

# Universidade do Minho

Licenciatura em Engenharia Informática

## Computação Gráfica

Phase 1 - Graphical Primitives

### Grupo 12

Ana Murta (A93284)  
Ana Henriques (A93268)  
Leonardo Freitas (A93281)  
Rui Coelho (A58898)

março, 2022

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>5</b>
<b>2</b>	<b>Enquadramento do Projeto</b>	<b>6</b>
<b>3</b>	<b>Generator</b>	<b>7</b>
3.1	Generator? What is that? . . . . .	7
3.2	Plano . . . . .	7
3.3	Cubo . . . . .	9
3.4	Esfera . . . . .	10
3.5	Cone . . . . .	12
3.5.1	A altura de cada stack . . . . .	13
3.5.2	O ângulo de rotação de cada slice . . . . .	13
3.5.3	O raio de cada stack . . . . .	13
3.5.4	How To BUILD! . . . . .	14
3.6	Cilindro - Extra . . . . .	16
3.7	Toro - Extra . . . . .	18
3.8	Ficheiro 3D . . . . .	19
3.9	Ficheiro XML . . . . .	20
<b>4</b>	<b>Engine</b>	<b>21</b>
4.1	Engine? What is that? . . . . .	21
4.2	Primitive & Point . . . . .	21
<b>5</b>	<b>Conclusão</b>	<b>23</b>

# Listas de Figuras

3.1	Dimensão com valor par . . . . .	8
3.2	Dimensão com valor ímpar . . . . .	8
3.3	Estratégia adotada para o plano . . . . .	8
3.4	plane 1 4 plane.3d . . . . .	8
3.5	Dimensão com valor ímpar . . . . .	9
3.6	box 1 4 box.3d . . . . .	10
3.7	Componentes da esfera. . . . .	10
3.8	Divisão da esfera em <i>slices</i> , <i>stacks</i> e triângulos. . . . .	10
3.9	Equações para calcular o tamanho de cada <i>step</i> . . . . .	11
3.10	Amplitude dos ângulos usados no cálculo dos pontos. . . . .	11
3.11	sphere 1 10 10 sphere.3d . . . . .	11
3.12	sphere 1 10 10 sphere.3d . . . . .	12
3.13	Teorema de Tales . . . . .	14
3.14	Cálculo de pontos para o cone . . . . .	15
3.15	cone 1 3 50 50 cone.3d . . . . .	15
3.16	<i>Slice</i> e raio do cilindro. . . . .	16
3.17	cylinder 1 2 100 cylinder.3d . . . . .	17
3.18	Raios do toro. . . . .	18
3.19	Equações paramétricas do toro. . . . .	18
3.20	torus 1 2 10 100 torus.3d . . . . .	19
3.21	Exemplo do <code>file.3d</code> . . . . .	19
3.22	Exemplo do conteúdo do <code>model.xml</code> . . . . .	20

4.1	Estrutura da classe <code>Point</code>	22
4.2	Estrutura da classe <code>Primitive</code>	22
4.3	<code>CMakeFile</code>	22

# Capítulo 1

## Introdução

Na primeira fase do trabalho prático, realizado no âmbito da disciplina de Computação Gráfica, foi desenvolvido um pequeno cenário gráfico 3D, usando algumas primitivas gráficas, nomeadamente: o plano, o cubo, a esfera e o cone. Como primitiva extra, também foi desenvolvida a primitiva do cilindro.

Para o efeito, foi requerido a implementação de duas aplicações: o **Generator** e o **Engine**. Enquanto que o *Generator* gera ficheiros 3d com as informações dos modelos, i.e., os vértices de cada primitiva, o *Engine* lê o ficheiro de configuração em XML e apresenta os modelos em 3D.

Para facilitar os testes e validar a construção das estruturas, foi implementada uma câmara de modo a ser possível observar cada uma das primitivas de várias perspetivas e ângulos diferentes.

Ao longo deste relatório, vão ser explicadas todas as estratégias adotadas para a construção de cada primitiva, suportando, sempre que necessário, esta explicação com fórmulas e imagens para demonstrar e simplificar o raciocínio aplicado.

## Capítulo 2

# Enquadramento do Projeto

Tendo como objetivo uma melhor organização do projeto, este foi estruturado em três diretorias: o *Generator*, o *Engine* e, por fim, o *Models*.

Na diretoria *Generator*, encontra-se o código que diz respeito à definição das coordenadas e, por conseguinte, a construção das primitivas. Existe, ainda, a classe `tinyxml2.cpp`, responsável pela configuração do ficheiro em XML.

Na diretoria *Models*, encontram-se os ficheiros .3d, onde estão guardados todos os pontos que definem os modelos, tendo estes sido fornecidos pelo *Generator*. Adicionalmente, assim que criados, estes ficheiros são guardados no ficheiro XML, contido, também, nesta diretoria, que posteriormente será utilizado pelo *Engine*.

Na diretoria *Engine*, encontra-se o código relativo à implementação em 3D dos vários modelos. Tal como supramencionado, esta diretoria irá ler o ficheiro XML da diretoria *Models* para aceder aos ficheiros .3d que contêm os pontos que definem os triângulos. Durante a leitura do XML e dos ficheiros .3d, as classes `Point.cpp` e `Primitiva.cpp` presentes nesta diretoria são utilizadas para armazenar os pontos numa estrutura. À semelhança da diretoria *Generator*, o *Engine* também tem uma classe `tinyxml2.cpp`.

# Capítulo 3

## Generator

### 3.1 Generator? What is that?

O Generator é responsável por gerar os vértices que definem os triângulos que dão origem às diferentes primitivas, seguindo os diferentes algoritmos de construção. Após serem gerados os vértices, estes são armazenados num ficheiro .3d, referido, por sua vez, no ficheiro XML.

Primeiramente, foi, então, criado um ficheiro **CMakeLists**, que permite gerar todo o projeto através do **CMake**. Relativamente ao tratamento e acesso dos ficheiros .3d e XML, foi necessário recorrer à biblioteca do **tinyXML2** e implementar um mecanismo para generalizar *paths* para os ficheiros, conferindo, assim, compatibilidade em diferentes máquinas. Para o efeito, foram usadas várias bibliotecas e manipulação de strings para obter o *path* pretendido.

### 3.2 Plano

A função **generatePlane** é responsável por gerar um plano em XZ, centrado na origem. Esta função recebe como argumentos os parâmetros passados pelo utilizador, que dizem respeito ao comprimento de cada lado e à dimensão do plano, escrevendo todos os pontos no ficheiro .3d identificado no comando. Como pertence ao plano XZ, o valor de y é 0 em todos os pontos. Relativamente às coordenadas x e z, estas tomam, inicialmente, os valores de **x = z = length/divisions**.

Consideremos o seguinte comando: `./projetoCG plane 3 4 plane.3d`

Neste caso, o plano terá 1 unidade de comprimento e será constituído por 4x4 quadrados. Já as coordenadas x e z tomam como valor inicial  $\frac{1}{4}$ .

A estratégia adotada encontra-se ilustrada nas figuras a seguir. Se a dimensão atribuída for par (figura 3.1 – 4x4), os pontos A e B distam, respetivamente,  $\frac{1*1}{4}$  e  $\frac{2*1}{4}$  da origem. Caso contrário, se esta for ímpar (figura 3.2 – 3x3), A e B passam a distanciar  $\frac{0.5*1}{3}$  e  $\frac{1.5*1}{3}$  da origem. Perante isto, atribuiriam-se valores a cada uma das linhas e colunas que constituem o plano para que, consoante os valores iniciais de x e z, possam ser calculados todos os seus pontos através de um ciclo.

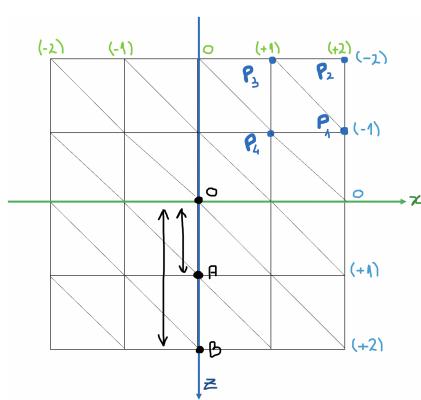


Figura 3.1: Dimensão com valor par

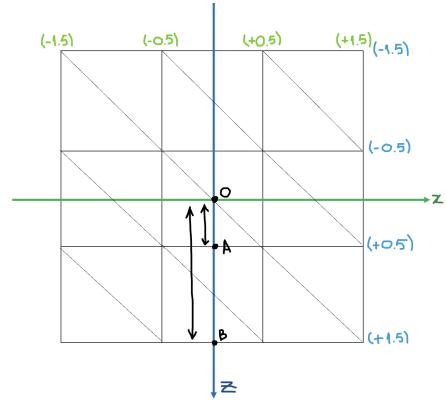


Figura 3.2: Dimensão com valor ímpar

Figura 3.3: Estratégia adotada para o plano

Seguindo o comando usado previamente como exemplo e aplicando a estratégia explicada, conseguimos definir os quatro pontos representados na figura 3.1:

$$P_1 (x*(+2), 0, z*(-1))$$

$$P_2 (x*(+2), 0, z*(-2))$$

$$P_3 (x*(+1), 0, z*(-2))$$

$$P_4 (x*(+1), 0, z*(-1))$$

Deste modo, o primeiro quadrado delimitado por estes quatro pontos é definido pela seguinte ordem, respeitando a regra da mão direita:  $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_1$ .

No final, obtemos a seguinte representação, gerada através do comando usado como exemplo:

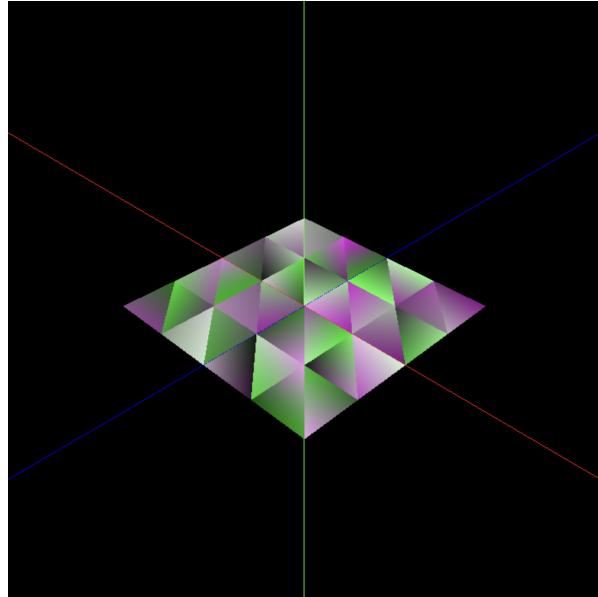


Figura 3.4: plane 1 4 plane.3d

### 3.3 Cubo

A função `generateBox` é responsável por gerar o cubo centrado na origem do referencial. Esta função recebe como parâmetros o comprimento de cada lado e a dimensão do cubo, escrevendo todos os pontos no ficheiro .3d identificado no comando.

A estratégia aplicada no desenho tridimensional do cubo consiste em gerar todas as faces ao mesmo tempo, sendo cada uma repartida em diversos triângulos.

Inicialmente, são atribuídos valores às variáveis *step* e *devistion*. A primeira representa o valor da deslocação de coordenada em coordenada. Já a segunda é utilizada para mover os pontos para uma determinada posição, colocando, assim, o cubo centrado na origem.

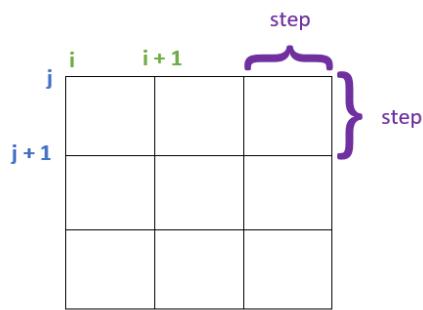


Figura 3.5: Dimensão com valor ímpar

Para a determinação dos pontos, considera-se que em cada linha temos um índice *j* e em cada coluna um índice *i*, possibilitando calcular as coordenadas do vértice, sendo que o modo de cálculo é independente da face em questão, aliás, este é o mesmo para todas as faces. Para além disto, para os cálculos das coordendas também é tido em conta os valores da variável *step* e *deviation*. Assim sendo, os cálculos utilizados são:

```

i * step - deviation
(i+1) * step - deviation
- deviation
j * step - deviation
(j+1) * step - deviation
length - deviation

```

Estes permitem, de acordo com a face que estamos a construir, determinar os valores de x,y e z que pretendemos.

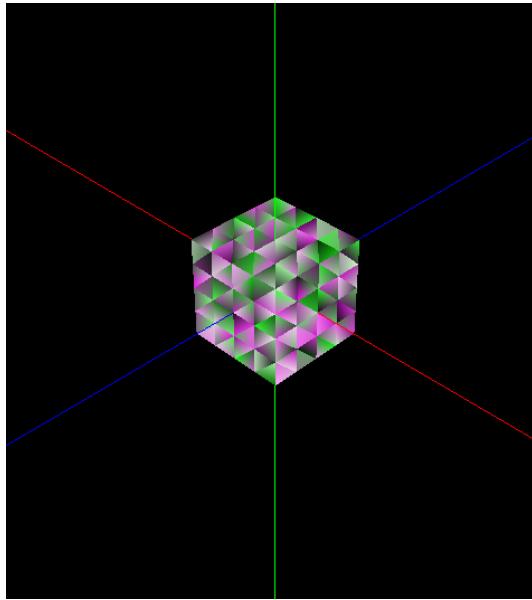


Figura 3.6: box 1 4 box.3d

### 3.4 Esfera

A geração de esferas é efetuada com recurso à função `generateSphere`, que permite a construção de uma esfera, centrada na origem do referencial. Através do *input* do utilizador, a função recebe os argumentos necessários para a construção da esfera: raio que descreve a esfera, número total de *slices*, número total de *stacks* e no nome ficheiro *.3d* onde serão registados os pontos calculados. A Figura 3.7 apresenta uma esfera, onde se encontram descritas estas variáveis referenciadas, particularmente, *stacks*, que descreverem os cortes em latitude, e *slices* descrevem os cortes longitudinais efetuados na esfera.

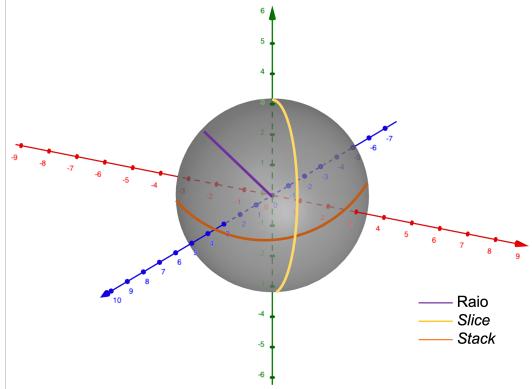


Figura 3.7: Componentes da esfera.

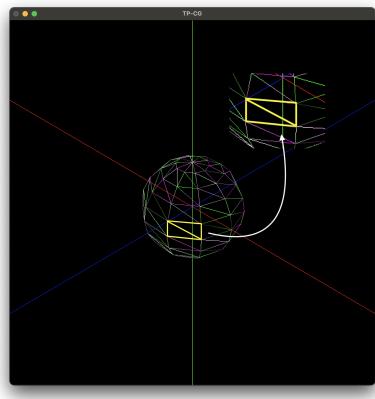


Figura 3.8: Divisão da esfera em *slices*, *stacks* e triângulos.

O desenho tridimensional da esfera consistiu na repartição da sua superfície em diversos triângulos, como é visível na Figura 3.8. Estes pontos, resultantes da intersecção entre *slices* e *stacks*, permitem a determinação dos triângulos usados no desenho.

Para a determinação dos pontos, consideram-se os ângulos  $\theta$ , com amplitude  $2\pi$ , e  $\phi$ , com amplitude  $\pi$ , descritos na Figura 3.10. Estes ângulos possibilitam o desenho completo na esfera, sendo usada uma fração de cada para determinar os vários pontos que compõem: para as *stacks* é definido um *stepStack* que determina a variação angular entre cada *stack* – acontecendo o mesmo para o caso das *slices*. As seguintes equações apresentadas na Figura 3.9 permitem efetuar o cálculo desse *step*<sup>1</sup>.

$$stepStack = \frac{endStack - startStack}{totalStacks}$$

$$stepSlice = \frac{endSlice - startSlice}{totalSlices}$$

Figura 3.9: Equações para calcular o tamanho de cada *step*

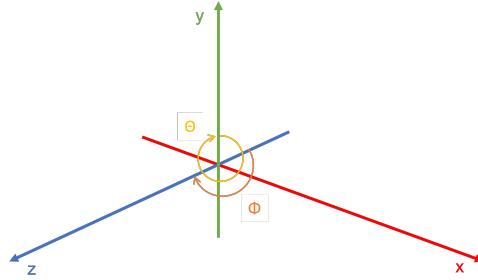


Figura 3.10: Amplitude dos ângulos usados no cálculo dos pontos.

Posto isto, o cálculo das coordenadas dos vários pontos é efetuado, iterativamente, com base nas funções trigonométricas apresentadas na Figura 3.11, tendo estes sido ordenados de acordo com a regra da mão direita.

$$x(\Theta, \Phi) = \cos(\Phi) * \sin(\Theta) * radius$$

$$y(\Theta, \Phi) = \cos(\Theta) * radius$$

$$z(\Theta, \Phi) = \sin(\Phi) * \sin(\Theta) * radius$$

Figura 3.11: sphere 1 10 10 sphere.3d

A Figura 3.12 resulta da execução do comando: `./ProjetoCG sphere 1 10 10 sphere.3d`

---

<sup>1</sup> *startSlice* corresponde ao ponto inicial da rotação, tendo sido escolhido o valor de 0.

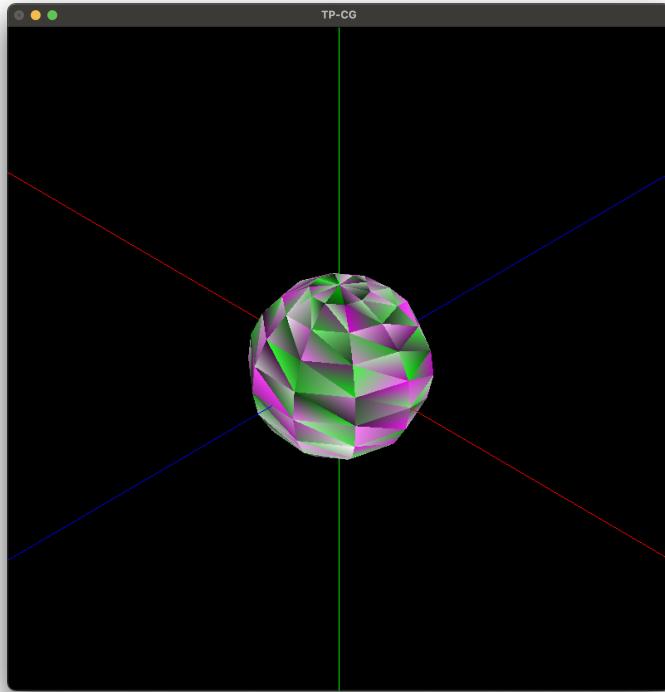


Figura 3.12: sphere 1 10 10 sphere.3d

### 3.5 Cone

A função **generateCone** é responsável por gerar o cone. Esta função, conforme solicitado no enunciado da Fase 1 do projeto, recebe como parâmetros um raio (*radius*), uma altura (*height*), um número de stacks (divisões horizontais), que correspondem às camadas do cone, e um número de slices, correspondentes às fatias do cone (divisões verticais). Para desenhar o cone, iniciou-se o processo pela sua base (base esférica), subindo o valor da altura da stack (*HeightofStack*) a cada iteração. Finalmente, dividindo o cone em stacks, temos de ter continuamente em atenção a altura do cone. No caso dos slices, é preciso ter cuidado com o ângulo originado por cada partição. A ideia inicial seria dividir o cone em secções de iguais dimensões.

Considerando o seguinte comando: `./projetoCG cone 1 2 4 3 cone.3d`

Neste caso, o cone é centrado na origem, tendo um raio de 1 unidade e uma altura de 2 unidades de comprimento. Além disto, o cone encontra-se dividido verticalmente em 4 slices e horizontalmente em 3 stacks.

Adotando esta estratégia de construção, torna-se necessário determinar diferentes variáveis de modo que o cone seja construído de forma correta. No decorrer deste relatório, focar-nos-emos nas 3 variáveis principais necessárias para a esta abordagem do problema:

- A altura de cada stack *HeightofStack*;
- O ângulo de rotação de cada slice;
- O raio de cada stack;

### 3.5.1 A altura de cada stack

O cálculo que determina a altura de cada stack é trivial visto que é unicamente preciso relacionar a altura total do cone com o número de stacks que foram passadas como parâmetro.

A altura de cada stack pode ser, então, calculada utilizando a seguinte expressão:

$$\text{HeightofStack} = \frac{\text{height}}{\text{stacks}}$$

### 3.5.2 O ângulo de rotação de cada slice

Relativamente ao que está associado a cada slice, o cálculo do ângulo de rotação de cada slice é, também, simples quando se sabe o número de slices, que, mais uma vez, é passado como parâmetro. Assim sendo, a fórmula resume-se à seguinte expressão:

$$\text{angle} = \frac{2 \times \pi}{\text{slices}}$$

### 3.5.3 O raio de cada stack

**Conceito de Semelhança de Triângulos:** uma semelhança de triângulos consiste, de forma geral, na proporção entre dois ou mais triângulos, i.e., existe uma relação de proporcionalidade se e somente se todos os seus lados e ângulos internos forem proporcionais ao outro triângulo.

**Teorema fundamental da semelhança de triângulos/Teorema de Tales:** toda a reta paralela a um dos lados do triângulo que interceta os outros dois lados determina um segundo triângulo semelhante ao primeiro.

Tendo em consideração o Conceito de Semelhança de Triângulos e o Teorema de Tales, temos conhecimento suficiente para entender o procedimento para calcular o raio para cada stack.

Observemos a figura seguinte:

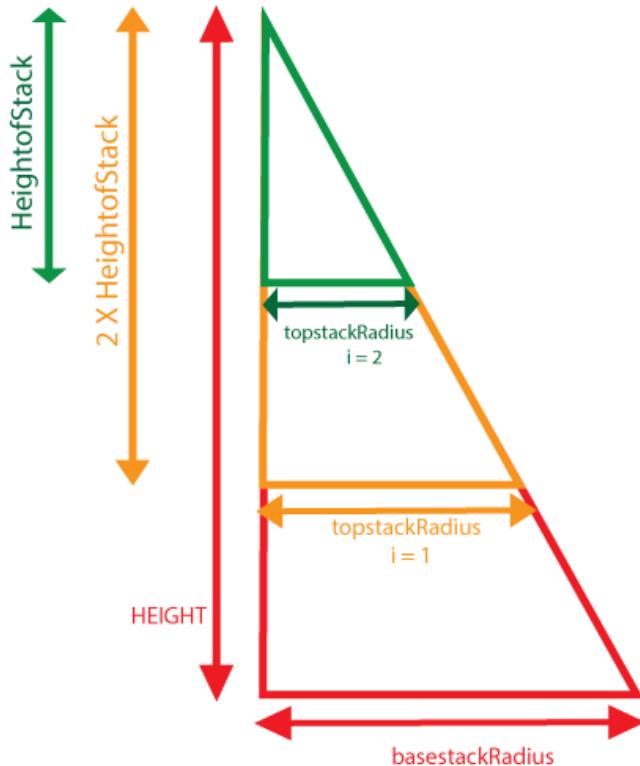


Figura 3.13: Teorema de Tales

Para aplicar o Teorema de Tales, representado na figura 3.13, usamos o triângulo formado pela *basestackRadius* e pela *height*. Posteriormente, usamos, à vez, cada um dos outros triângulos de cada stack. No nosso caso, já sabemos qual é a *HeightofStack*, sobrando apenas como incógnita o valor de *topstackRadius*. Isto traduz-se na seguinte expressão matemática:

$$\text{topstackRadius} = \frac{\text{basestackRadius} \times (\text{height} - (\text{HeightofStack} \times i))}{\text{height}}$$

$$0 \leq i < \text{stacks}, i \in N$$

### 3.5.4 How To BUILD!

O algoritmo usado para a construção do cone é simples: para cada stack e slice, serão calculados 4 pontos do plano por elas formadas. Assim que calculados, o passo a seguir é ordená-los, respeitando a regra da mão direita para obter os triângulos que representam o cone. Na figura 3.14, podemos ver como os pontos foram determinados e ordenados.

```

//Circunferencia de pontos, dados os slices a altura e o raio construido por stack
for (double i = 0; i < stack; i++) {
    double basestackRadius = ((radius * (height - (HeightofStack * i))) / height);
    double topstackRadius = ((radius * (height - (HeightofStack * (i + 1)))) / height);

    double angle = (2 * M_PI) / slices;

    for (int c = 0; c < slices; c++) {
        double alpha = angle * c;
        double alpha2 = angle * (c + 1);

        float p1x = cos(alpha) * basestackRadius;
        float p1y = HeightofStack * i;
        float p1z = -sin(alpha) * basestackRadius;

        float p2x = cos(alpha2) * basestackRadius;
        float p2y = HeightofStack * i;
        float p2z = -sin(alpha2) * basestackRadius;

        float p3x = cos(alpha) * topstackRadius;
        float p3y = HeightofStack * (i + 1);
        float p3z = -sin(alpha) * topstackRadius;

        float p4x = cos(alpha2) * topstackRadius;
        float p4y = HeightofStack * (i+1);
        float p4z = -sin(alpha2) * topstackRadius;
    }
}

```

Figura 3.14: Cálculo de pontos para o cone

Na figura 3.15 temos o *output* final se o ficheiro .3d for gerado e lido corretamente:

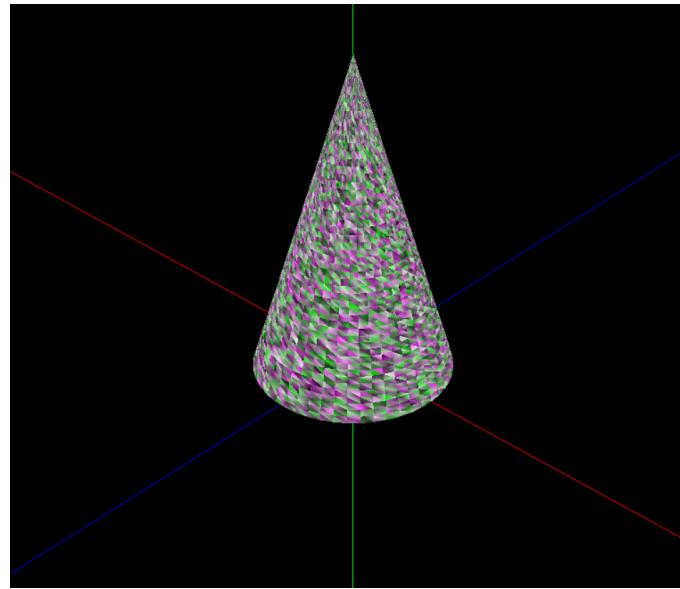


Figura 3.15: cone 1 3 50 50 cone.3d

### 3.6 Cilindro - Extra

Adicionalmente aos sólidos previamente apresentados, o grupo introduziu a construção de cilindros com recurso à função **generateCylinder**, que permite a construção de um cilindro centrado na origem do referencial. Para a criação do mesmo, o utilizador insere como argumentos o raio e altura do cilindro, o número de *slices* no qual este será dividido e o nome do ficheiro .3d para gravação dos pontos calculados. A Figura 3.16 demonstra os parâmetros considerados para a construção do cilindro.

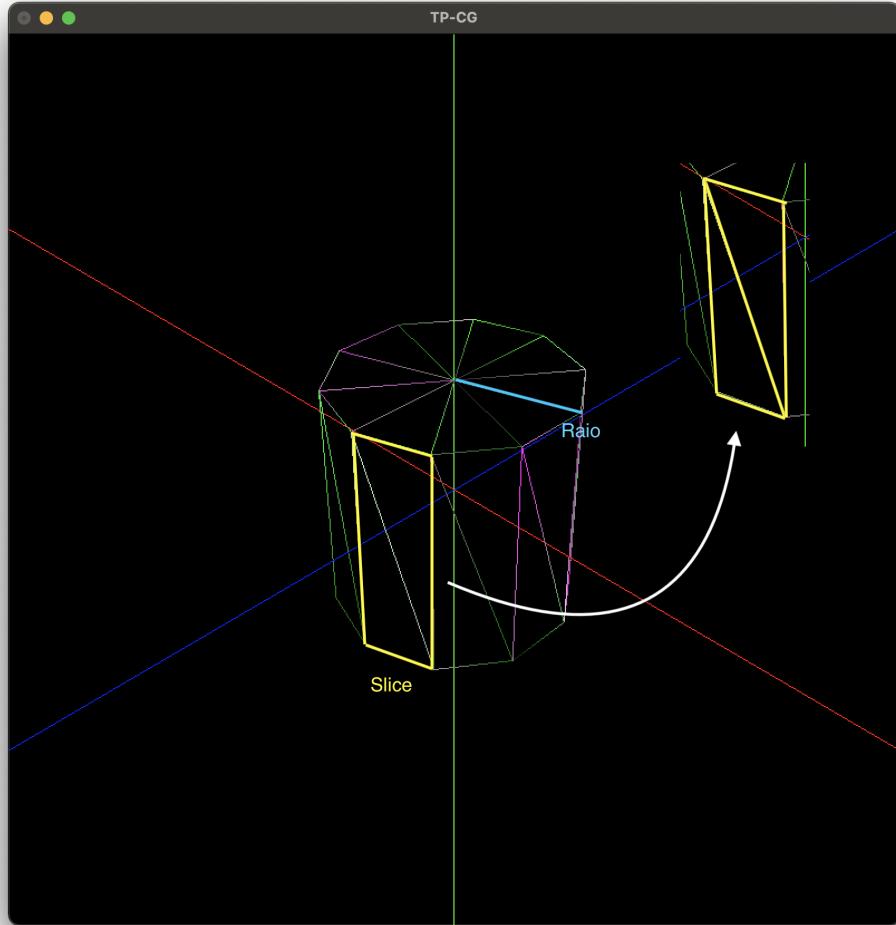


Figura 3.16: *Slice* e raio do cilindro.

A construção do cilindro centra-se, essencialmente, na construção simultânea de pontos na base e no topo – devidamente centrados, em função da origem. Para tal, foi determinado, tal como na esfera, um *step* que corresponde à amplitude angular em cada iteração de cálculo das coordenadas dos pontos. Para cada ponto, em cada iteração, apenas é necessário determinar as coordenadas correspondentes aos valores de  $x$  e  $z$  – que correspondem ao valor do ponto na circunferência da base/topo –, fazendo-se variar, apenas, o valor da coordenada em  $y$  para determinar o ponto correspondente à base, para  $y = -\frac{height}{2}$ , ou ao topo,  $y = \frac{height}{2}$ .

A Figura 3.17 resulta da execução do comando: `./ProjetoCG cylinder 1 2 100 cylinder.3d` e demonstra a visualização dos pontos calculados, dos triângulos gerados e, por fim, do cilindro colorido.

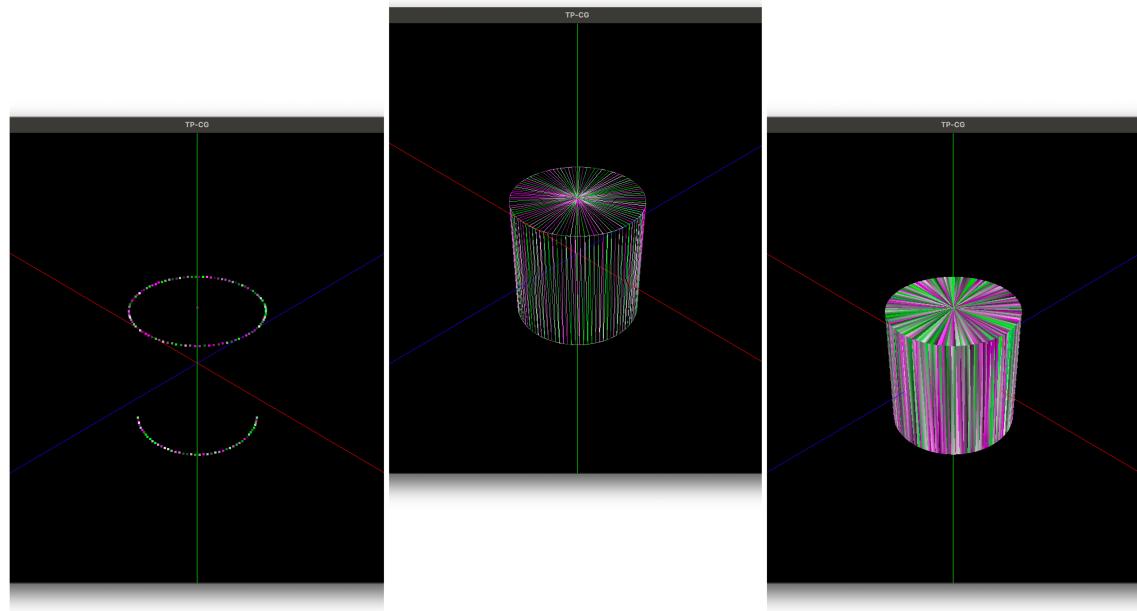


Figura 3.17: cylinder 1 2 100 cylinder.3d

### 3.7 Toro - Extra

Como funcionalidade adicional, o programa desenvolvido permite ainda a criação de um toro, vulgarmente chamado *donut*, com recurso à função **generateTorus**. A Figura 3.18 apresenta as medidas centrais para a construção do toro:  $r$ , que representa o raio do cilindro do toro, e  $R$ , que descreve a distância do centro do referencial ao centro do toro. Estes são dois dos parâmetros passados por *input* à função referenciada acima. Adicionalmente, ainda é necessário o número de *slices*, o número de *stacks* usados na construção do toro e o nome do ficheiro *.3d* onde serão guardados os pontos calculados.

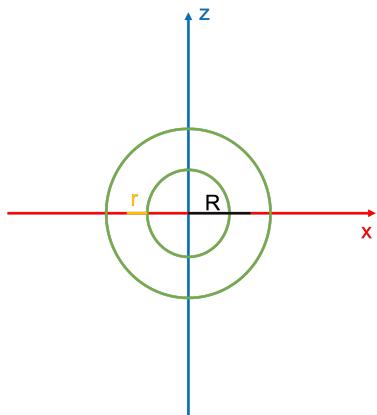


Figura 3.18: Raios do toro.

$$x(\Theta, \Phi) = (R + r * \cos\Theta)\cos\Phi$$

$$y(\Theta, \Phi) = \sin\Theta$$

$$z(\Theta, \Phi) = (R + r * \cos\Theta)\sin\Phi$$

Figura 3.19: Equações paramétricas do toro.

A determinação dos pontos usados na construção do toro centra-se num processo iterativo que percorre as diversas divisões formadas pelas *slices* e pelas *stacks*. Para cada um destes parâmetros, é definido um deslocamento angular – *stepSlice* e *stepStack* – que permite determinar os diversos pontos entre  $[0, 2\pi]$ , tanto em torno do toro, como em torno do cilindro que o define. Os diversos pontos, foram calculados com recurso às equações paramétricas apresentadas na Figura 3.19.

Após o cálculo dos pontos, estes foram ordenados para que fossem construídos de acordo com a regra da mão direita. A Figura 3.20 resulta da execução do comando: `/ProjetoCG torus 1 2 10 100 torus.3d` e demonstra a vizualização dos pontos calculados, dos triângulos gerados e, por fim, do toro colorido.

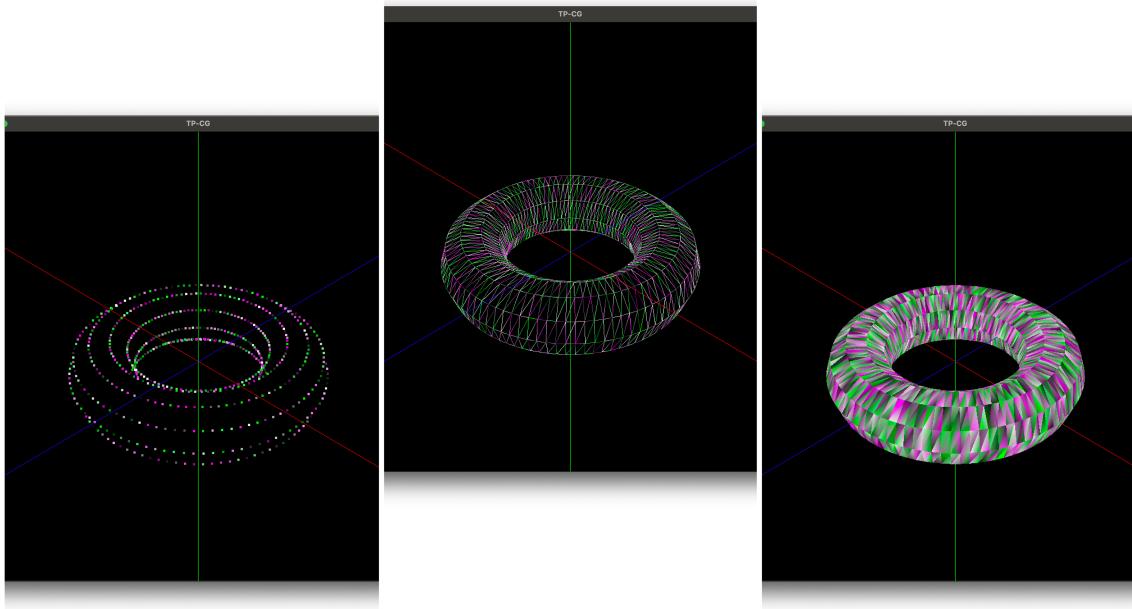


Figura 3.20: torus 1 2 10 100 torus.3d

### 3.8 Ficheiro 3D

Quando uma primitiva é produzida, é gerado um ficheiro .3d (figura 3.21), que segue sempre a mesma estrutura independentemente do tipo de figura construída.

A primeira linha deste ficheiro contém o número total de pontos que formam a figura em questão e, nas restantes linhas, encontram-se, então, todos os pontos calculados pelo *Generator*.

```
15000
0.980000,0.060000,-0.000000
1.000000,0.000000,-0.000000
0.992115,0.000000,-0.125333
0.980000,0.060000,-0.000000
0.992115,0.000000,-0.125333
0.972272,0.060000,-0.122827
0.000000,0.000000,0.000000
0.992115,0.000000,-0.125333
1.000000,0.000000,-0.000000
```

Figura 3.21: Exemplo do file.3d

### 3.9 Ficheiro XML

Conforme dito no enunciado proposto para esta fase, o Engine precisa de ler um ficheiro XML (figura 3.22), previamente gerado, para que possa ser gerada a primitiva. Neste ficheiro, é passada uma referência para cada ficheiro .3d, em que, no caso de já existir um ficheiro com aquele nome, este será reescrito com a nova informação. No entanto, a informação anterior não é apagada do XML, sendo desenhadas as duas figuras sobrepostas.

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

Figura 3.22: Exemplo do conteúdo do `model.xml`

# Capítulo 4

## Engine

### 4.1 Engine? What is that?

Até este ponto do relatório, foi abordada toda a camada do *Generator* responsável por gerar todos os pontos que definem as primitivas. Todavia, existe outra peça fundamental para que o programa seja funcional: o *Engine*.

O *Engine* funciona como um “artista” dado que lê os pontos que estão no ficheiro .3d e desenha os triângulos necessários para que a figura apareça completamente desenhada.

Inicialmente, o *Engine* acede ao ficheiro XML a partir do `tinyXML2`, lendo-o apenas uma vez. Após obter as primitivas escritas no XML e ler os respetivos ficheiros .3d, toda a informação contida nestes ficheiros é guardada em memória e, mais tarde, desenhada.

### 4.2 Primitive & Point

Perante a necessidade de guardar grandes quantidades de informação em memória, decidiu-se criar duas classes `Primitive` e `Point`, que contêm estruturas internas capazes de armazenar os dados necessários para o *Engine*.

A classe `Point` (figura 4.1) possui as variáveis de instância que denotam um ponto, i.e., armazena as coordenadas x, y e z de cada vértice. Como cada figura é constituída por vários pontos, a classe `Primitive` (figura 4.2) contém um vetor de elementos da classe `Point` onde são armazenadas as coordenadas dos triângulos que formam essa figura. Deste modo, cada objeto da classe `Primitive` será uma das primitivas acima explicadas.

No final, assim como no *Generator*, foi criada uma `CMakeFile` (figura 4.3) que permite compilar o projeto em diversos sistemas operativos e incluir os diferentes ficheiros utilizados.

```

class Point::PointBuilder {
    float x;
    float y;
    float z;

public:
    PointBuilder() {
        x = 0;
        y = 0;
        z = 0;
    }

    PointBuilder(float x, float y, float z) {
        this->x = x;
        this->y = y;
        this->z = z;
    }
}

```

Figura 4.1: Estrutura da classe Point

```

class Primitive::PrimitiveBuilder {
    //Variável privada que guarda os vértices
private:
    std::vector<Point> vertices;

public:
    PrimitiveBuilder() = default;

    PrimitiveBuilder(std::vector<Point> vertices) {
        for (size_t i = 0; i < vertices.size(); i++) {
            this->vertices.push_back(vertices.at(i));
        }
    }

    std::vector<Point> getVertices() {
        return vertices;
    }

    int getNrVertices() {
        return vertices.size();
    }

    Point getPoint(int index) {
        return vertices[index];
    }

    void addPoint(Point p) {
        vertices.push_back(p);
    }

    ~PrimitiveBuilder() = default;
};

```

Figura 4.2: Estrutura da classe Primitive

```

cmake_minimum_required(VERSION 3.5)
# Project Name
PROJECT(ProjetoCG)

set_property(GLOBAL PROPERTY USE_FOLDERS ON)
add_executable(${PROJECT_NAME} main.cpp tinyxml2.cpp tinyxml2.h Point.cpp Primitive.cpp Point.h Primitive.h)

find_package(OpenGL REQUIRED)
include_directories(${OPENGL_INCLUDE_DIRS})
link_directories(${OpenGL_LIBRARY_DIRS})
add_definitions(${OpenGL_DEFINITIONS})

if(NOT OPENGL_FOUND)
    message(ERROR "OPENGL not found!")
endif(NOT OPENGL_FOUND)

if (WIN32)
    message(STATUS "Toolkits_DIR set to: ${TOOLKITS_FOLDER}")
    set(TOOLKITS_FOLDER "" CACHE PATH "Path to Toolkits folder")

    if (NOT EXISTS "${TOOLKITS_FOLDER}/glut/GL/glut.h" OR NOT EXISTS "${TOOLKITS_FOLDER}/glut/glut32.lib")
        message(ERROR " GLUT not found")
    endif(NOT EXISTS "${TOOLKITS_FOLDER}/glut/GL/glut.h" OR NOT EXISTS "${TOOLKITS_FOLDER}/glut/glut32.lib")

    include_directories(${TOOLKITS_FOLDER}/glut )
    target_link_libraries(${PROJECT_NAME} ${OPENGL_LIBRARIES}
                           ${TOOLKITS_FOLDER}/glut/glut32.lib)

    if (EXISTS "${TOOLKITS_FOLDER}/glut/glut32.dll" )
        file(COPY ${TOOLKITS_FOLDER}/glut/glut32.dll DESTINATION ${CMAKE_BINARY_DIR})
    endif(EXISTS "${TOOLKITS_FOLDER}/glut/glut32.dll")

    set_property(DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR} PROPERTY VS_STARTUP_PROJECT ${PROJECT_NAME})
else (WIN32) #Linux and Mac

    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wno-deprecated -std=c++11")
    find_package(GLUT REQUIRED)
    FIND_LIBRARY(GLUT_LIBRARY GLUT )
    FIND_LIBRARY(OpenGL_LIBRARY OpenGL )

endif()

```

Figura 4.3: CMakeFile

# Capítulo 5

## Conclusão

A realização desta primeira fase do trabalho prático permitiu conhecer e trabalhar com a linguagem *C++*, bem como consolidar os conceitos lecionados nas aulas relativos ao conhecimento em ferramentas de Computação Gráfica, nomeadamente o *OpenGL* e o *GLUT*.

Concluída esta primeira fase, o grupo consegue identificar alguns aspetos positivos, como a construção de duas primitivas extra: o cilindro e o toro. Contudo, ao longo da realização desta fase, foram sentidas algumas dificuldades, principalmente em guardar o conteúdo dos ficheiros .3d em memória aquando da implementação da diretoria *Engine*, e em descobrir a ordem pela qual os pontos eram escritos nos ficheiros .3d de modo a respeitar a regra da mão direita.

Posto isto, consideramos que o resultado final foi conseguido com sucesso apesar dos diversos desafios enfrentados. Como trabalho a melhorar, o grupo gostaria de ter implementado a câmera que “navega pela superfície”, i.e., que se movimenta sobre o objeto fixo, permitindo, também, a entrada e saída da estrutura.