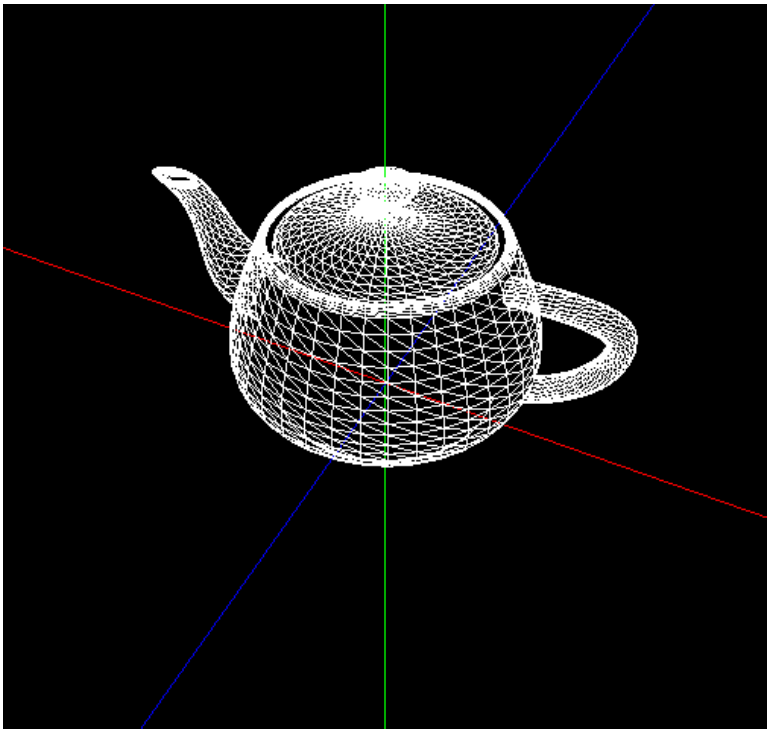

Relatório 1ª Fase | Computação Gráfica
Grupo 25 | 2024/2025

Afonso Dionísio	João Lobo	Rafael Seara	Rita Camacho
(A104276)	(A104356)	(A104094)	(A104439)

Índice

1. Introdução

O presente relatório apresenta informações relativas à **1ª Fase do Trabalho Prático** da Unidade Curricular **Computação Gráfica**, pertencente ao 2º Semestre do 3º Ano da Licenciatura em Engenharia Informática, realizada no ano letivo 2024/2025, na Universidade do Minho.

A 1ª Fase consistiu em implementar duas componentes essenciais do projeto, o *generator* e o *engine*, assim como algumas primitivas simples (plano, caixa, esfera e cone). Como extras, realizamos 3 primitivas adicionais (cilindro, torus e icosfera), mas também a importação de ficheiros `.obj`, 3 modos de utilização da câmera, e uma interface visual informativa.

2. Visão Geral da Arquitetura do Projeto

O projeto encontra-se dividido em dois programas principais: ***generator*** e ***engine***. Este primeiro tem como propósito a geração de figuras primitivas gráficas, resultando em ficheiros `.3d`. Esses ficheiros são, depois, embutidos em ficheiros `.xml`, lidos e renderizados pelo *engine*.

3. *Generator*

O programa *Generator* é responsável pela **geração de modelos .3d**, que serão posteriormente renderizados pelo *engine*. As figuras geradas são obtidas através de cálculos algébricos. Nesta fase, foi apenas necessário gerar posições relativas aos vértices do modelo.

Para invocar o programa, é necessário definir o nome do modelo a ser gerado e preencher os seguintes argumentos:

Model (Figura)	**Parameters** (Parâmetros)
sphere	radius (`float`), slices (`int`), stacks (`int`)
box	length (`float`), divisions (`int`)
cone	radius (`float`), height (`float`), slices (`int`), stacks (`int`)
plane	length (`float`), divisions (`int`)
cylinder	radius (`float`), height (`float`), slices (`int`), stacks (`int`)
torus	outer_radius (`float`), inner_radius (`float`), slices (`int`), stacks (`int`)
icosphere	radius (`float`), subdivisions (`int`)

O ficheiro `.3d` gerado tem um formato parecido ao `.obj`, onde para cada linha do ficheiro, esta começa com um identificador do seu tipo (e.g `v` para vértice) e os dados correspondentes. Para esta fase, como foi apenas necessário gerar informação relativa aos vértices do objeto, só há linhas deste tipo, sendo o formato então:

```
v <x> <y> <z>
```

Onde `x`, `y` e `z` indicam a posição do vértice no espaço.

O conjunto de **três vértices seguidos** corresponde a **um triângulo**.

```
v 0.5 0 0
v 0 0.5 0
v 0 0 0.5
```

Listagem 1: Exemplo de um ficheiro `.3d`

Numa próxima fase, onde será necessário adicionar funcionalidades novas ao formato `.3d`, a modularidade construída tornará mais fácil esse mesmo processo.

3.1. Figuras Primitivas Gráficas

Foram implementadas as figuras obrigatórias: **Caixa**, **Cone**, **Esfera** e **Plano** e, adicionalmente: **Cilindro**, **Torus** e **Icosfera**.

3.1.1. Plano

Para gerar um plano centralizado na origem, apoiado no plano xz, é necessário definir o **tamanho de um dos lados** (`float`) e o **número de divisões** (`int`). Com essas informações, o primeiro passo é calcular o tamanho de cada subdivisão:

$$\text{divisionSize} = \frac{\text{length}}{\text{divisions}}$$

Além disso, determinamos metade do comprimento para centralizar o plano na origem:

$$\text{halfLength} = \frac{\text{length}}{2}$$

O plano é construído iterando sobre as divisões ao longo dos eixos x e z. Inicialmente, quatro pontos são definidos para cada célula da grade.

$$\begin{aligned} P_1 &= (x * \text{divisionSize}) - \text{halfLength} \\ P_2 &= (x * \text{divisionSize}) - \text{halfLength} \\ P_3 &= ((x+1) * \text{divisionSize}) - \text{halfLength} \\ P_4 &= ((x+1) * \text{divisionSize}) - \text{halfLength} \end{aligned}$$

Estes pontos são deslocados ao longo do eixo z até atingir o número total de divisões e, em seguida, repetimos o processo ao longo do eixo x, formando os triângulos necessários para compor a malha.

Para garantir a visualização do plano em ambas as faces, foi necessário definir duas normais em sentidos opostos.

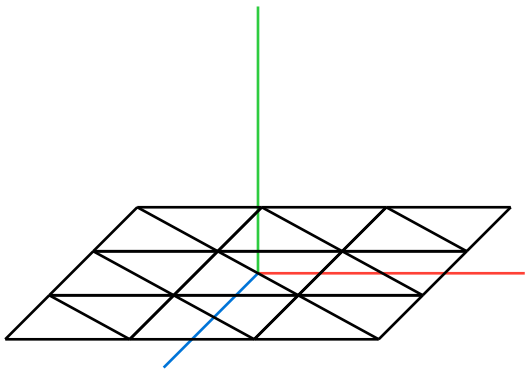

Figura 1: Geração de um plano

3.1.2. Caixa

Para gerar uma caixa centralizada na origem, são necessários dois parâmetros: o **tamanho total da caixa** (`float`) e o **número de subdivisões em cada face** (`inteiro`). O objetivo é dividir cada uma das faces da caixa em pequenas células quadradas, compostas por dois triângulos cada, formando uma malha triangular adequada para renderização.

Inicialmente, é calculado o semi-tamanho da caixa, que define a sua extensão desde o centro até uma das arestas:

$$\text{halfSize} = \frac{\text{size}}{2.0}$$

Em seguida, determina-se o tamanho de cada subdivisão (célula):

$$\text{step} = \frac{\text{size}}{\text{divisions}}$$

Uma caixa é composta por 6 faces: frente, trás, esquerda, direita, cima e baixo. Cada face é composta por `divisions x divisions` células, e cada célula é composta por dois triângulos. Cada triângulo é definido por 3 vértices, o que significa que cada célula adiciona 6 vértices à lista final.

O código percorre cada subdivisão nas duas direções de cada face. Para cada célula da grelha da face, são calculados os quatro cantos:

$$\begin{aligned} \text{v1} &= \text{-halfSize} + i * \text{step} \\ \text{u1} &= \text{-halfSize} + j * \text{step} \\ \text{v2} &= \text{v1} + \text{step} \\ \text{u2} &= \text{u1} + \text{step} \end{aligned}$$

Com estas coordenadas, são formados triângulos para cada face:

Face Frontal (Z positivo)

Os vértices são colocados com $z = \textit{halfSize}$.

Face Traseira (Z negativo)

Exatamente como a face frontal, mas com $z = -\textit{halfSize}$.

Face Esquerda (X negativo)

Nesta face, os quadrados percorrem os eixos Y e Z com $x = -\textit{halfSize}$.

Face Direita (X positivo)

Idêntico à face esquerda, mas com $x = \textit{halfSize}$.

Face Superior (Y positivo)

Aqui percorremos X e Z, com $y = \textit{halfSize}$.

Face Inferior (Y negativo)

Idêntico à face superior, mas com $y = -halfSize$.

Todos os vértices são acumulados numa lista (`vector<vec3> vertices`), que é depois devolvida como parte do modelo .3d final.

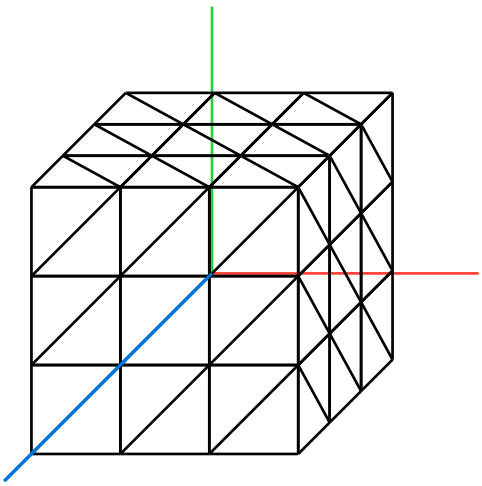

Figura 2: Geração de uma caixa

3.1.3. Esfera

Para a geração da esfera, centrada na origem, foram necessários como parâmetros o **raio** (`float`), ***slices*** (`inteiro`) e ***stacks*** (`inteiro`).

Foram utilizadas **coordenadas esféricas** no cálculo dos pontos, convertendo-as em coordenadas cartesianas através da seguinte fórmula:

$$f(r, \alpha, \beta) = (r\cos(\beta)\sin(\alpha), r\sin(\beta), r\cos(\beta)\cos(\alpha))$$

Para cada secção da esfera (composta pela *slice* e *stack* atuais), são calculados **4 pontos**, permitindo desenhar 2 triângulos:

$$P_1 = f\left(r, \text{slice\_atual} \times \frac{2\pi}{\text{slices}}, (\text{stack\_atual} + 1) \times \frac{\pi}{\text{stacks}} - \frac{\pi}{2}\right)$$

$$P_2 = f\left(r, (\text{slice\_atual} + 1) \times \frac{2\pi}{\text{slices}}, (\text{stack\_atual} + 1) \times \frac{\pi}{\text{stacks}} - \frac{\pi}{2}\right)$$

$$P_3 = f\left(r, \text{slice\_atual} \times \frac{2\pi}{\text{slices}}, \text{stack\_atual} \times \frac{\pi}{\text{stacks}} - \frac{\pi}{2}\right)$$

$$P_4 = f\left(r, (\text{slice\_atual} + 1) \times \frac{2\pi}{\text{slices}}, \text{stack\_atual} \times \frac{\pi}{\text{stacks}} - \frac{\pi}{2}\right)$$

Há ainda a necessidade de rodar a esfera 90 graus no eixo do z, de forma a que a função f funcione corretamente.

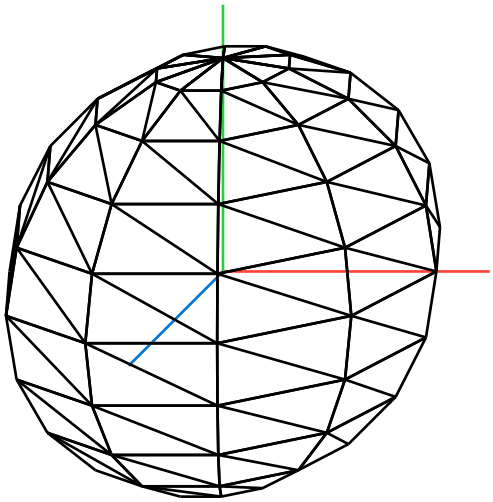

Figura 3: Geração de uma esfera

3.1.4. Cone

A geração do cone deve ser centrada na origem, sendo necessário indicar o **raio da base** (`float`), a **altura** (`float`), o **número de divisões ao longo da circunferência** (`slices`, `inteiro`) e o **número de subdivisões verticais** (`stacks`, `inteiro`).

Tal como nas outras figuras, começamos por calcular o tamanho do setor circular (`sliceSize`) e a altura de cada subdivisão (`stackSize`).

$$\text{sliceSize} = \frac{2 * \pi}{\text{slices}}$$

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

Inicialmente, definimos o centro da base e, de seguida, construímos a superfície lateral do cone. Cada subdivisão vertical reduz progressivamente o raio da base até chegar ao vértice do cone. Para isso, calculamos os raios das secções inferiores e superiores para cada `stack`:

$$\text{currentRadius} = \text{radius} - \text{stack} * \frac{\text{radius}}{\text{stacks}}$$

$$\text{nextRadius} = \text{radius} - (\text{stack} + 1) * \frac{\text{radius}}{\text{stacks}}$$

Cada fatia (`slice`) é composta por pequenas secções triangulares, geradas ao longo da altura do cone. Cada subdivisão é formada por dois triângulos que conectam quatro vértices adjacentes:

- `bottomLeft` e `bottomRight`: pertencem ao nível atual.
- `topLeft` e `topRight`: pertencem ao nível superior.

Os triângulos são formados conectando esses pontos:

- (`topLeft`, `bottomLeft`, `bottomRight`)
- (`topLeft`, `bottomRight`, `topRight`)

Além da superfície lateral, é necessário desenhar a base do cone. Para isso, utilizamos o centro da base e conectamos cada setor circular da base ao longo dos `slices`.

Cada fatia da base é formada pelos pontos `baseBottomLeft`, `baseBottomRight` e `baseMiddle`, garantindo que a base fica completamente preenchida.

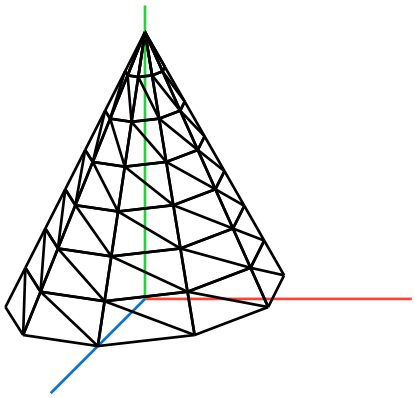

Figura 4: Geração de um cone

3.1.5. Cilindro

A geração do cilindro deve ser centralizada na origem, sendo necessário indicar o **raio** (`float`), a **altura** (`float`), o **número de *slices*** (`int`) e o **número de *stacks*** (`int`).

Tal como nas outras figuras, começamos por calcular o `sliceSize` e a metade da altura do sólido (`halfHeight`). Inicialmente, definimos os centros das bases inferior e superior e, em seguida, desenhamos as faces laterais do cilindro.

É importante ter em conta o número de *slices* e de *stacks*, uma vez que um menor número de *stacks* melhora o desempenho. Isto acontece porque desenhamos e calculamos menos pontos na construção do cilindro, permitindo a criação da estrutura com triângulos maiores, conforme ilustrado na figura.

Por fim, desenhamos também as bases superior e inferior, garantindo que o cilindro fica completamente fechado.

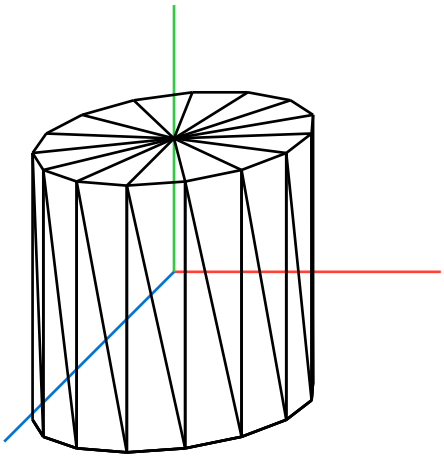

Figura 5: Geração de um cilindro

3.1.6. Torus

De forma a gerar o Torus, é necessário receber como argumentos o **raio, *R*** (`float`), o **raio do tubo, *r*** (`float`), ***slices*** (`inteiro`) e ***stacks*** (`inteiro`).

O cálculo dos pontos foi realizado a partir das seguintes fórmulas:

$$\varphi = \frac{\text{Slice\_atual} * 2\pi}{\text{Slices}}$$

$$\theta = \frac{\text{Stack\_atual} * 2\pi}{\text{Stacks}}$$

$$x = (R + r * \cos(\varphi)) * \cos(\theta)$$

$$y = r * \sin(\theta)$$

$$z = (R + r * \cos(\varphi)) * \sin(\theta)$$

Em cada secção, composta pela *slice* e *stack* atuais, são calculados 4 pontos (2 na *slice* atual, 2 na seguinte) que permitem desenhar 2 triângulos.

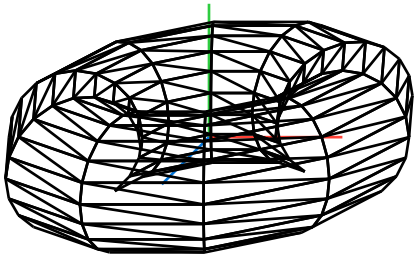

Figura 6: Geração de um torus

3.1.7. Icosfera

Para gerar a icosfera, são necessários os seguintes parâmetros: **raio** (`float`) e **subdivisões** (`int`).

Enquanto primitiva adicional, tivemos mais liberdade na sua geração. A icosfera começa como um **icosaedro** (icosfera com subdivisões = **1**), poliedro convexo com **12** vértices e **20** faces (triângulos equiláteros) e, a partir daí, procede-se à divisão de cada face em **4 novos triângulos equiláteros menores**.

Este processo permite, à medida que o número de subdivisões fornecido aumenta, aproximar-se cada vez mais de uma esfera, composta por apenas triângulos equiláteros (a geração ideal será entre 1 a 5 subdivisões, visto que o número de triângulos quadruplica com o aumento de 1 subdivisão, o que torna a geração muito pesada e complexa).

A transformação do antigo triângulo {A, B, C} em 4 novos triângulos menores é realizada encontrando **os pontos médios** dos 3 vértices iniciais do triângulo: A, B e C. Assim, são formados os novos vértices AB, AC e BC, originando consequentemente os novos triângulos:

{AB, A, AC}, {AB, BC, AC}, {B, AB, BC} e {BC, AC, C}.

Este processo é repetido até alcançar o número de subdivisões desejado.

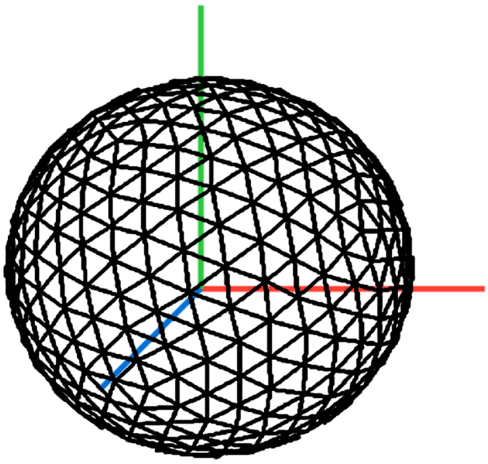

Figura 7: Geração de uma icosfera

4. *Engine*

O *Engine* é a aplicação responsável pela renderização de cenas compostas por modelos 3D (`.3d` ou `.obj`).

As cenas são guardadas através de ficheiros `.xml` que contêm a seguinte informação:

- **Janela**
  - Altura
  - Comprimento
- **Câmera**
  - Posição
  - Ponto de foco
  - `Up`
  - `FOV`, `Near` e `Far`
- **Modelos**

Estes ficheiros de cena são interpretados de forma estrutural com recurso à biblioteca `tinyxml`.

A estrutura da cena foi implementada de forma hierárquica, de modo a suportar futuras adições de objetos de tipos diferentes (como luzes).

4.1. Renderização de Modelos

Na fase de inicialização do programa, este lê todos os modelos utilizados na cena, e carrega os seus vértices para a memória. A cada `frame` renderizado, o programa itera pelos vértices e gera triângulos para cada grupo de 3 vértices. Esta implementação repete informação de vértices em memória, não sendo ideal. No entanto, será otimizada numa próxima fase ao implementarmos VBOs.

4.1.1. Parsing de `.obj`

Nesta fase, para podermos observar modelos mais complexos e diferentes, decidimos implementar o *parsing* de ficheiros *Wavefront* `.obj`. O *parsing* destes ficheiros é bastante parecido com o dos `.3d`, sendo composto por vértices e por faces. Ao contrário do nosso ficheiro para representar modelos 3D, o `.obj` não repete vértices para representar triângulos, utilizando em vez **faces**, que contêm os índices dos vértices que as formam. Sendo assim, o que o nosso programa faz é passar pelas faces e inserir os vértices utilizados em memória. Esta implementação é rudimentar, pois apenas renderiza modelos que só possuem faces triangulares.

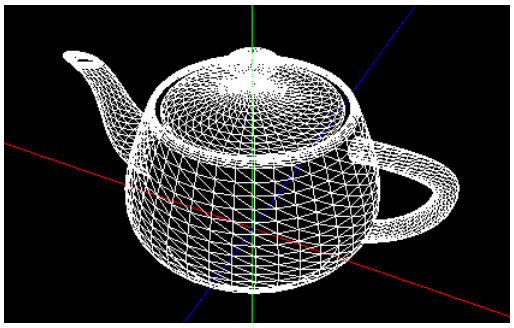

Figura 8: "*Utah Teapot*" - Martin Newell

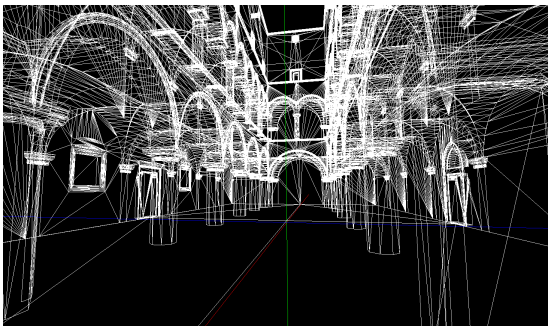

Figura 9: "*Sponza*" - Frank Meinl

4.2. Câmara

A câmara do `engine`, apesar de ser configurada inicialmente pela configuração da cena, pode ser alterada durante a execução do programa para utilizar outro modo. Para esta fase, foram implementados três modos:

4.2.1. *Static*

Neste modo, a câmara fica sempre na mesma posição. É o modo *default* do `engine`, sendo utilizado na inicialização das cenas.

4.2.2. *Turntable*

Neste modo, a câmera move-se de forma circular (em órbita) à volta de um ponto fixo. O utilizador pode ativar este modo pressionando a tecla `T`.

4.2.3. *Freecam*

Neste modo, ativado através da tecla `F`, o utilizador pode movimentar a câmara pela cena em primeira pessoa. Este suporta a rotação da mesma através de movimentos no `MOUSE`, e a alteração da sua posição através das teclas `W` (frente), `A` (esquerda), `S` (trás), `D` (direita), `SHIFT` (baixo) e `SPACE` (cima).

4.3. *UI*

O `engine` possui uma *interface* sob a cena a ser renderizada com informação útil. Esta permite que o utilizador troque de cena (selecionando um novo ficheiro `.xml`), e que veja o valor de `FPS` (*frames per second*) num dado momento, sendo útil para ir analisando a *performance* da implementação. Recorremos à biblioteca `Dear ImGui` para facilitar a implementação da *interface*.

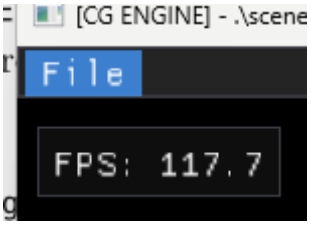

Figura 10: Medidor de `FPS`

5. Conclusão

Concluindo, acreditamos ter alcançado uma base bastante sólida na 1ª fase do projeto, o que nos permitirá avançar eficientemente nos próximos passos e tarefas, garantindo assim ótimos resultados nas fases seguintes.