

Universidade do Minho

Escola de Engenharia

Computação Gráfica

Trabalho Prático – Fase I

Grupo 3

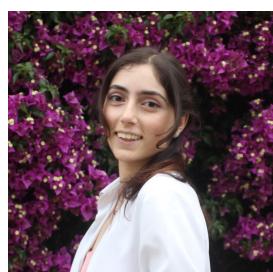
2 de março 2025



Ana Oliveira



Humberto Gomes



Mariana Cristino



Sara Lopes

A104437

A104348

A90817

A104179

Resumo

Para a realização deste trabalho prático, desenvolvemos duas aplicações, o **generator** e a **engine**. O **generator** é capaz de gerar modelos 3D de planos, cubos, esferas, e cones, como requerido pelo enunciado, mas também de outras figuras, como cilindros e tori. A **engine** é um programa que usa OpenGL [1] para apresentar cenas definidas em formato XML, contendo modelos no formato Wavefront OBJ [2], incluindo os criados pelo **generator**. Nesta fase, o nosso foco foi o de desenvolver uma arquitetura sólida para estes programas, utilizando já conceitos de OpenGL moderno, como VBOs e *shaders*. Esta arquitetura será expandida nas próximas fases, para suportar diversas outras funcionalidades, como transformações geométricas e movimento da câmara.

1 *Generator*

1.1 Funcionamento do programa

O programa **generator** é responsável por gerar modelos 3D de figuras geométricas, ou seja, ficheiros com os vértices e as faces triangulares destas figuras. Nesta fase, o **generator** deve ser capaz de gerar as seguintes figuras: planos, cubos, esferas e cones. Além de implementarmos a geração destas figuras, como funcionalidade adicional, também desenvolvemos um gerador de cilindros e de tori. As várias possibilidades de uso do comando **generator** são enumeradas abaixo:

```
./generator plane    <length>      <divisions>           <file>
./generator box      <length>      <grid>                <file>
./generator sphere   <radius>       <slices>      <stacks>    <file>
./generator cone     <radius>       <height>      <slices> <stacks> <file>
./generator cylinder <radius>       <height>      <slices> <stacks> <file>
./generator torus    <majorRadius> <minorRadius> <slices> <stacks> <file>
```

Internamente, o **generator** começa por interpretar os argumentos dados, saindo com a mensagem acima em caso de erro. Caso contrário, gera, em memória, os conjuntos de vértices e de faces que constituem o modelo 3D que, por fim, são escritos para um ficheiro no formato descrito abaixo.

1.2 Formato .3d

Decidimos que o formato de saída do `generator` seria o Wavefront OBJ [2]. O uso de um formato já existente apresentou diversas vantagens:

- Foi possível desenvolver a `engine` e o `generator` em paralelo: os desenvolvedores do `generator` não precisavam de ter a `engine` funcional para testar o seu código, visto que já existem diversas ferramentas para visualizar os ficheiros OBJ exportados pelo `generator`.
- Foi possível, sem qualquer código adicional, apresentar na `engine` modelos 3D oriundos de ferramentas de modelação, muito mais complexos do que os gerados pelo `generator`.
- A forma de representação de faces triangulares neste formato é facilmente mapeável para *index buffers* do OpenGL. Deste modo, não seriam necessárias alterações ao `generator` quando estes fossem implementados (algo que já foi feito nesta fase do trabalho).

O *parser* desenvolvido para este formato suporta apenas o essencial para o funcionamento desta primeira fase: posições de vértices e constituições de faces triangulares. O formato OBJ é textual, e representar um vértice é tão simples como ter uma linha começada por um `v`, ao qual se seguem as coordenadas do vértice separadas por espaços. O exemplo abaixo representa as coordenadas (0, 0.5, 1):

```
v 0 0.5 1
```

Faces triangulares são representadas em linhas começadas por um `f`, ao qual se seguem os índices dos três vértices da face. É de notar que a contagem dos índices começa em 1, e não em 0. Por exemplo, para representar uma face triangular, formada pelo primeiro, segundo e terceiro vértices do ficheiro, tem-se:

```
f 1 2 3
```

Visto que a estrutura de um ficheiro `.3d` separa os vértices de um modelo das suas faces, o processo de criação dos modelos 3D das várias figuras é dividido em duas fases: a geração do conjunto de pontos que os constituem, e o seu agrupamento em faces triangulares.

1.3 Plano horizontal

O primeiro passo para a geração da nuvem de pontos de um plano é o cálculo do comprimento de uma divisão, $d = \frac{L}{N}$, onde L simboliza o comprimento do plano e N o número de divisões. Depois, determinam-se os dois vetores que definem a direção do plano. Estes poderiam ser os vetores diretores dos eixos x e z , mas é mais simples que estes tenham o comprimento de uma divisão do plano.

$$\vec{i} = (d, 0, 0) \quad \vec{j} = (0, 0, d)$$

Depois, encontra-se o ponto do plano com os menores valores das coordenadas x e z . Como se deseja que o plano esteja centrado na origem, as coordenadas x e z do ponto desejado serão o simétrico da metade do comprimento do plano:

$$P_0 = \left(-\frac{L}{2}, 0, -\frac{L}{2} \right)$$

Depois, qualquer ponto P do plano pode ser definido como a adição a P_0 de uma combinação linear de \vec{i} e \vec{j} , limitando a números inteiros os coeficientes multiplicativos dos vetores:

$$P = P_0 + \alpha \vec{i} + \beta \vec{j} \quad \alpha, \beta \in \{0, 1, \dots, N\}$$

Na prática, o plano é gerado iterando pelos valores inteiros possíveis de α e de β , incrementando primeiro β , e só depois de α , o que dá origem a uma nuvem de pontos como a que pode ser observada na figura abaixo:

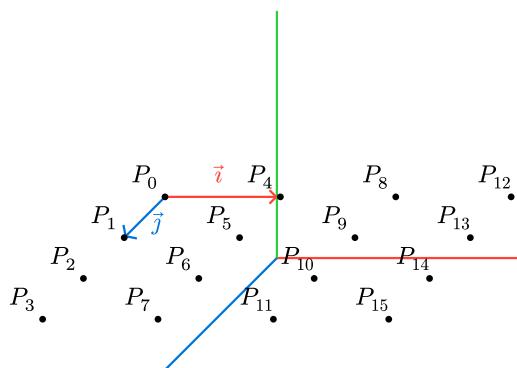


Figura 1: Nuvem de pontos resultante da geração de um plano de três divisões.

Depois, os vértices gerados podem ser agrupados nos triângulos que formam o plano. Para tal, começa-se com o primeiro vértice do plano, P_0 . Considera-se também o vértice seguinte, P_1 , e os dois vértices com os valores de z de P_0 e P_1 , mas com o seguinte valor de x possível (na "linha" seguinte). Um exemplo de um conjunto destes quatro vértices pode ser visto na figura abaixo. Com estes vértices, gera-se um quadrado, ou seja, dois triângulos. Este processo repete-se para todos os vértices onde é aplicável, ou seja, todos com exceção dos pontos com os maiores valores de x ou de z possíveis.

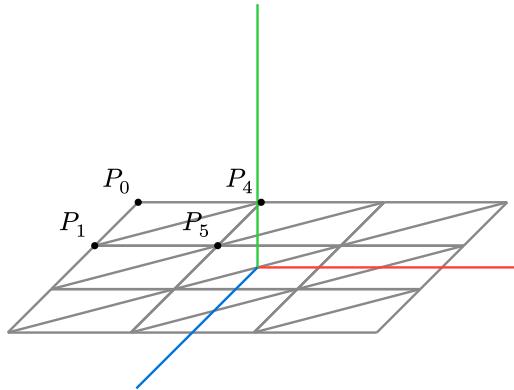


Figura 2: Triângulos de um plano gerado, e pontos utilizados na primeira iteração do ciclo de geração de triângulos.

Uma otimização feita pela `engine` é *face culling*, ou seja, desenhar apenas as faces voltadas para a câmara. No caso do plano, deseja-se que os seus triângulos estejam voltados para cima, pelo que os seus vértices devem estar ordenados na ordem contrária à dos ponteiros do relógio. Para os quatro pontos apresentados acima, os triângulos gerados são os seguintes:

$$T_1 = (P_1, P_4, P_0) \quad T_2 = (P_1, P_5, P_4)$$

1.4 Cubo

De um modo simples, o processo de geração de um cubo consiste na repetição da geração de um plano seis vezes, uma vez para cada face. No entanto, algumas diferenças devem ser evidenciadas. Tal como no plano, após ser calculado o comprimento de uma divisão do cubo, são definidos os vetores diretores dos vários planos que são as faces do cubo. Há três pares de vetores diretores, cada um utilizado para gerar duas faces opostas:

$$\begin{array}{lll}
\vec{i}_1 = (1, 0, 0) & \vec{j}_1 = (0, 1, 0) & \text{(faces dianteira e traseira)} \\
\vec{i}_2 = (0, 1, 0) & \vec{j}_2 = (0, 0, 1) & \text{(faces esquerda e direita)} \\
\vec{i}_3 = (0, 0, 1) & \vec{j}_3 = (1, 0, 0) & \text{(faces superior e inferior)}
\end{array}$$

Ao contrário do que acontece no plano, estes vetores encontram-se normalizados, o que é útil para determinar o ponto de cada face a partir do qual os seus restantes pontos serão gerados. Tal como acontece no plano, para o cubo estar centrado na origem, é necessário que todas as coordenadas deste ponto inicial estejam a uma distância de $\frac{L}{2}$ da origem, onde L representa o comprimento do lado do cubo. Por exemplo, para o primeiro par de vetores diretores, os pontos são os seguintes, correspondentes às faces traseira e dianteira, respetivamente:

$$P_{0^-} = \left(-\frac{L}{2}, -\frac{L}{2}, -\frac{L}{2} \right) \quad P_{0^+} = \left(-\frac{L}{2}, -\frac{L}{2}, +\frac{L}{2} \right)$$

De um modo geral, nas coordenadas onde \vec{i} ou \vec{j} têm um valor não nulo, a coordenada do ponto inicial é $-\frac{L}{2}$. A coordenada restante pode assumir, conforme a face, ou $\frac{L}{2}$ ou $-\frac{L}{2}$. Matematicamente, o vetor perpendicular à face é dado por:

$$\vec{n} = (1, 1, 1) - \vec{i} - \vec{j}$$

A partir deste vetor, os pontos iniciais das faces são dados por:

$$P_{0^-} = -\frac{L}{2}(\vec{i} + \vec{j} + \vec{n}) \quad P_{0^+} = -\frac{L}{2}(\vec{i} + \vec{j} - \vec{n})$$

Com os vetores diretores e os pontos iniciais de cada face, é possível normalizar os vetores diretores e gerar os pontos de cada face, seguindo o mesmo processo utilizado para o plano. Abaixo, apresenta-se um exemplo da nuvem de pontos de um cubo gerado:

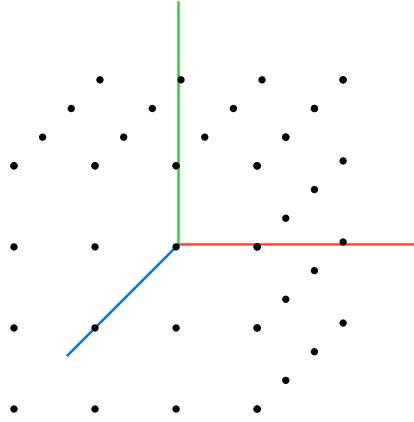


Figura 3: Nuvem de pontos resultante da geração de um cubo de três divisões. Os pontos das faces ocultas não foram representados nesta figura.

Depois de gerada a nuvem de pontos, o processo de geração dos triângulos de cada face é muito semelhante ao do plano. No entanto, a ordem dos vértices de cada triângulo difere conforme a face do cubo a ser construída. Considere-se o exemplo abaixo, o das faces superior e inferior de um cubo com apenas uma divisão:

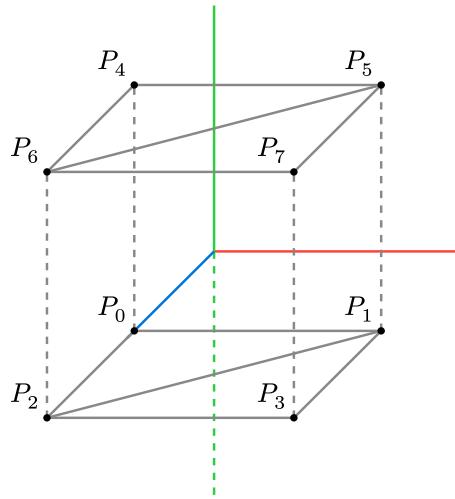


Figura 4: Faces superior e inferior de um cubo de apenas uma divisão.

Para que os triângulos da face inferior estejam voltados para o exterior do cubo, ou seja, para baixo, os seus pontos são ordenados no sentido dos ponteiros do relógio:

$$T_1 = (P_1, P_2, P_0) \quad T_2 = (P_1, P_3, P_2)$$

Na face superior, a ordem dos pontos dos triângulos é contrária, no sentido contrário ao dos ponteiros do relógio:

$$T'_1 = (P_5, P_4, P_6) \quad T'_2 = (P_7, P_5, P_6)$$

Para cada par de faces opostas, cada face será sujeita, conforme o seu vetor normal, a uma ordenação distinta dos pontos dos seus triângulos. Após aplicar o processo de geração de triângulos a todas as faces, é dada por concluída a construção do modelo do cubo.

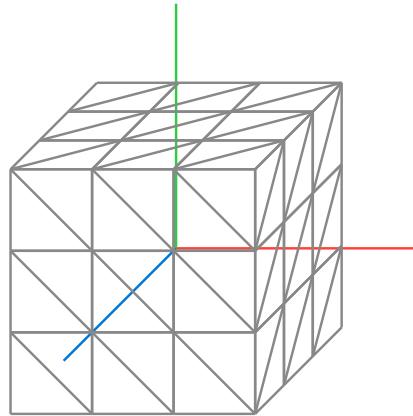


Figura 5: Triângulos resultantes da geração de um cubo de três divisões. As faces ocultas não foram representadas nesta figura.

1.5 Esfera

Para a construção da esfera, utilizámos um sistema de coordenadas esféricas, no qual a posição de cada ponto na esfera é determinada por dois ângulos: o ângulo polar, θ , e o ângulo azimutal, ϕ .

A parametrização de um ponto sobre uma esfera de raio r pode ser definida pelas seguintes equações [3]:

$$x = r \sin(\theta) \cos(\phi)$$

$$y = r \cos(\theta)$$

$$z = r \sin(\theta) \sin(\phi)$$

onde:

- θ é o ângulo polar, que varia entre 0 (pólo norte) e π (pólo sul), e que determina a latitude do ponto na esfera;

- ϕ é o ângulo azimutal, que varia entre 0 e 2π , e que define a longitude do ponto.

Este modelo matemático permite gerar uma esfera no espaço tridimensional. No entanto, para a sua implementação computacional, é necessário discretizar estes valores e definir um conjunto finito de pontos e faces que aproximem à sua forma contínua.

1.5.1 Geração dos vértices

Para discretizar a superfície da esfera, divide-se o intervalo de θ em *stacks* (fatias horizontais), e o intervalo de ϕ em *slices* (segmentos verticais). Assim, as incrementações angulares são calculadas da seguinte forma:

$$\Delta\theta = \frac{\pi}{N_{\text{stacks}}} \quad \Delta\phi = \frac{2\pi}{N_{\text{slices}}}$$

Ao percorrer os valores de θ e ϕ , é possível gerar um conjunto ordenado de pontos distribuídos sobre a esfera. A construção dos vértices segue os seguintes passos:

1. Criação do pólo norte: o primeiro ponto criado é o pólo norte, localizado em $(0, r, 0)$.
2. Geração dos pontos intermédios: para cada *stack* com exceção das extremidades (que são os pólos), ou seja, iterando-se sobre os valores de θ , determina-se a altura y dos pontos e o raio da secção circular correspondente:

$$y = r \cos(\theta)$$

$$r_{\text{secção}} = r \sin(\theta)$$

Para cada *slice*, ou seja, iterando-se sobre os valores de ϕ , é calculada a posição dos pontos da secção circular:

$$x = r_{\text{secção}} \cos(\phi)$$

$$z = r_{\text{secção}} \sin(\phi)$$

3. Criação do pólo sul: Após a geração das *stacks* intermédias, é criado o pólo sul em $(0, -r, 0)$.

1.5.2 Construção das faces

Após a geração dos vértices, é necessário definir as faces triangulares que compõem a superfície esférica. A triangulação é realizada da seguinte forma:

1. Ligação ao pólo norte: cada ponto da primeira *stack* (depois da do pólo norte) forma um triângulo com o pólo norte e o ponto seguinte na mesma *stack*. Para garantir a continuidade circular, o último ponto da *stack* liga-se novamente ao primeiro:

$$T_1 = (P_1, P_3, P_2)$$

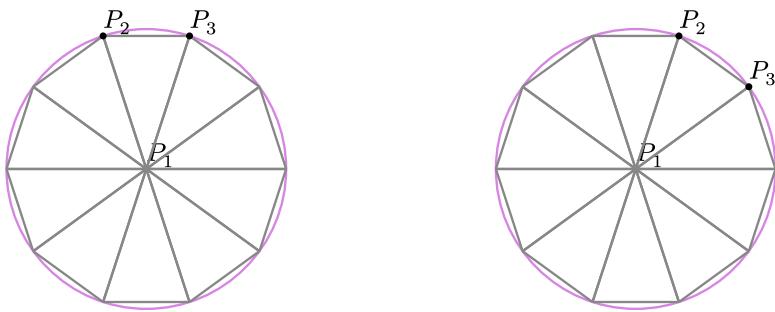


Figura 6: Ilustração da ligação dos vértices da primeira *stack* ao pólo norte, formando os triângulos iniciais da esfera. São apresentadas duas iterações deste processo.

2. Ligação das *stacks* intermédias: para cada fatia horizontal da esfera, os vértices são ligados de forma a criar duas faces triangulares por quadrilátero:

$$T_1 = (P_1, P_3, P_4) \quad T_2 = (P_1, P_4, P_2)$$

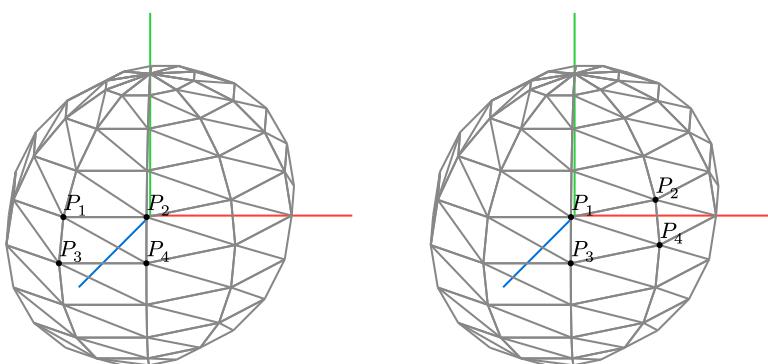


Figura 7: Triângulos de uma esfera gerada, e os pontos utilizados em duas iterações do ciclo de geração de triângulos.

Este método assegura uma ligação contínua entre os pontos e evita a repetição de vértices desnecessários.

3. Ligação ao pólo sul: o mesmo processo utilizado para o pólo norte é aplicado à *stack* mais próxima do pólo sul, assim garantindo que todas as fatias inferiores são corretamente ligadas ao vértice final. No entanto, a ordem dos vértices nos triângulos é invertida, em comparação com o pólo norte. Esta inversão garante que a normal de cada triângulo aponta para baixo, conforme ilustrado nos círculos da figura 6. No pólo norte, os vértices são ordenados no sentido anti-horário (por exemplo, P_1, P_3, P_2), enquanto que no pólo sul, a ordem é invertida (P_1, P_2, P_3).

1.6 Cone

Para gerar a nuvem de pontos do cone, é necessário ter em atenção dois vértices especiais, o centro da sua base e o vértice do seu topo, com as coordenadas abaixo:

$$P_0 = (0, 0, 0) \quad P_n = (0, H, 0) \quad , \text{ onde } H \text{ representa a altura do cone}$$

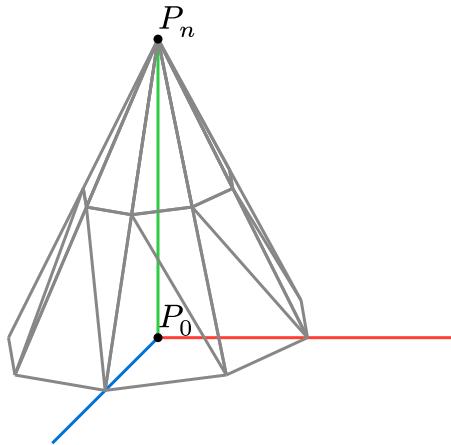


Figura 8: Primeiro vértice (P_0) e último vértice (P_n) de um cone.

Os restantes vértices são todos obtidos do mesmo modo: para cada *stack*, geram-se tantos vértices quantos o número de *slices*.

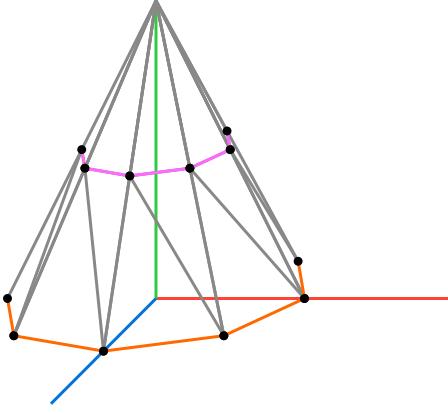


Figura 9: Duas *stacks* de um cone. Não são apresentados vértices ocultados pelas faces visíveis.

Para que as *stacks* sejam equidistantes uma da outra, a distância entre *stacks* será o quociente entre a altura do cone e o número de *stacks*. Por conseguinte, a coordenada y da i -ésima *stack* será:

$$y_i = \frac{H}{N_{stacks}} \times i \quad i \in \{0, 1, \dots, N_{stacks} - 1\}$$

Cada *stack* no modelo será uma aproximação da secção transversal do cone por um plano horizontal, ou seja a aproximação de uma circunferência. Na base do cone, o raio desta circunferência é o dado pelo utilizador, r . Este raio tende para 0 à medida que se caminha para o vértice do topo do cone. Logo, para a *stack* i , o raio da circunferência a aproximar pode ser calculado por uma interpolação linear:

$$r_i = \frac{H - y_i}{H} \times r \quad i \in \{0, 1, \dots, N_{stacks} - 1\}$$

Dentro de cada *stack*, as coordenadas x e z de cada ponto são determinadas de acordo com a equação trigonométrica de uma circunferência:

$$x = r_i \cos(\theta) \quad z = r_i \sin(\theta) \quad \theta \in [0, 2\pi[$$

Para se obterem tantos pontos quantas *slices*, e para que estes estejam uniformemente distribuídos pelo perímetro da circunferência, o ângulo θ é sempre incrementado em $\frac{2\pi}{N_{slices}}$ até o perímetro da circunferência ter sido coberto, dando origem à seguinte equação de geração de pontos:

$$x = r_i \cos \left(\frac{2\pi}{N_{\text{slices}}} \times j \right) \quad z = r_i \sin \left(\frac{2\pi}{N_{\text{slices}}} \times j \right) \quad j \in \{0, 1, \dots, N_{\text{slices}} - 1\}$$

Com os pontos das *stacks*, o centro da base, e o vértice do topo do cone, fica definida a nuvem de pontos desta superfície.

Depois de estar criada a nuvem de pontos, constroem-se, em primeiro lugar, as faces triangulares da base do cone. Para cada *slice*, gera-se uma face, que tem como vértices a origem (P_0), um vértice da base (P_1), e o vértice da *slice* seguinte (P_2), como mostra a figura abaixo:

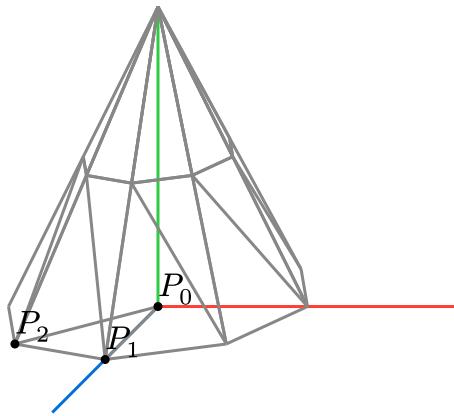


Figura 10: Primeira iteração da construção das faces da base de um cone (com 8 *slices* e 2 *stacks*).

Para que a normal desta face esteja voltada para fora do cone, ou seja, para baixo, os vértices do triângulo são ordenados do seguinte modo, lembrando que esta ordem também se aplica aos restantes vértices da base:

$$T = (P_0, P_1, P_2)$$

De seguida, constroem-se as faces laterais do cone, com exceção das da última *stack* (a de maior ordenada). É construído um quadrilátero (dois triângulos) por cada *slice* na *stack*, utilizando um vértice da *stack* (P_1), o seu adjacente seguinte na *stack* (P_2), e os dois pontos correspondentes a estes vértices na *stack* seguinte (P_9 e P_{10}). A figura abaixo mostra os vértices utilizados na primeira iteração deste processo:

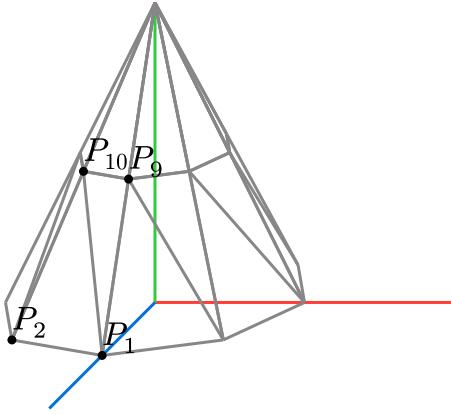


Figura 11: Primeira iteração da construção das faces laterais de um cone (com 8 *slices* e 2 *stacks*).

Para que os vetores normais destas faces apontem para fora do cone, utiliza-se a seguinte ordem para os pontos dos triângulos, nesta e noutras iterações:

$$T_1 = (P_1, P_9, P_{10}) \quad T_2 = (P_1, P_{10}, P_2)$$

Por último, constroem-se as faces da última *stack*. Ao contrário do que acontece com as restantes *stacks*, já não se constroem quadriláteros, mas sim triângulos. Para cada *slice*, constrói-se um triângulo, formado pelo vértice do cone (P_{17}), por um vértice da última *stack* (P_9) e pelo vértice seguinte nessa *stack* (P_{10}), como mostra a figura abaixo:

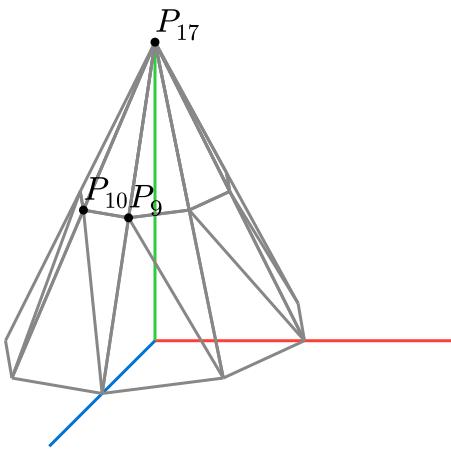


Figura 12: Primeira iteração da construção das faces da última *stack* de um cone (com 8 *slices* e 2 *stacks*).

Para que estas faces sejam visíveis do exterior do cone, os seus vértices são ordenados do seguinte modo:

$$T = (P_9, P_{17}, P_{10})$$

1.7 Cilindro

Para a construção do cilindro, utilizámos um sistema de coordenadas cilíndricas, onde a posição de cada ponto é determinada pelo raio, r , pelo ângulo azimutal, ϕ , e pela altura y .

A parametrização de um vértice sobre um cilindro de raio r e altura h pode ser definida pelas seguintes equações [4]:

$$x = r \cos(\phi)$$

$$y \in [0, h]$$

$$z = r \sin(\phi)$$

onde ϕ é o ângulo azimutal, que varia entre 0 e 2π , e que determina a posição do ponto ao longo da circunferência da base.

Este modelo matemático permite representar um cilindro tridimensionalmente. Para a sua implementação computacional, discretizam-se estes valores, e geram-se conjuntos finitos de pontos e faces que aproximam a sua forma contínua.

1.7.1 Geração dos vértices

Para discretizar a superfície do cilindro, o intervalo de ϕ é dividido em *slices* (segmentos verticais), enquanto que a altura y é dividida em *stacks* (fatias horizontais). Assim, os incrementos são calculados da seguinte forma:

$$\Delta\phi = \frac{2\pi}{N_{\text{slices}}} \quad \Delta y = \frac{h}{N_{\text{stacks}}}$$

Os vértices são gerados da seguinte forma:

1. Criação da superfície lateral: para cada *stack* ao longo da altura, itera-se sobre os valores de ϕ , determinando-se a posição dos pontos na circunferência da secção correspondente.

2. Criação dos vértices centrais das bases: Após gerar os pontos da superfície lateral, adicionam-se dois vértices centrais:

- O centro da base superior, localizado em $(0, h, 0)$.
- O centro da base inferior, localizado em $(0, 0, 0)$.

1.7.2 Construção das faces

Após a geração dos vértices, constroem-se as faces triangulares que compõem o cilindro. Esta etapa é realizada em três partes:

1. Construção da superfície lateral: cada quadrilátero entre duas *stacks* sucessivas é dividido em duas faces triangulares:

$$T_1 = (P_4, P_6, P_7) \quad T_2 = (P_4, P_7, P_5)$$

Isto garante uma cobertura uniforme da superfície lateral do cilindro.

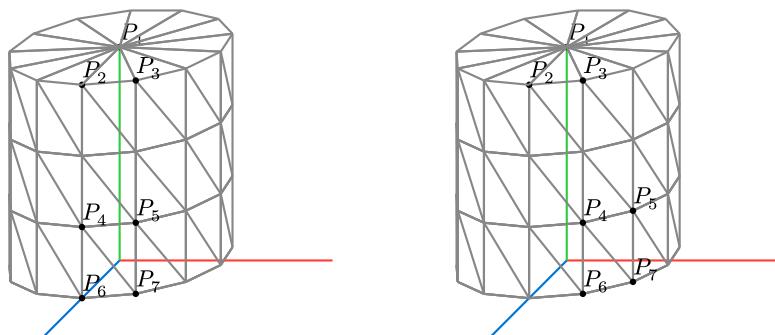


Figura 13: Triângulos de um cilindro gerado, e os pontos utilizados em duas iterações do ciclo de geração de triângulos das faces laterais.

2. Construção da base superior: cada ponto da última *stack* é ligado ao vértice central da base superior, de modo a formar triângulos radiais:

$$T = (P_1, P_2, P_3)$$

3. Construção da base inferior: o mesmo processo é repetido para a base inferior. No entanto, a ordem dos vértices não é a mesma que a utilizada na base superior. Por exemplo, caso os

pontos P_1 , P_2 e P_3 , da figura 8, estivessem na base inferior, o triângulo gerado seria (P_1, P_3, P_2) . Assim, as normais dos triângulos desta base apontem para baixo, o que é importante para o funcionamento correto do *back-face culling*.

1.8 Torus

Matematicamente, as coordenadas de um ponto do *torus* são definidas do seguinte modo [5]:

$$x = (-R + r \cos \phi) \cos \theta \quad y = r \sin \phi \quad z = (-R + r \cos \phi) \sin \theta$$

R e r são os parâmetros dados para a geração do *torus*, os raios maior e menor, respectivamente. Os ângulos θ e ϕ variam ambos no intervalo $[0, 2\pi]$.

O ângulo θ define a rotação do ponto em torno do eixo central do *torus*, percorrendo o círculo principal de raio R . Assim, ao variar θ , o ponto desloca-se ao longo do anel do *torus*, mantendo-se sempre na mesma posição relativa dentro do tubo. Já o ângulo ϕ determina a posição do ponto dentro da secção transversal do tubo, que tem raio r . Ao variar ϕ , o ponto descreve um movimento circular dentro do tubo, movendo-se ao longo da circunferência menor do *torus*.

Para gerar uma nuvem de pontos uniformemente distribuídos, dividem-se os ângulos em fatias (*slices*) e segmentos (*stacks*) iguais, utilizando:

$$\theta_i = i \cdot \frac{2\pi}{N_{\text{slices}}} \quad i \in \{0, 1, \dots, N_{\text{slices}}\}$$

$$\phi_j = j \cdot \frac{2\pi}{N_{\text{stacks}}} \quad j \in \{0, 1, \dots, N_{\text{stacks}}\}$$

Assim, o *torus* é gerado iterando sobre os valores inteiros possíveis de i e j , incrementando primeiro j e só depois i , o que dá origem à nuvem de pontos desejada.

Após a geração dos vértices, estes são agrupados nos triângulos que compõem a superfície do *torus*. Para isso, considera-se um vértice de referência, como P_1 , juntamente com o vértice adjacente na mesma fatia (P_3) e os dois vértices correspondentes na fatia seguinte (P_2 e P_4), e geram-se as duas faces triangulares do quadrilátero formado por estes quatro vértices. Este processo repete-se para todos os vértices onde é aplicável, garantindo que cada célula quadrangular é subdividida em dois triângulos. A figura seguinte ilustra esse processo, mostrando a

organização dos triângulos resultantes:

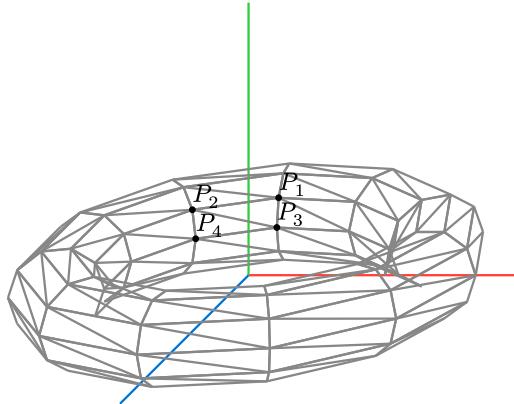


Figura 14: Triângulos de um *torus* gerado. As faces ocultas não são apresentadas.

Pretende-se que os triângulos do *torus* estejam voltados para fora, pelo que os seus vértices devem estar ordenados na ordem contrária à dos ponteiros do relógio. Para os quatro pontos apresentados acima, os triângulos gerados são os seguintes:

$$T_1 = (P_1, P_2, P_3) \quad T_2 = (P_2, P_4, P_3).$$

O *torus* é a única superfície côncava gerada, o que trás problemas relativos à orientação das suas normais. As normais de todas as faces apontam para "fora" do *torus*, ou seja, as das faces interiores ("do buraco") apontam para o centro do *torus*, e as das faces exteriores apontam no sentido contrário.

Idealmente, a parte interna do *torus* não deveria ser visível quando este é observado de fora, pois a própria geometria bloquearia essa visão. No entanto, como apenas os contornos das faces são desenhados, e a direção das normais é o único critério utilizado de momento para determinar se uma face é ou não desenhada, pode ocorrer um efeito onde a parte interna do *torus* é visível, como mostra a figura abaixo. Esse fenómeno acontece porque as normais da parte interna apontam para a câmara, tal como as normais das faces que a deveriam cobrir.

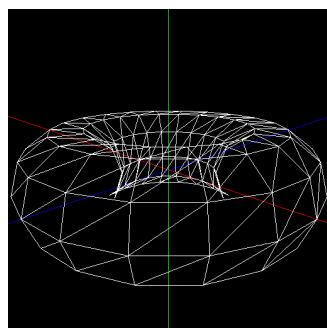


Figura 15: Faces do interior do *torus* visíveis.

Este problema será mitigado na fase 4 do trabalho, onde a superfície dos polígonos será preenchida por completo, e as faces internas do *torus* serão cobertas pelas faces externas frontais.

2 *Engine*

2.1 Criação da janela

Para a criação da janela, a biblioteca GLFW foi utilizada [6], escolhida principalmente pela sua API moderna. Ao contrário do GLUT [7], utilizado nas aulas práticas da UC, a API do GLFW não obriga a guardar o estado da aplicação em variáveis globais para que os métodos associados à janela lhe possam aceder. Isto possibilita a criação de abstrações RAII em torno da janela criada, o que por sua vez permite a utilização das várias funcionalidades de C++ para gestão automática de memória. Ademais, a API do GLFW também permite um maior controlo sobre o ciclo de resposta a eventos e renderização.

Já na primeira fase do projeto, optou-se por utilizar uma versão do OpenGL mais atual do que a que é utilizada nas aulas práticas, o *core profile* do OpenGL 4.6. Apesar do seu uso exigir um maior esforço na primeira fase, com a utilização obrigatória de VBOs e de *shaders*, esta versão suporta mais funcionalidades do que as versões anteriores, permite obter um melhor desempenho, e é suportada por depuradores como RenderDoc [8]. Para utilizar esta versão de OpenGL, um *loader* é necessário, e o GLAD [9] foi escolhido devido ao seu elevado grau de personalização.

2.2 Cena (Scene)

A classe **Scene** assume um papel fulcral na **engine**, pois é responsável por interpretar ficheiros XML com os conteúdos que devem ser renderizados. Para a sua análise sintática destes ficheiros, a biblioteca TinyXML-2 [10] é utilizada. Segue um exemplo de uma cena em formato XML:

```
<world>
    <window width="512" height="512" />
    <camera>
        <position x="3" y="2" z="1" />
        <lookAt x="0" y="0" z="0" />
        <up x="0" y="1" z="0" />
        <projection fov="60" near="1" far="1000" />
    </camera>
    <group>
```

```

<models>
    <model file="../models/box.3d" />
</models>
</group>
</world>

```

O processo de análise inicia-se com a leitura do elemento raiz, `<world>`, que contém todos os elementos de configuração. A `Scene` começa por extrair as dimensões da janela a partir do elemento `<window>` e as propriedades da câmara a partir do elemento `<camera>`. Depois, prossegue à leitura dos objetos presentes na cena. Os elementos `<group>` podem conter outros elementos `<group>`, ou então elementos `<model>`, que especificam os modelos 3D a serem carregados e renderizados. Nesta primeira fase, por simplicidade, esta estrutura hierárquica é linearizada durante o carregamento de uma cena (a `engine` armazena um `std::vector` de modelos), mas isto é algo que terá de ser mudado para suportar as transformações geométricas da próxima fase.

O leitor de cenas procura evitar o carregamento do mesmo modelo mais do que uma vez. Com recurso a um dicionário (`std::unordered_map`), são armazenados os modelos já carregados. Quando um elemento `<model>` é encontrado, caso este refira um ficheiro `.3d` carregado anteriormente, ele é reutilizado; caso contrário, o novo modelo é carregado e armazenado no dicionário.

2.3 Entidade (Entity)

A classe `Entity` representa um objeto 3D individual na cena, combinando um modelo (`Model`, uma abstração para um *buffer* de vértices na GPU) com outra informação relevante como, por exemplo, a sua cor. Esta classe é também responsável por renderizar cada objeto corretamente, transformando o seu modelo de acordo com os atributos definidos.

Uma otimização feita à `Entity` é o uso de um `std::shared_ptr<Model>` para referenciar o modelo associado a cada objeto. Este apontador inteligente permite a partilha do mesmo modelo por várias entidades que o usem, reduzindo o consumo de memória da GPU. Ademais, o uso de contagem de referências permite a libertação automática de memória, seguindo os princípios RAII de C++.

2.4 Câmara

A câmara é responsável por definir a perspetiva e o enquadramento da cena 3D. A sua configuração segue um modelo de câmara em primeira pessoa, com a posição, orientação e projeção definidas a partir do ficheiro XML da cena.

A posição da câmara é representada pelo vetor `position`, que define a sua localização no espaço tridimensional. A direção para onde a câmara está orientada é determinada pelo vetor `lookAt`, um ponto no espaço para onde a lente da câmara aponta. O vetor `up` especifica a orientação vertical da câmara, garantindo a sua correta rotação no espaço. Estes três vetores são utilizados para calcular a matriz de visualização.

Uma vez que este projeto usa *shaders*, não é possível utilizar a função `gluLookAt` [11], usada nas aulas práticas para o cálculo da matriz de visualização. No entanto, como utilizamos a biblioteca `glm` [12], é usada a função equivalente `glm::lookAt`, que devolve uma matriz que pode ser passada ao *shader* de vértices.

A projeção da câmara é controlada pelo campo de visão (*Field of View – FOV*) e pelos planos de recorte próximo e distante (`near` e `far`). O FOV define o ângulo de abertura da câmara, influenciando a sensação de profundidade da cena, enquanto os planos de recorte estabelecem os limites mínimo e máximo da região visível. A matriz de projeção é calculada com a função `glm::perspective`, que utiliza o *FOV*, o *aspect ratio* da janela, e os planos de recorte para definir a forma como os objetos são projetados no espaço 3D.

A matriz final da câmara resulta da multiplicação da matriz de projeção pela matriz de visualização, e é atualizada sempre que a janela é redimensionada. É utilizada para transformar todos os objetos da cena, garantindo um enquadramento adequado dos mesmos e a renderização correta da cena.

De momento, a `engine` não suporta que o utilizador controle a posição da câmara dinamicamente, movendo-a após a leitura do ficheiro XML. No entanto, pretendemos implementar esta funcionalidade na segunda fase do projeto.

2.5 Comportamento da engine

O primeiro passo na execução da `engine` é a criação da janela e do seu contexto OpenGL. Este é necessário para os passos seguintes: a leitura da cena, onde é necessário criar VBOs para os modelos 3D, a instanciação dos eixos do sistema de coordenadas, também suportados por

VBOs, e a criação dos *shaders* de vértices e de fragmentos. Uma vez que os VBOs e os *shaders* não são um objetivo da primeira fase deste trabalho, o seu funcionamento só será descrito em detalhe num relatório futuro.

Estando inicializada a `engine`, segue-se um ciclo em que a janela reage a eventos e desenha os seus conteúdos. Neste ciclo, são suportados os eventos de passagem de tempo, necessidade de atualizar os conteúdos da janela, e redimensionamento da janela. Para se subscrever a um evento, basta sobrescrever o seu método correspondente na classe `Window`:

```
virtual void onUpdate(float time, float timeElapsed) = 0;
virtual void onRender() = 0;
virtual void onResize(int width, int height) = 0;
```

De momento, apenas os métodos `onRender` e `onResize` são utilizados. Nestes, desenham-se os conteúdos da janela e atualiza-se a matriz de projeção da câmara, respetivamente. Em relação ao processo de renderização, este consiste no envio da matriz da câmara para o *shader* de vértices, na renderização dos eixos usando `glDrawArrays`, e na iteração por todos os objetos da cena, usando `glDrawElements` para desenhar cada um. No futuro, serão adicionados métodos de resposta a eventos de *input* do utilizador, que o permitirão executar ações como, por exemplo, mover a câmara.

3 Resultados obtidos

Nesta fase inicial do projeto, o foco principal foi o desenvolvimento da infraestrutura da `engine`, incluindo a implementação da *pipeline* de renderização e o carregamento de modelos 3D. Para demonstrar a funcionalidade da `engine`, foram gerados e carregados diversos modelos 3D, tanto simples figuras geométricas como modelos mais complexos obtidos na Internet.

3.1 Figuras geométricas geradas

Os seguintes modelos foram criados utilizando o `generator` implementado:

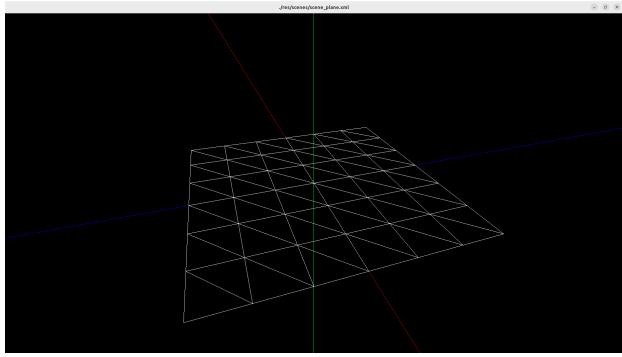


Figura 16: Plano (plane)

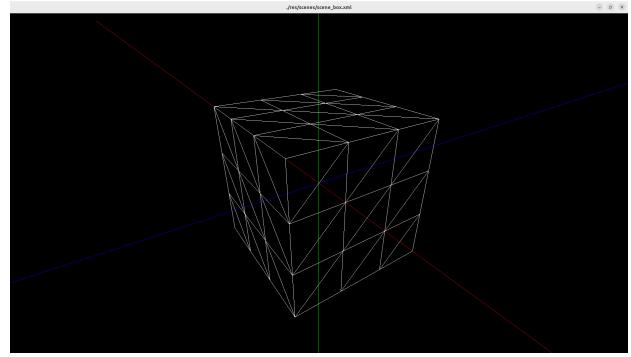


Figura 17: Cubo (box)

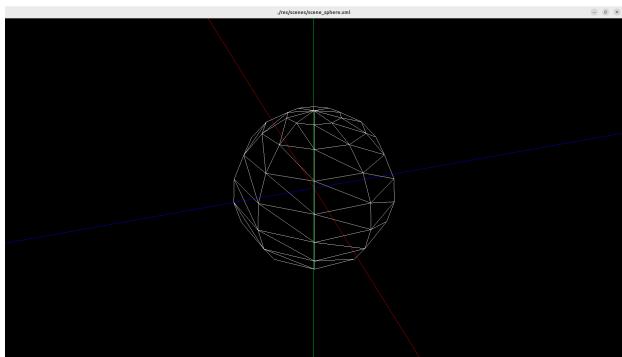


Figura 18: Esfera (sphere)

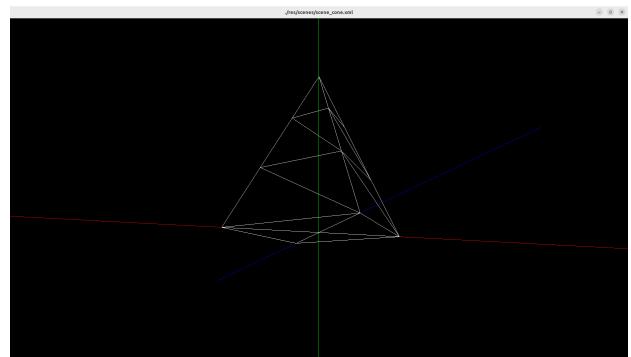


Figura 19: Cone (cone)

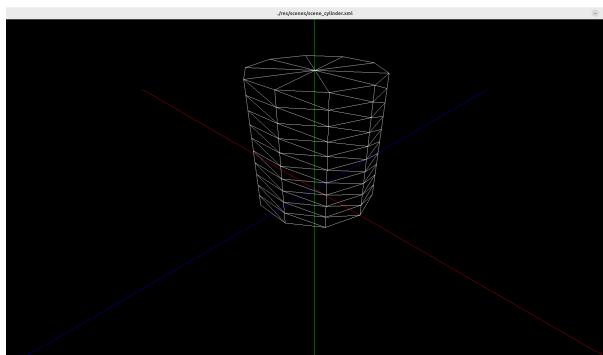


Figura 20: Cilindro (cylinder)

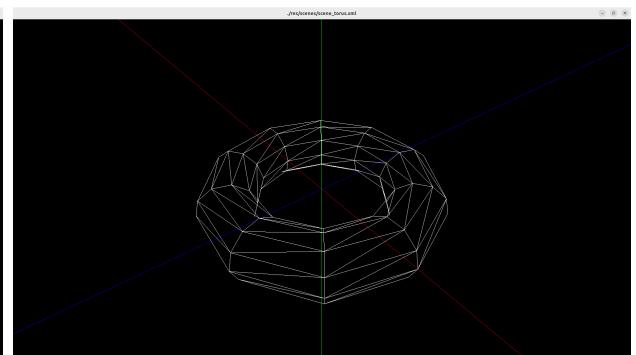


Figura 21: Torus (torus)

3.2 Cenas fornecidas pela docênciā da UC

A docênciā da UC forneceu, juntamente com o enunciado do trabalho, algumas cenas a serem testadas no trabalho. A `engine` renderizou-as como esperado:

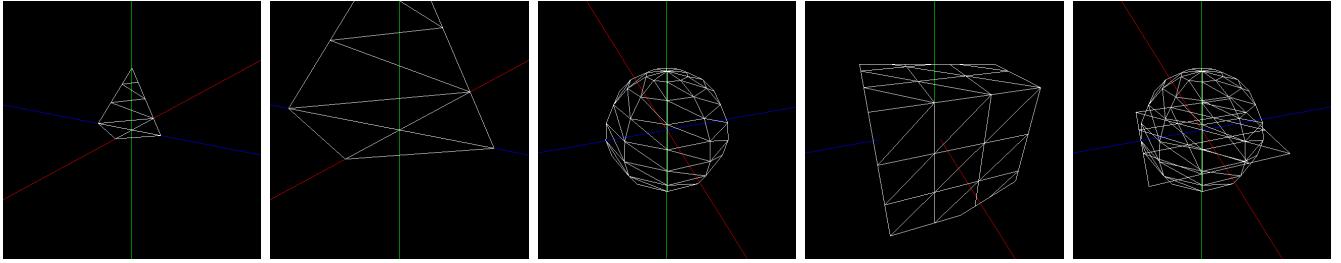


Figura 22: Renderização das cenas de teste fornecidas pela docência da UC.

Em particular, o último caso de teste demonstra a capacidade da `engine` de apresentar corretamente cenas com mais de um objeto.

3.3 Modelos 3D complexos

Para testar a capacidade da `engine` de carregar modelos mais complexos, foram obtidos os seguintes modelos 3D da Internet [13]:

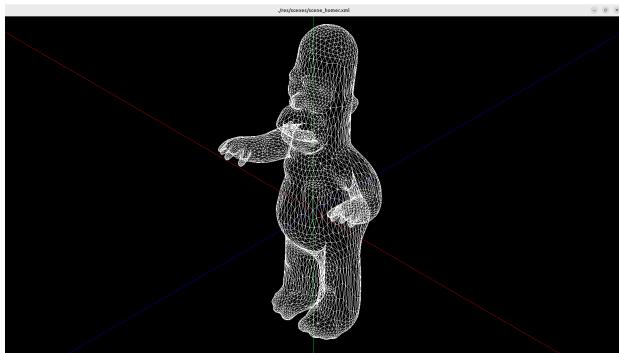


Figura 23: Homer

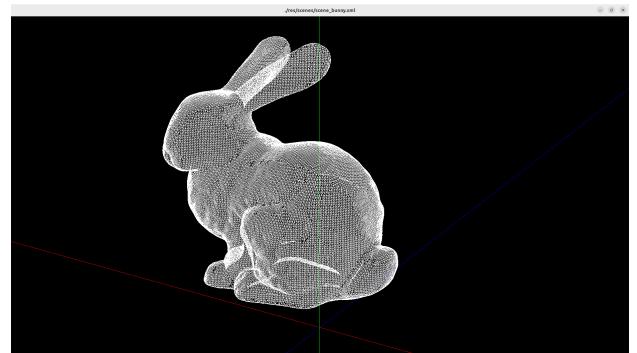


Figura 24: Coelho de Stanford

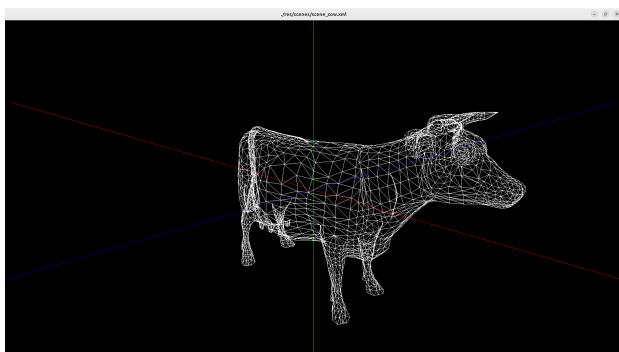


Figura 25: Vaca

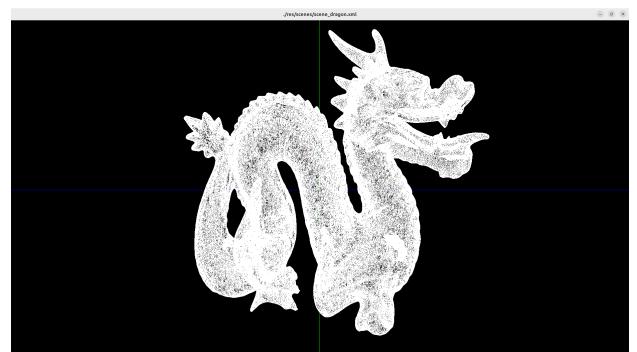


Figura 26: Dragão de Stanford

Em particular, a renderização do modelo do dragão demonstrou a capacidade da `engine` de lidar com modelos de alta complexidade, com um grande número de faces (871 mil).

4 Conclusão e Trabalho Futuro

A primeira fase do projeto foi concluída com sucesso, de modo que os objetivos previamente definidos foram alcançados. Ademais, também implementámos a geração de mais figuras geométricas (o cilindro e o *torus*), e a implementação de Vertex Buffer Objects (VBOs) já nesta fase melhorou o desempenho da `engine` e permitiu o uso de versões mais recentes de OpenGL, com outras funcionalidades que podem vir a ser usadas no futuro.

Em relação ao trabalho futuro, pretendemos realizar uma reestruturação da organização interna das cenas. A abordagem atual, que armazena os objetos numa estrutura linear, será substituída por um sistema de grupos, que permitirá a aplicação de transformações (translações, rotações, e escalas) a conjuntos de modelos de forma simultânea.

Adicionalmente, tencionamos expandir as capacidades da câmara, implementando três modos distintos de visualização: orbital, livre e em terceira pessoa. Esta diversidade de perspetivas enriquecerá a experiência do utilizador, e permitirá a exploração de diferentes estilos de interação com as cenas 3D.

5 Bibliografia

- [1] "The Industry's Foundation for High Performance Graphics." OpenGL. Accessed: Mar. 2, 2025. [Online.] Available: <https://www.opengl.org/>
- [2] "Wavefront OBJ File Format Summary." FileFormat.Info. Accessed: Mar. 2, 2025. [Online.] Available: <https://www.fileformat.info/format/wavefrontobj/egff.htm>
- [3] "OpenGL Sphere." Songho CA. Accessed: Mar. 1, 2025. [Online.] Available: https://www.songho.ca/opengl/gl_sphere.html
- [4] "OpenGL Cylinder, Prism & Pipe." Songho CA. Accessed: Mar. 1, 2025. [Online.] Available: https://www.songho.ca/opengl/gl_cylinder.html
- [5] "Torus." Wolfram MathWorld. Accessed: Mar. 1, 2025. [Online.] Available: <https://mathworld.wolfram.com/Torus.html>
- [6] "An OpenGL library." GLFW. Accessed: Feb. 27, 2025. [Online.] Available: <https://www.glfw.org>
- [7] "About." The freeglut project. Accessed: Feb. 27, 2025. [Online.] Available: <https://freeglut.sourceforge.net/>

- [8] "RenderDoc." RenderDoc Accessed: Feb. 28, 2025. [Online.] Available: <https://renderdoc.org/>
- [9] "Glad - Multi-Language GL/GLES/EGL/GLX/WGL Loader-Generator based on the official specs." Feb. 28, 2025. [Online.] Available: <https://glad.dav1d.de/>
- [10] "TinyXML-2." GitHub. Mar. 2, 2025. [Online.] Available: <https://github.com/leethomason/tinyxml2>
- [11] "gluLookAt." Khronos Registry. Accessed: Mar. 2, 2025. [Online.] Available: <https://registry.khronos.org/OpenGL-Refpages/gl2.1/xhtml/gluLookAt.xml>
- [12] "glm." GitHub. Accessed: Mar. 2, 2025. [Online.] Available: <https://github.com/g-truc/glm>
- [13] "common-3d-test-models" GitHub. Accessed: Mar. 2, 2025. [Online.] Available: <https://github.com/alecjacobson/common-3d-test-models>