```

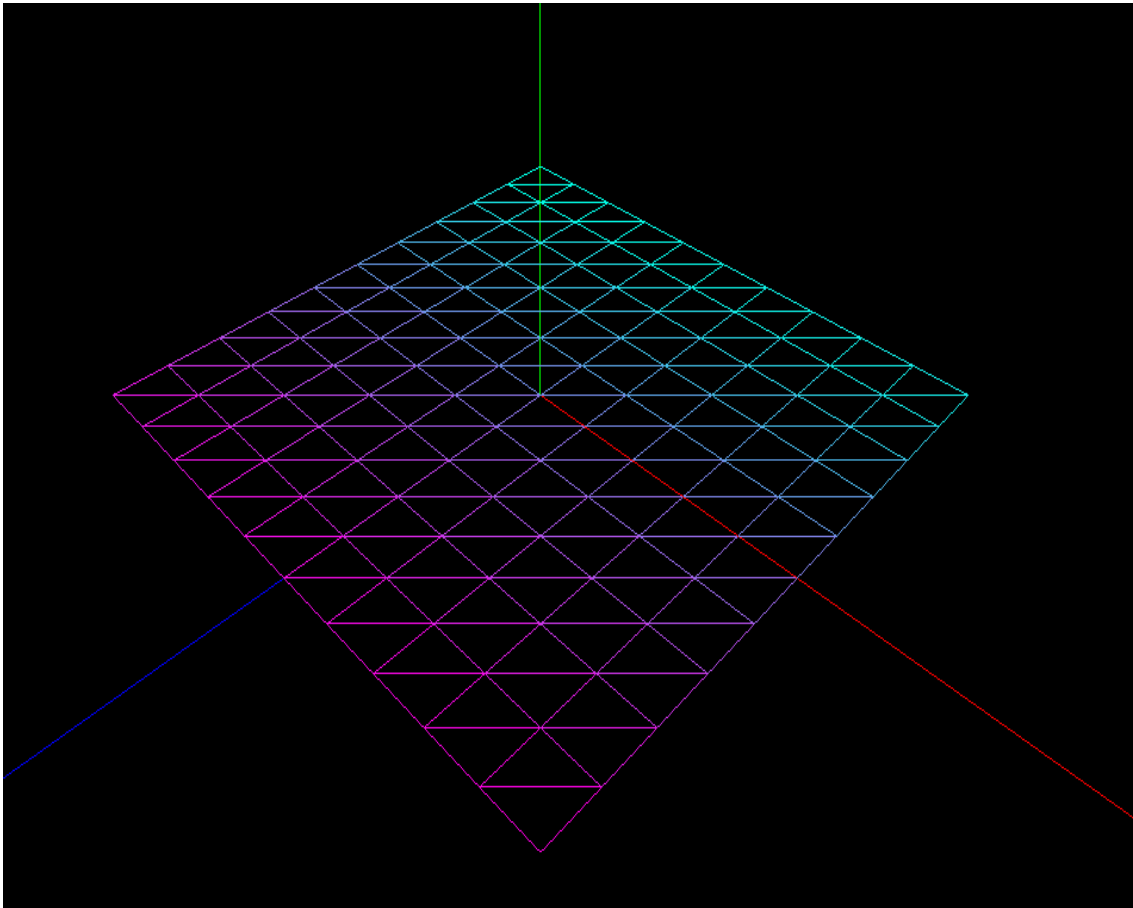
```

        coords.push_back(p4);
        coords.push_back(p3);
    }
}

```

O código inserido anteriormente descobre os pontos necessários criando múltiplos quadrados ao longo do eixo  $x$  (correspondentes ao número de divisões pretendidas) e repetindo este processo ao longo do eixo  $z$ .

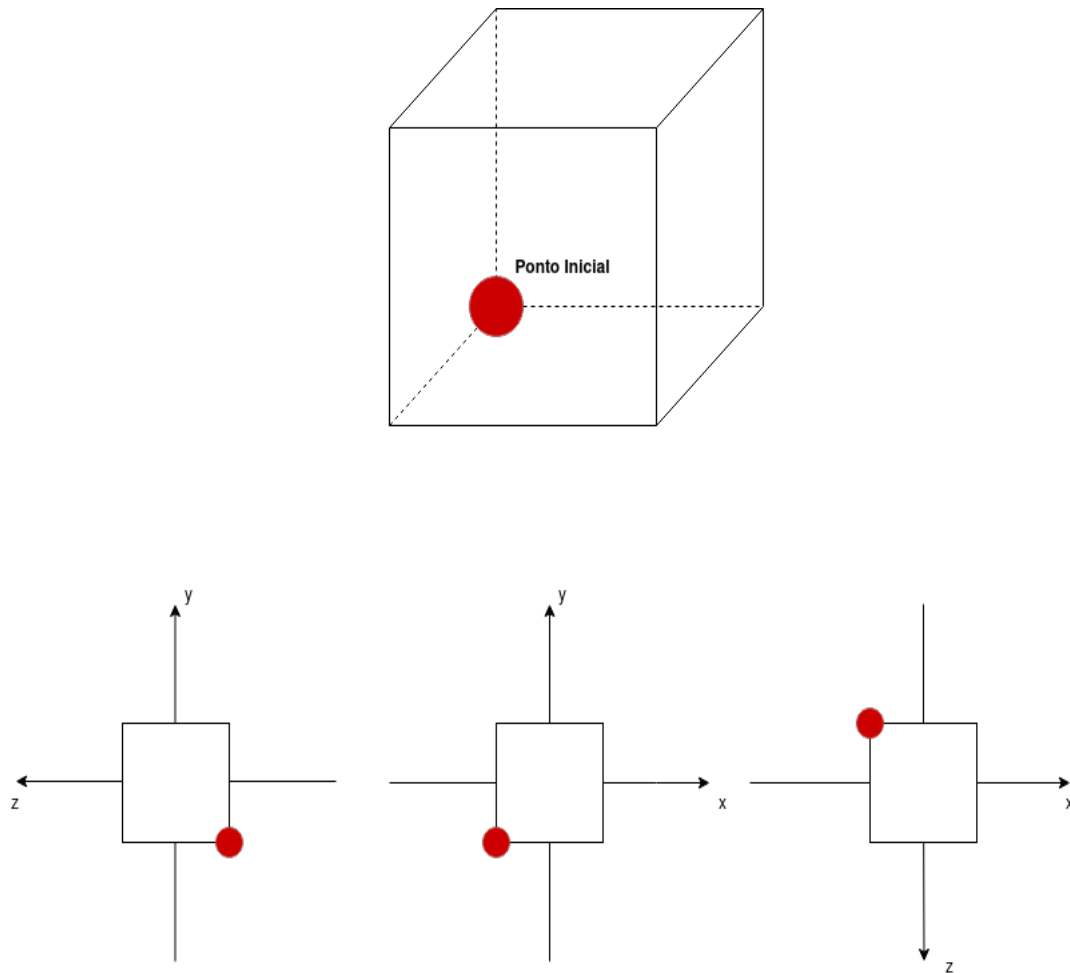
O resultado da implementação reflete-se na seguinte imagem.



**Figura 3.** Plano: Lado - 10 ; Número de divisões - 10

## 3.2 Caixa

A caixa é produzida a partir de um determinado comprimento do lado e do número de divisões das faces. De imediato, estabeleceu-se que o algoritmo utilizado para o plano possivelmente poderia ser adaptado de forma a concretizar esta primitiva. De forma a simplificar o algoritmo o máximo possível, selecionou-se um único ponto a partir do qual toda a caixa poderia surgir. Este ponto estaria ligado a 3 das faces e, assim, dar-lhes-ia origem através do algoritmo referido.



**Figura 4.** Lógica da Caixa

Para obter as restantes 3 faces, aproveitamos o facto de cada uma destas ser bastante semelhante por ter uma das faces conhecidas paralela a ela própria e, por isso, os seus pontos só diferem entre eles por uma única componente do ponto. Por exemplo, a base e o topo da caixa apenas diferem entre si pelo valor de  $z$ .

O referido anteriormente resulta no seguinte algoritmo:

```
for (size_t j = 0; j < n_divisions; ++j) {
    for (size_t k = 0; k < n_divisions; ++k) {

        // Base triangles
        Point p1 =
            starting_point + Point::cartesian(k*plane_step,0,j*plane_step);
        Point p2 =
            starting_point + Point::cartesian(k*plane_step,0,(j+1)*plane_step);
        Point p3 =
            starting_point + Point::cartesian((k+1)*plane_step,0,j*plane_step);
        Point p4 =
            starting_point + Point::cartesian((k+1)*plane_step,0,(j+1)*plane_step);
```



```

coords.push_back(p3);
coords.push_back(p2);
coords.push_back(p1);

coords.push_back(p3);
coords.push_back(p4);
coords.push_back(p2);

// Top triangles
p1.set_y(length/2); p2.set_y(length/2);
p3.set_y(length/2); p4.set_y(length/2);

coords.push_back(p1);
coords.push_back(p2);
coords.push_back(p3);

coords.push_back(p2);
coords.push_back(p4);
coords.push_back(p3);

// Back Right triangles
p1 =
    starting_point + Point::cartesian(k*plane_step, j*plane_step, 0);
p2 =
    starting_point + Point::cartesian(k*plane_step, (j+1)* plane_step, 0);
p3 =
    starting_point + Point::cartesian((k+1)*plane_step, j*plane_step, 0);
p4 =
    starting_point + Point::cartesian((k+1)*plane_step, (j+1)*plane_step, 0);

coords.push_back(p1);
coords.push_back(p2);
coords.push_back(p3);

coords.push_back(p2);
coords.push_back(p4);
coords.push_back(p3);

// Front Right triangles
p1.set_z(length/2); p2.set_z(length/2);
p3.set_z(length/2); p4.set_z(length/2);

coords.push_back(p3);
coords.push_back(p2);
coords.push_back(p1);

coords.push_back(p3);
coords.push_back(p4);
coords.push_back(p2);

```

```

    // Left Back triangles
    p1 =
        starting_point + Point::cartesian(0,k*plane_step,j*plane_step);
    p2 =
        starting_point + Point::cartesian(0,(k+1)*plane_step, j*plane_step);
    p3 =
        starting_point + Point::cartesian(0,k*plane_step,(j+1)*plane_step);
    p4 =
        starting_point + Point::cartesian(0,(k+1)*plane_step,(j+1)*plane_step);

    coords.push_back(p3);
    coords.push_back(p2);
    coords.push_back(p1);

    coords.push_back(p3);
    coords.push_back(p4);
    coords.push_back(p2);

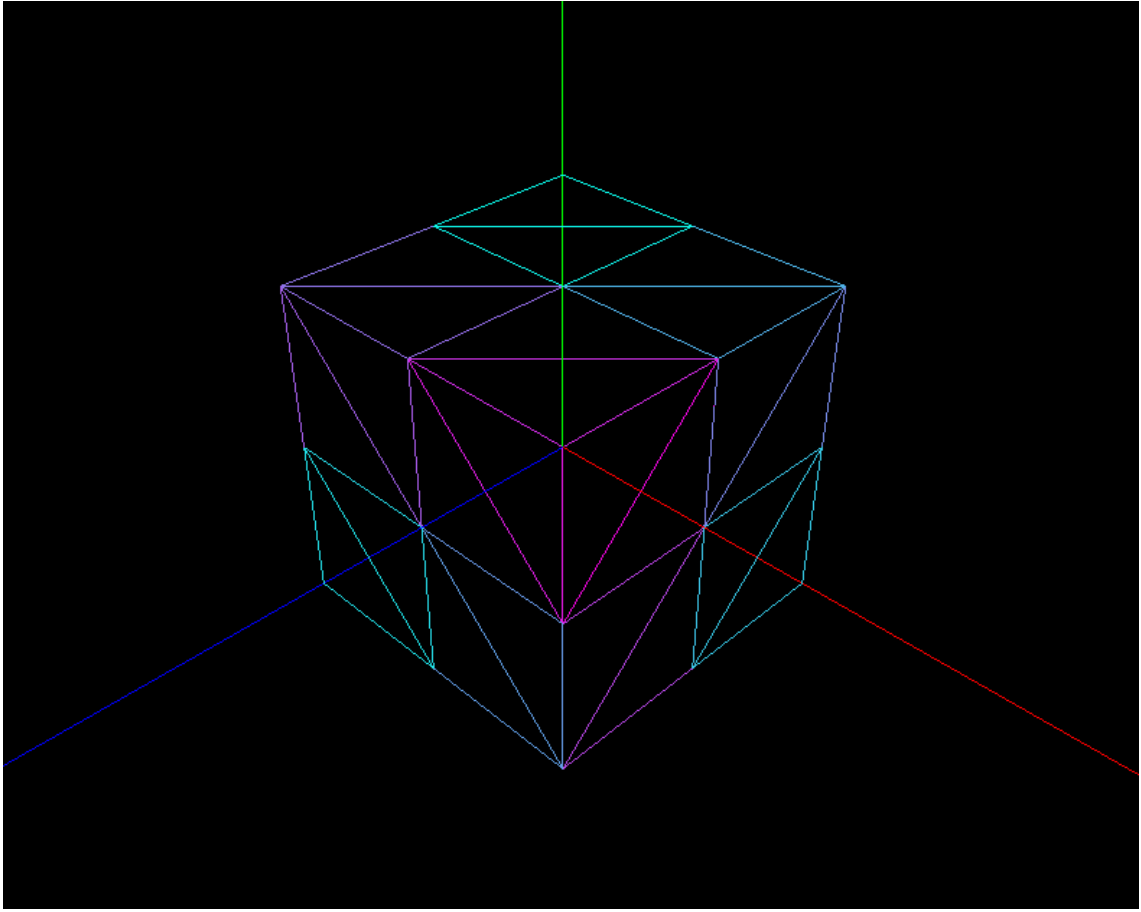
    // Left Front triangles
    p1.set_x(length/2); p2.set_x(length/2);
    p3.set_x(length/2); p4.set_x(length/2);

    coords.push_back(p1);
    coords.push_back(p2);
    coords.push_back(p3);

    coords.push_back(p2);
    coords.push_back(p4);
    coords.push_back(p3);
}
}

```

O resultado do algoritmo elaborado é o seguinte:



**Figura 5.** Caixa: Lado - 5 ; Número de Divisões - 2

### 3.3 Cone

De forma a ser produzido um cone, é necessário saber a sua altura, o raio da sua base, o número de *stacks* e o número de *slices*. Tendo por base um bolo de casamento, podemos considerar que as *slices* representarão a fatia do bolo e as *stacks* o nível do bolo.

Como é sabido, um cone poderá ser visto como um conjunto de círculos em que o raio vai diminuindo até chegar a um ponto único que representa o topo do cone. Desta forma, e sabendo que deveríamos ser capazes de dividir a sua estrutura em *stacks* e *slices*, conseguimos deduzir as seguintes equações:

$$\text{altura de cada stack} = \frac{\text{altura do cone}}{\text{número de stacks}} \quad (3.1)$$

$$\alpha = \frac{2\pi}{\text{número de slices}} \quad (3.2)$$

Além disso, tendo em conta que pretendemos diminuir o raio conforme aumentamos nas *stacks* do cone, temos ainda:

$$\text{passo do raio} = -\frac{\text{raio do cone}}{\text{número de stacks}} \quad (3.3)$$

Por fim, tendo em conta que construindo círculo a círculo podemos, de certa forma, construir os círculos como se estes estivessem num plano 2D com a altura fixa, no entanto, para isso, teremos de saber a altura a que ele se encontra de forma a sabermos o parâmetro  $y$  das suas coordenadas:

$$\text{altura da stack} = \text{altura de cada stack} \times \text{stack atual} \quad (3.4)$$

Como temos o parâmetro  $y$  assim definido, podemos utilizar coordenadas polares de forma a obter os valores de  $x$  e  $z$ :

$$x = \text{raio da stack} \times \sin(\alpha \text{ da slice}) \quad (3.5)$$

$$z = \text{raio da stack} \times \cos(\alpha \text{ da slice}) \quad (3.6)$$

Tendo isto definido e sabendo que a construção dos sólidos é feita através de triângulos, é necessário que, para cada *slice* em cada *stack*, se trabalhe com quatro pontos em que dois serão da *stack* atual (em que cada um será de uma *slice*) e os outros dois serão da próxima *stack* (em que cada um será de uma *slice*).

É ainda importante realçar que as condições *if* do algoritmo surgem de forma a evitar a construção de faces repetidas.

Assim sendo, obtém-se o seguinte algoritmo:

```
float stack_height = height / n_stacks;
float alpha = (2 * M_PI) / n_slices;
float radius_step = -radius / n_stacks;

float curr_radius = radius;
float curr_height = 0;
for (size_t stack = 1; stack <= n_stacks; ++stack) {

    float next_radius = radius + (radius_step * stack);
    float next_height = stack_height * stack;

    float curr_alpha = 0;
    for (size_t slice = 1; slice <= n_slices; ++slice) {
        float next_alpha = alpha * slice;
        // P1 --- P2
        // |       |
        // P3 --- P4
        Point p1 = Point::cartesian(
            next_radius*sin(curr_alpha), next_height, next_radius*cos(curr_alpha)
        );
        Point p2 = Point::cartesian(
            next_radius*sin(next_alpha), next_height, next_radius*cos(next_alpha)
        );
        Point p3 = Point::cartesian(
            curr_radius*sin(curr_alpha), curr_height, curr_radius*cos(curr_alpha)
        );
        Point p4 = Point::cartesian(
```

```

        curr_radius*sin(next_alpha),curr_height,curr_radius*cos(next_alpha)
    );

    if (stack != n_stacks) {
        coords.push_back(p4);
        coords.push_back(p2);
        coords.push_back(p1);
    }

    coords.push_back(p4);
    coords.push_back(p1);
    coords.push_back(p3);

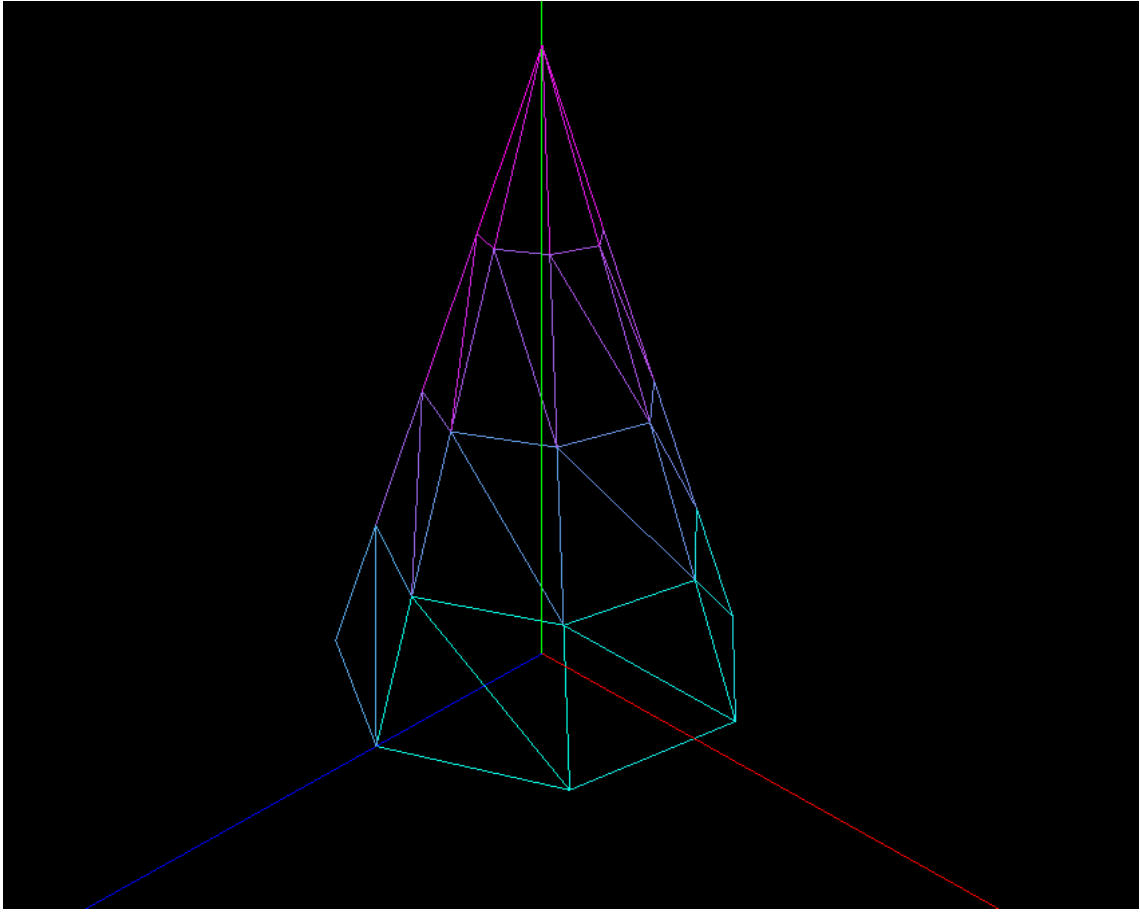
    if (stack == 1) {
        // draw base
        Point o = Point::cartesian(0, 0, 0);
        coords.push_back(p4);
        coords.push_back(p3);
        coords.push_back(o);
    }

    curr_alpha = next_alpha;
}

curr_radius = next_radius;
curr_height = next_height;
}

```

Este algoritmo resulta na seguinte representação:



**Figura 6.** Cone: Raio - 3 ; Altura - 8; Slices - 7; Stacks - 4

### 3.4 Esfera

De forma a construir a esfera, é necessário receber o seu raio, o número de *slices* e o número de *stacks*.

Através do conhecimento básico de trigonometria, foi deduzido que se poderia obter a altura de cada *stack* através da seguinte fórmula:

$$altura da stack = \frac{\pi}{número de stacks} \quad (3.7)$$

Além disso, tendo em conta que será necessária a utilização de coordenadas esféricas, podemos ainda deduzir o  $\alpha$  e o  $\beta$  necessários para a construção da esfera. De notar, que o valor de  $\alpha$  será alterado consoante a *slice* e o valor de  $\beta$  será alterado consoante a *stack*.

Temos então as seguintes equações:

$$\alpha da slice = \frac{2\pi}{número de slices} \times slice atual \quad (3.8)$$

$$\beta da stack = \frac{\pi}{2} - altura da stack \times stack atual \quad (3.9)$$

De forma semelhante à construção do cone, a construção é feita de forma iterativa com base em quatro pontos em que dois serão da *stack* atual (em que cada um será de uma slice) e os outros dois serão da próxima *stack* (em que cada um será de uma slice).

Tendo tudo isto em conta, utilizando coordenadas esféricas, poderemos criar os pontos da seguinte forma:

$$P_1 = (\text{raio da esfera}, \alpha, \beta) \quad (3.10)$$

No momento da sua criação, eles são convertidos para pontos cartesianos, visto que estes são o tipo de pontos que o *OpenGL* aceita, da seguinte forma:

$$P_{1x} = (\text{raio da esfera} \times \cos(\beta) \times \sin(\alpha)) \quad (3.11)$$

$$P_{1y} = (\text{raio da esfera} \times \sin(\beta)) \quad (3.12)$$

$$P_{1z} = (\text{raio da esfera} \times \cos(\beta) \times \cos(\alpha)) \quad (3.13)$$

Por fim, tal como no cone, as condições *if* do algoritmo surgem de forma a evitar a construção de faces repetidas.

Assim sendo, obtém-se o seguinte algoritmo:

```
float stacks_height = M_PI / n_stacks;
float alpha = (2 * M_PI) / n_slices;

float current_stack_beta = M_PI / 2;
for (size_t stack = 1; stack <= n_stacks; ++stack) {
    float next_stack_beta = M_PI / 2 - stacks_height * stack;

    float current_slice_alpha = 0;
    for (size_t slice = 1; slice <= n_slices; ++slice) {
        float next_slice_alpha = alpha * slice;
        // P3 --- P4
        // |      |
        // P1 --- P2
        Point p1 = Point::spherical(radius, current_slice_alpha, current_stack_beta);
        Point p2 = Point::spherical(radius, next_slice_alpha, current_stack_beta);
        Point p3 = Point::spherical(radius, current_slice_alpha, next_stack_beta);
        Point p4 = Point::spherical(radius, next_slice_alpha, next_stack_beta);

        if (stack != n_stacks) {
            coords.push_back(p1);
            coords.push_back(p3);
            coords.push_back(p4);
        }

        if (stack != 1) {
            coords.push_back(p2);
            coords.push_back(p1);
        }
    }
}
```

```

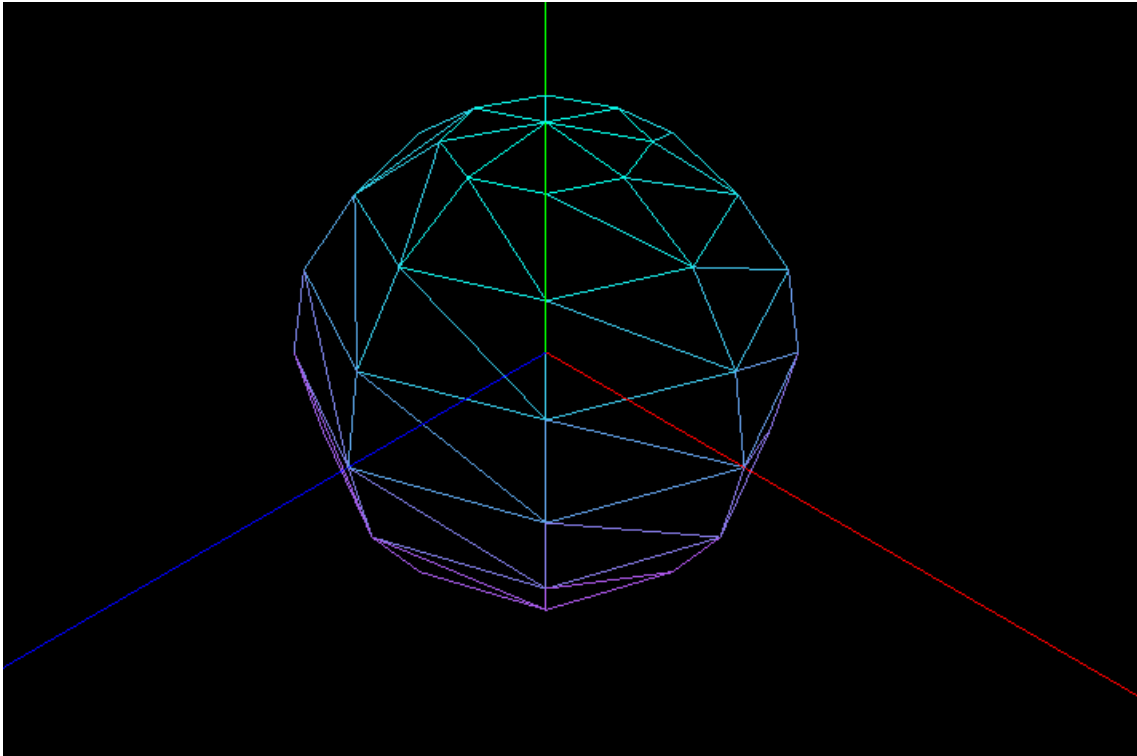
        coords.push_back(p4);
    }

    current_slice_alpha = next_slice_alpha;
}

current_stack_beta = next_stack_beta;
}

```

O referido previamente resulta numa esfera como a seguinte:



**Figura 7.** Esfera: Raio - 3 ; Slices - 8; Stacks - 8

## 3.5 Cilindro

Para a construção do cilindro é necessário receber o seu raio, a sua altura, o número de *slices* e o número de *stacks*.

De forma a concebê-lo é possível abordar uma estratégia muito semelhante à que foi utilizada no cone. No entanto, o raio deste não variará consoante a *stack* em que se encontra, mantendo-se sempre constante. Desta forma, deduzimos as seguintes equações de forma a ser possível a construção deste sólido:

$$altura\ da\ stack = \frac{altura\ do\ cilindro}{número\ de\ stacks} \quad (3.14)$$

$$\alpha = \frac{2\pi}{número\ de\ slices} \quad (3.15)$$



Tal como aconteceu na construção da esfera e do cone, a construção processou-se, de forma iterativa, com base em quatro pontos em que dois serão da *stack* atual (em que cada um será de uma slice) e os outros dois serão da próxima *stack* (em que cada um será de uma slice).

É ainda importante realçar que as condições *if* no algoritmo permitem a construção da base e do topo do cilindro.

Assim, obtém-se o seguinte algoritmo:

```
float stack_height = height / n_stacks;
float alpha = (2 * M_PI) / n_slices;

float curr_height = 0;
for (size_t stack = 1; stack <= n_stacks; ++stack) {
    float next_height = stack_height * stack;

    float curr_alpha = 0;
    for (size_t slice = 1; slice <= n_slices; ++slice) {
        float next_alpha = alpha * slice;
        // P1 --- P2
        // |       |
        // P3 --- P4
        Point p1 = Point::cartesian(
            radius * sin(curr_alpha), next_height, radius * cos(curr_alpha)
        );
        Point p2 = Point::cartesian(
            radius * sin(next_alpha), next_height, radius * cos(next_alpha)
        );
        Point p3 = Point::cartesian(
            radius * sin(curr_alpha), curr_height, radius * cos(curr_alpha)
        );
        Point p4 = Point::cartesian(
            radius * sin(next_alpha), curr_height, radius * cos(next_alpha)
        );

        coords.push_back(p4);
        coords.push_back(p2);
        coords.push_back(p1);

        coords.push_back(p4);
        coords.push_back(p1);
        coords.push_back(p3);

        if (stack == 1) {
            // draw base
            Point o = Point::cartesian(0, 0, 0);
            coords.push_back(p4);
            coords.push_back(p3);
            coords.push_back(o);
        }
    }
}
```

```

    }

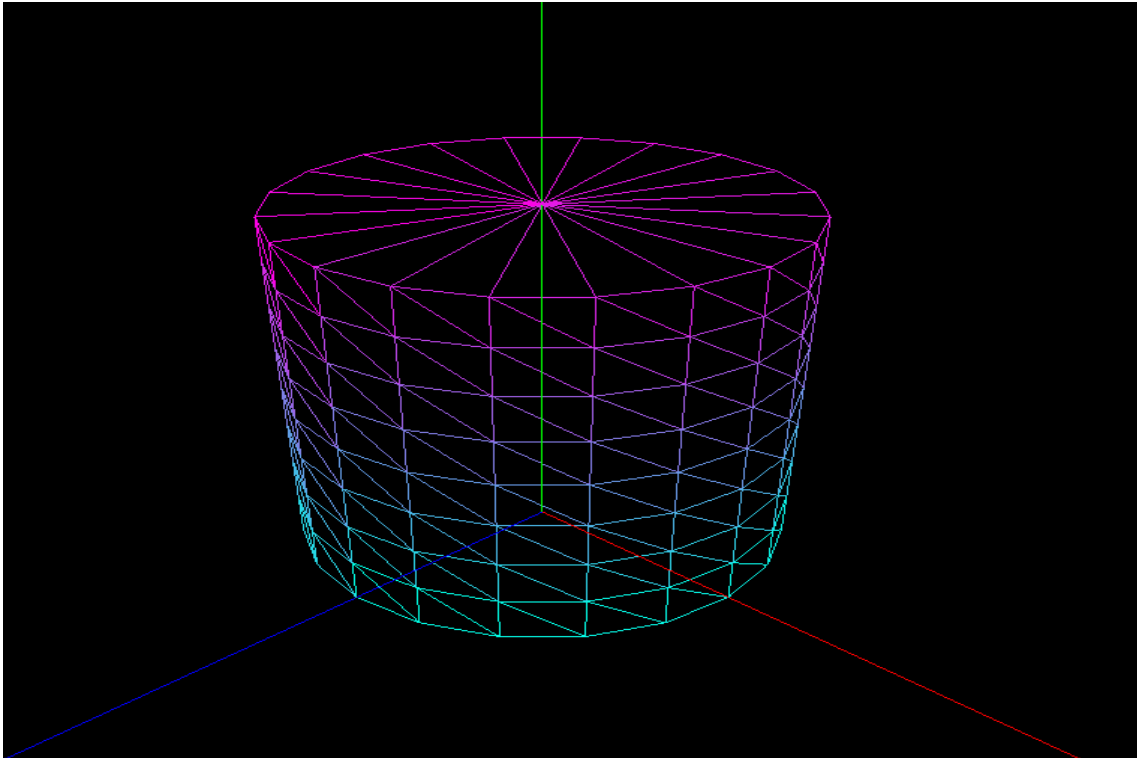
    if (stack == n_stacks){
        // draw top
        Point o = Point::cartesian(0, next_height, 0);
        coords.push_back(o);
        coords.push_back(p1);
        coords.push_back(p2);
    }

    curr_alpha = next_alpha;
}

curr_height = next_height;
}

```

O algoritmo anterior resulta numa representação semelhante à seguinte:



**Figura 8.** Cilindro: Raio - 3 ; Altura - 5; Slices - 20; Stacks - 8

## 3.6 Torus

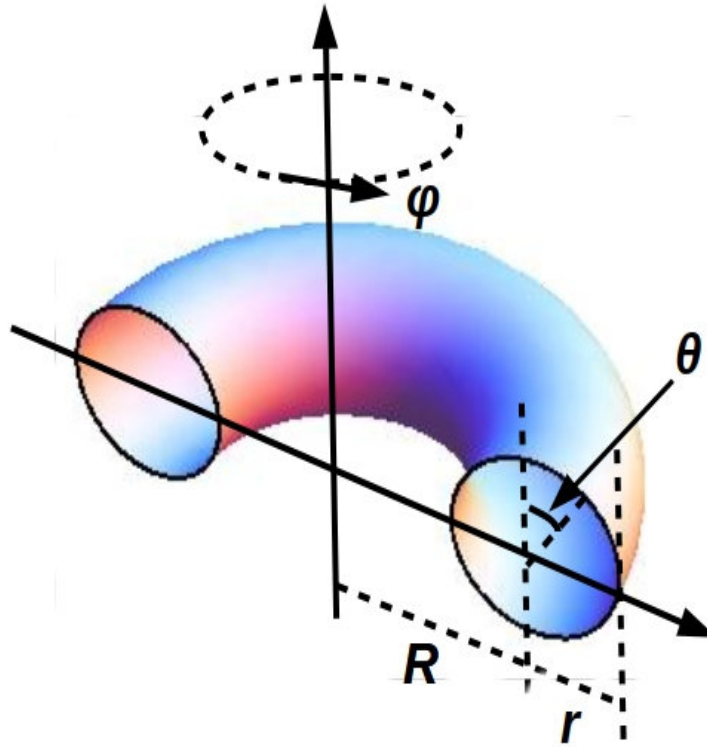
De forma a ser possível a construção do torus, figura semelhante a um *donut*, é necessário receber o seu raio, a distância do seu ponto central ao seu eixo de rotação, o número de *slices* e o número de *stacks*.

Além disso, para a sua construção, são ainda necessários os ângulos  $\varphi$ , que representa a rotação ao longo seu eixo de rotação, e  $\Theta$  que representa a rotação no seu "tubo". Tendo em conta que ambos deverão efetuar uma rotação completa, que  $\varphi$  dependerá do número de *slices* e que  $\Theta$  dependerá do número de *stacks*, obtemos as seguintes equações:

$$\text{passo do } \varphi = \frac{2\pi}{\text{número de slices}} \quad (3.16)$$

$$\text{passo do } \theta = \frac{2\pi}{\text{número de stacks}} \quad (3.17)$$

De notar que  $\Theta$  irá variar conforme a *stack* em que se encontra e que  $\varphi$  irá variar conforme a *slice* em que se encontra.



**Figura 9.** Representação do Torus

Assim, considerando que o seu raio é dado por  $r$  e que a distância do seu ponto central ao seu eixo de rotação é dada por  $R$ , temos que um ponto do torus é dado da seguinte forma:

$$P_{1x} = (R + r \times \cos(\theta \text{ da stack})) \times \cos(\varphi \text{ da slice}) \quad (3.18)$$

$$P_{1y} = r \times \sin(\theta \text{ da stack}) \quad (3.19)$$

$$P_{1z} = (R + r \times \cos(\theta \text{ da stack})) \times \sin(\varphi \text{ da slice}) \quad (3.20)$$

Assim, à semelhança daquilo que aconteceu na construção dos sólidos anteriores, a construção processou-se, de forma iterativa, com base em quatro pontos em que dois serão da *stack* atual (em que cada um será de uma *slice*) e os outros dois serão da próxima *stack* (em que cada um será de uma *slice*).

Obtém-se, então, o seguinte algoritmo:

```
double phi_step = 2 * M_PI / n_slices;
double theta_step = 2 * M_PI / n_stacks;

float curr_theta = 0;
for(size_t stack = 1; stack <= n_stacks; stack++){
    float next_theta = theta_step * stack;

    float curr_phi = 0;
    for(size_t slice = 1; slice <= n_slices; slice++){
        float next_phi = phi_step * slice;
        // P3 --- P4
        // |       |
        // P1 --- P2
        Point p1 = Point::cartesian(
            (out_radius + torus_radius * cos(curr_phi)) * cos(curr_theta),
            torus_radius * sin(curr_phi),
            (out_radius + torus_radius * cos(curr_phi)) * sin(curr_theta)
        );
        Point p2 = Point::cartesian(
            (out_radius + torus_radius * cos(next_phi)) * cos(curr_theta),
            torus_radius * sin(next_phi),
            (out_radius + torus_radius * cos(next_phi)) * sin(curr_theta)
        );
        Point p3 = Point::cartesian(
            (out_radius + torus_radius * cos(next_phi)) * cos(next_theta),
            torus_radius * sin(next_phi),
            (out_radius + torus_radius * cos(next_phi)) * sin(next_theta)
        );
        Point p4 = Point::cartesian(
            (out_radius + torus_radius * cos(curr_phi)) * cos(next_theta),
            torus_radius * sin(curr_phi),
            (out_radius + torus_radius * cos(curr_phi)) * sin(next_theta)
        );

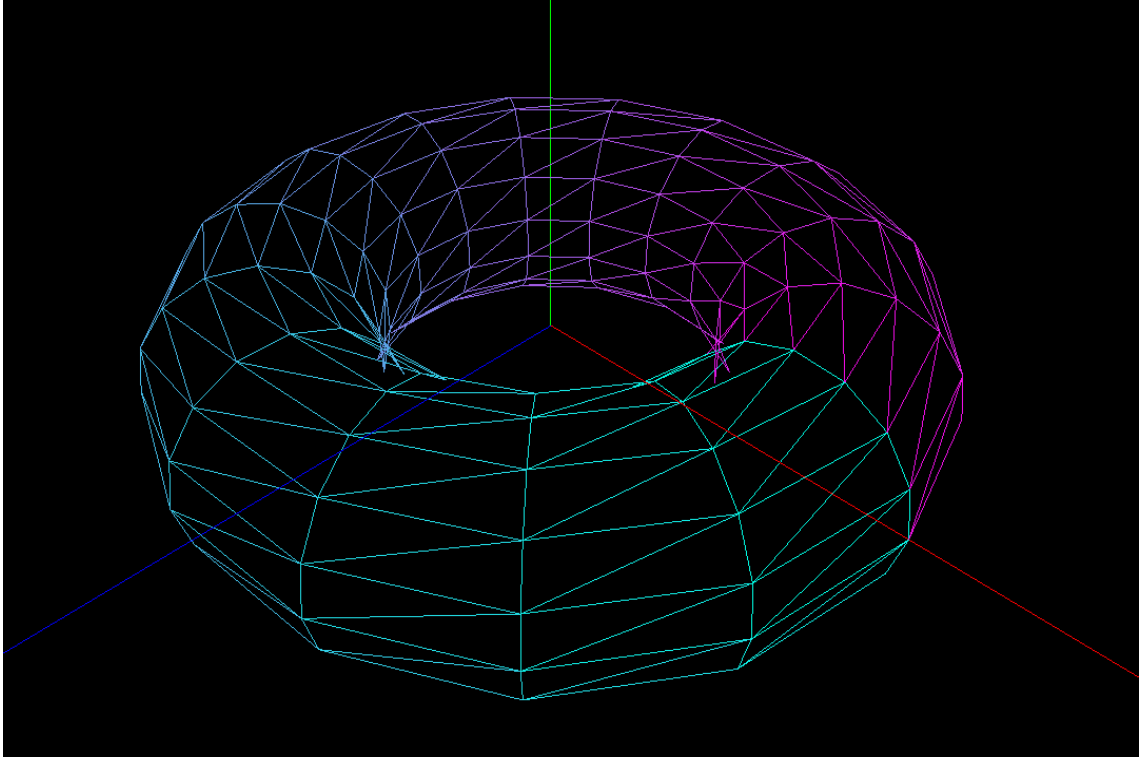
        coords.push_back(p1);
        coords.push_back(p2);
        coords.push_back(p4);

        coords.push_back(p2);
        coords.push_back(p3);
        coords.push_back(p4);

        curr_phi = next_phi;
    }

    curr_theta = next_theta;
}
```

Daqui podemos obter uma primitiva com aspeto semelhante ao seguinte:



**Figura 10.** Torus: Raio anel - 2 ; Raio da circunferência - 5; Slices - 15; Stacks - 15

## 4. Engine

Nesta fase, como forma de visualizar os sólidos gerados, foi desenvolvido um motor 3D simples, mas expansível. Este lê os ficheiros XML das *scenes*, coloca a câmara na posição especificada e permite ainda o movimento da câmara quer de uma forma orbital em torno de um ponto, como de uma forma mais livre, onde a câmara aponta para a posição do cursor e move-se nesta direção.

Uma vez que não existe iluminação nem texturas, optamos por adicionar cor aos polígonos, com base na sua posição.

### 4.1 Carregamento da *scene*

Para ler o ficheiro XML decidimos utilizar a biblioteca `tinycl2` devido à sua simplicidade e por ter apenas as ferramentas que necessitamos, não sendo muito pesada. Além disso decidimos utilizar a biblioteca `result` em vez de utilizar exceções, para obtermos um código mais limpo e legível.

Com este intuito também foram desenvolvidas as seguintes `macros`, que complementam a utilização do `result`:

```
#define CHECK_NULL(nullable, err)
do {
    if (!nullable) {
        return cpp::failure(err);
    }
} while (0)

#define CHECK_RESULT(result)
do {
    if (result.has_error()) {
        return cpp::failure(result.error());
    }
} while (0)
```

### 4.1.1 Carregamento das primitivas

Para ler um ficheiro de vértices, apenas é necessário passar o seu *path* à função `parse_model`, este lê o ficheiro e carrega os pontos para um vetor que é utilizado para inicializar a classe `Model`.

```
auto parse_model(std::string_view file_path) noexcept
-> cpp::result<Model, ParseError>
{
    std::ifstream file(file_path.data());
    if (!file.is_open()) {
        return cpp::failure(ParseError::PRIMITIVE_FILE_NOT_FOUND);
    }

    float x, y, z;
    auto points = std::vector<Point>();
    while (file >> x >> y >> z) {
        points.push_back(Point::cartesian(x, y, z));
    }

    return Model(std::move(points));
}
```

A classe `Model`, por sua vez, implementa um método *draw* que envia a informação dos vértices para o GPU, indicando que fazem parte de triângulos.

```
auto Model::draw() const noexcept -> void {
    glBegin(GL_TRIANGLES);
    auto max = (float) _points.size();
    auto i = 0.f;
    for (auto&& p : _points) {
        glColor3f(i / max, (max - i) / max, 0.9);
        ++i;
        glVertex3f(p.x(), p.y(), p.z());
    }
    glEnd();
}
```

### 4.1.2 Leitura do XML

Nesta versão inicial do *parser* apenas são lidos os campos necessários, reportando um erro se não se encontrarem disponíveis ou se estiverem mal formatados, e reportando um aviso se não estiverem presentes mas existir um valor *default* para ser utilizado.

A leitura do ficheiro é feita apenas uma vez ao inicializar o programa e a sua estrutura é armazenada em classes que replicam a hierarquia do XML. Com esta estrutura de dados torna-se simples efetuar transformações quer na câmara, quer nos objetos, além de ser facilmente expansível a propriedades que serão adicionadas nas próximas fases.

## 4.2 Câmara

Uma vez que o *OpenGL* trata das coordenadas da câmara como coordenadas cartesianas, decidimos guardar sempre pontos cartesianos, deste modo, não é necessário recalculas as coordenadas do parâmetro `lookAt` a todos os *frames*.

### 4.2.1 Orbital

Neste modo a câmara orbita em torno da origem. Para isso, a sua posição é tratada como um ponto esférico.

Assim, para mover a câmara horizontalmente apenas temos de alterar o valor de  $\alpha$ , e para mover verticalmente alterar o  $\beta$ . Para aproximar e afastar a câmara alteramos o raio.

Uma vez que a posição da câmara é armazenada como um ponto cartesiano, temos que calcular os valores do raio, de  $\alpha$  e de  $\beta$  do ponto esférico, para tal, utilizamos os seguintes métodos da classe `Point`:

```
float Point::radius() const noexcept {
    return sqrt(_x*_x + _y*_y + _z*_z);
}

float Point::alpha() const noexcept {
    return atan2(_x, _z);
}

float Point::beta() const noexcept {
    return atan(_y / sqrt(_z*_z + _x*_x));
}
```



Com isto, apenas temos que fazer as transformações necessárias consoante o *input* do teclado, limitar o  $\beta$ , para não exceder o campo de visão e recalcular a posição da câmara.

```
auto Camera::react_key_orbit(unsigned char key, int x, int y) noexcept -> void {
    auto radius = _eye.radius();
    auto alpha = _eye.alpha();
    auto beta = _eye.beta();

    switch (key) {
    case 'w':
        beta += 0.1;
        break;
    case 's':
        beta -= 0.1;
        break;
    case 'a':
        alpha -= 0.1;
        break;
    case 'd':
        alpha += 0.1;
        break;
    case '+':
        radius -= 0.5;
        break;
    case '-':
        radius += 0.5;
        break;
    default:
        return;
    }

    if (beta < -1.5) {
        beta = -1.5;
    } else if (beta > 1.5) {
        beta = 1.5;
    }

    _eye = Point::spherical(radius, alpha, beta);
}
```

### 4.2.2 FPV

Esta câmara funciona mudando o ponto para o qual a câmara está apontada, e movendo-a segundo a reta que liga esse ponto à posição atual da câmara.

Neste modo, no começo de cada *frame*, o cursor é movido para o centro da janela.

Quando há um movimento do cursor, é calculado o vetor que começa na posição da câmara e termina no ponto para qual a câmara está a olhar. Tratando este vetor como um vetor esférico, reparamos que se alterarmos as suas componentes  $\alpha$  e  $\beta$ , obtemos um vetor que inicia no mesmo ponto (na posição atual da câmara), mas termina num ponto diferente.

Subtraindo a posição atual do rato pelo centro da janela obtemos a quantidade de píxeis que o rato foi movido no *frame*. Como este valor é elevado, multiplicámo-lo por um fator de sensibilidade, e assim, este pode ser somado ao vetor calculado anteriormente.

Finalmente, apenas temos de realizar uma translação da posição da câmara sobre o vetor calculado e obtemos o novo ponto para o qual a câmara tem de apontar.

```
auto Camera::cursor_motion(int center_x, int center_y, int x, int y) noexcept
    -> void
{
    if (_mode == CameraMode::FPV && (x != 0 || y != 0)) {
        auto vec = _center - _eye;

        auto radius = vec.radius();
        auto alpha = vec.alpha();
        auto beta = vec.beta();

        alpha += (center_x - x) * 0.001;
        beta += (center_y - y) * 0.001;

        if (beta < -1.5) {
            beta = -1.5;
        } else if (beta > 1.5) {
            beta = 1.5;
        }

        _center = _eye + Point::spherical(radius, alpha, beta);
    }
}
```

Neste modo, o movimento é feito relativamente ao ponto ao qual a câmara está apontado.

Inicialmente, é calculado o mesmo vetor utilizado para alterar a posição da câmara. Será através deste que iremos alterar a posição da câmara. No entanto, como os seus valores dependem da distância da câmara ao ponto observado, temos de o normalizar antes de usar.

Depois de normalizado, este pode ser multiplicado por um fator que depende do *input* do teclado e, em seguida, somado à posição atual da câmara.

Para a distância entre a posição da câmara e do ponto observado se manter constante, também somamos o vetor calculado ao ponto observado.

```
auto Camera::react_key_fpv(unsigned char key, int x, int y) noexcept -> void {
    auto vec = _center - _eye;
    vec.normalize();
    switch (key) {
        case 'w':
            vec = vec * 0.5;
            break;
        case 's':
            vec = vec * -0.5;
            break;
        default:
            return;
    }
    _eye = _eye + vec;
    _center = _center + vec;
}
```

## 5. Conclusão e Trabalho Futuro

A implementação deste programa permitiu a familiarização com a linguagem de programação *C++* e com *OpenGL* e as ferramentas que este disponibiliza para a área de Computação Gráfica.

Considera-se que estão implementadas as bases para desenvolver futuramente um projeto mais complexo, que se sirva do *generator* e *engine* elaborados.

Futuramente, é pretendido melhorar a câmara implementada definindo um mecanismo que confira velocidades aos tipos de movimento, para tornar a representação mais realista e interessante.