

Universidade do Minho

Escola de Engenharia

Computação Gráfica

Trabalho Prático – Fase II

Grupo 3

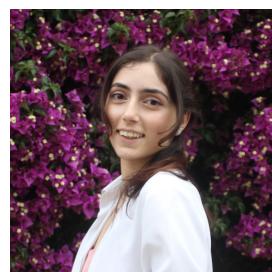
30 de março 2025



Ana Oliveira



Humberto Gomes



Mariana Cristino



Sara Lopes

A104437

A104348

A90817

A104179

Resumo

Para esta fase do trabalho prático, complementámos os programas `generator` e `engine` previamente desenvolvidos. Ao `generator`, adicionámos a possibilidade de gerar uma cena com o Sistema Solar, e modelos com rodas dentadas, fitas de Möbius, e garrafas de Klein. À `engine`, adicionámos transformações geométricas dos objetos, uma interface com o utilizador, movimentos da câmara (livre e orbital), e *frustum culling* baseado em esferas. Na próxima fase, pretendemos desenvolver sobre os resultados aqui obtidos para adicionar diversas outras funcionalidades, explorando curvas de Bézier e de Catmull-Rom para a geração e animação de modelos, respetivamente.

1 *Generator*

1.1 Geração do Sistema Solar Estático

O sistema solar foi representado por um uma estrutura hierárquica, definida de acordo com a atração gravítica real dos corpos celestes. No topo da hierarquia, encontra-se o Sol, que serve de ponto central e de referência para a órbita dos restantes corpos celestes. Cada planeta foi colocado num grupo independente, filho do grupo do Sol. Quando aplicável, os satélites naturais (luas) e os anéis dos planetas são inseridos como grupos filhos do planeta correspondente, preservando a hierarquia entre os corpos. As cinturas de asteroides, por sua vez, são representadas como grupos filhos do Sol, uma vez que orbitam em torno dele.

Cada grupo define as transformações aplicadas aos seus elementos, como translações, rotações, e escalas, que são acumuladas ao longo da árvore de grupos, ou seja, as transformações são aplicadas sequencialmente e as transformações de cada corpo são relativas ao seu elemento pai. Esta organização simplifica a expressão das translações e escalas relativas dos corpos celestes. Ademais, apesar de não haver movimento do sistema solar neste momento, esta estrutura hierárquica torna mais fácil que, no futuro:

- os planetas orbitem em torno do Sol;
- as luas orbitem em torno do planeta a que pertencem;
- os anéis acompanhem o seu planeta respetivo;
- os asteroides se posicionem nas suas cinturas em órbitas relativas ao Sol.

A estrutura XML da cena gerada segue a hierarquia descrita anteriormente. Abaixo, apresenta-se um exemplo de cena, apenas com o Sol, a Terra e a Lua:

```

<group>
    <!-- Grupo do sol -->
    <group>
        <transform>
            <translate x="0" y="0" z="0" />
            <scale x="30" y="30" z="30" />
        </transform>
        <models>
            <model file="../models/sphere.3d" />
        </models>
    <!-- Grupo da Terra -->
    <group>
        <transform>
            <translate x="28" y="0" z="-3" />
            <scale x="0.2" y="0.2" z="0.2" />
        </transform>
        <models>
            <model file="../models/sphere.3d" />
        </models>
    <!-- Grupo da Lua -->
    <group>
        <transform>
            <translate x="3" y="0" z="0" />
            <scale x="0.05" y="0.05" z="0.05" />
        </transform>
        <models>
            <model file="../models/sphere.3d" />
        </models>
    </group>
    </group>
</group>

```

Neste exemplo:

- O grupo no topo da hierarquia contém o Sol, posicionado na origem e aumentado com uma escala (`scale`).
- Dentro do grupo do Sol, encontra-se a Terra, num grupo posicionado com uma translação (`translate`). O tamanho da Terra também é ajustado com uma escala.
- Dentro do grupo da Terra está a Lua, com a sua própria transformação relativa.

Além dos planetas e das luas, existem dois corpos celestes adicionais com formas de representação específicas no XML:

- As cinturas de asteroides são compostas por esferas, cubos e cilindros, escolhidos aleatoriamente. Estas figuras são adicionadas um a grupo filho do Sol, e são posicionadas com coordenadas aleatórias em torno de um raio definido. Cada asteroide recebe transformações próprias de translação, rotação e escala, simulando a dispersão e dimensão variável destes corpos celestes.
- Os anéis dos planetas utilizam o modelo de um *torus*, gerado na primeira fase deste trabalho prático. Para simular a sua aparência achatada, é aplicada uma escala com fator inferior a 1 no eixo *y*. Além disso, os anéis são orientados com uma rotação por um diferente eixo consoante o seu planeta.

Parte do sistema solar gerado pode ser observado na figura abaixo:

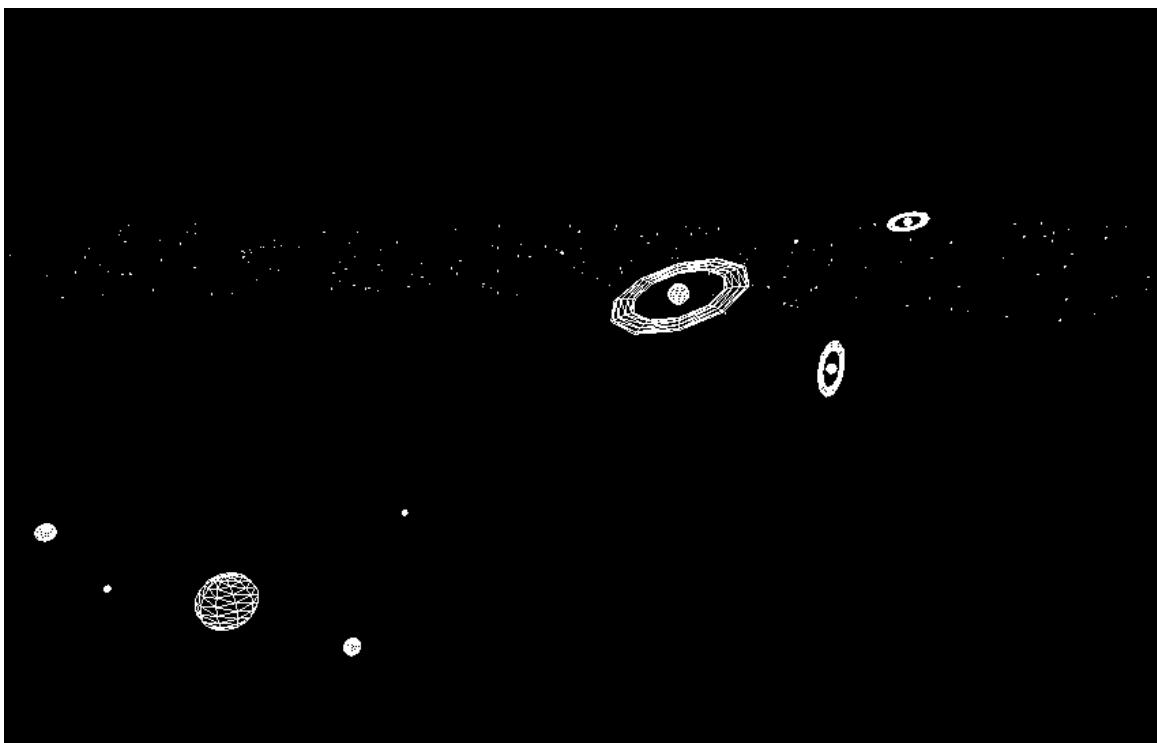


Figura 1: Sistema solar com anéis e cinturas de asteroides visíveis.

O sistema solar pode ser gerado de forma personalizada, através dos seguintes parâmetros definidos pelo utilizador, via linha de comando, especificados ao `generator` pela ordem apresentada abaixo:

- `sceneScale` – Escala global, aplicada a toda a cena;

- `sunSizeFactor` – Escala aplicada ao Sol;
- `planetSizeFactor` – Escala aplicada aos planetas;
- `moonSizeFactor` – Escala aplicada aos satélites naturais;
- `distanceFactor` – Proporção da distância entre os diferentes corpos celestes;
- `asteroidBeltDensity` – Densidade definida para as cinturas de asteroides;
- `ringSizeFactor` – Proporção dos anéis em relação ao planeta.

Ainda existem muitas melhorias que podem ser feitas ao sistema solar, como o movimento dos corpos e a adição de texturas, que se pretendem implementar ao longo das próximas fases do trabalho prático.

1.2 Fita de Möbius

Uma funcionalidade adicional que se implementou no `generator` foi a geração de fitas de Möbius. Para gerar uma fita de Möbius, é necessário ter-se um raio, uma largura, o número de *slices*, o número de *stacks*, e o ficheiro para onde será exportado o modelo gerado. Ademais, uma fita de Möbius clássica possui uma torção (uma meia volta), mas para tornar a geração da figura mais interessante, é possível variar de número de torções do modelo gerado. Abaixo, apresenta-se a linha de comandos do `generator` que deve ser usada para gerar uma fita de Möbius:

```
./generator mobiusStrip <radius> <width> <twists> <slices> <stacks> <file>
```

Matematicamente, as coordenadas de um ponto da *fita de Möbius* são definidas do seguinte modo, onde R , W e T são os parâmetros dados para a geração da *fita de Möbius*, o raio, a largura e o número de torções, respetivamente [1]:

$$\begin{aligned} x &= \left(R + W \cdot \frac{\theta}{2} \cdot \cos\left(T \cdot \frac{\phi}{2}\right) \right) \cdot \cos \phi & \phi \in [0, 2\pi[\\ y &= W \cdot \frac{\theta}{2} \cdot \sin\left(T \cdot \frac{\phi}{2}\right) & \theta \in [-1, 1] \\ z &= \left(R + W \cdot \frac{\theta}{2} \cdot \cos\left(T \cdot \frac{\phi}{2}\right) \right) \cdot \sin \phi \end{aligned}$$

Para gerar uma nuvem de pontos uniformemente distribuídos, discretiza-se ϕ e θ em *slices* e *stacks*, respetivamente, e percorre-se a faixa pelas suas fatias e camadas:

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

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

Após a geração dos vértices, estes são agrupados nos triângulos que compõem a superfície da fita de Möbius. Para isso, considera-se um vértice de referência, 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 , como mostra a figura abaixo. Com estes quatro vértices, geram-se quatro faces triangulares: duas faces voltadas para um lado e outras duas faces para o outro. Este processo repete-se para todos os vértices onde é aplicável, garantindo que cada célula quadrangular é subdividida em quatro triângulos, dois para cada lado.

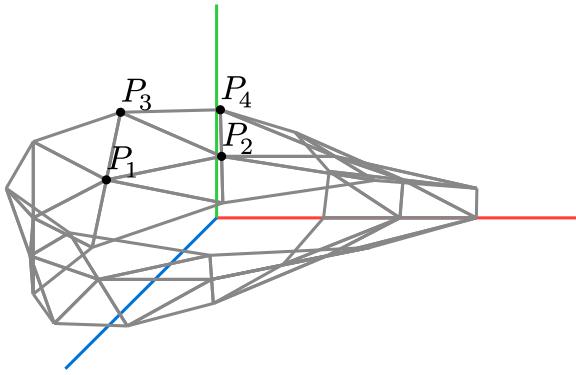


Figura 2: Pontos utilizados numa iteração da geração de faces da fita de Möbius.

Como a *fita de Möbius* é uma superfície não orientável, deve ser possível ver os dois lados da fita quando esta é desenhada. É por isto que se geram faces dos dois lados da fita: estes são necessários para esta ser visualizada mesmo quando o *back-face culling* está ativo. Para gerar faces com diferentes orientações, a ordem em que os pontos são especificados é alterada conforme a direção desejada. Por exemplo, para os quatro pontos apresentados acima, os triângulos gerados são os seguintes:

$$T_1 = (P_1, P_2, P_3) \quad T_2 = (P_2, P_1, P_3) \quad T_3 = (P_3, P_4, P_2) \quad T_4 = (P_4, P_3, P_2)$$

1.3 Garrafa de Klein

Também foi implementada a geração outras superfícies não orientáveis, como a garrafa de Klein, que não possui um lado interno e externo distinguíveis. A parametrização de um vértice sobre

a garrafa de Klein de raio r pode ser definida pelas seguintes equações [2]:

$$x = -\frac{2}{15} r \cos(\theta) (3 \cos \phi - 30 \sin \theta + 90 \cos^4 \theta \sin \theta - 60 \cos^6 \theta \sin \theta + 5 \cos \theta \cos \phi \sin \theta)$$

$$y = -\frac{1}{15} r \sin(\theta) (3 \cos \phi - 3 \cos^2 \theta \cos \phi - 48 \cos^4 \theta \cos \phi + 48 \cos^6 \theta \cos \phi - 60 \sin \theta + 5 \cos \theta \cos \phi \sin \theta - 5 \cos^3 \theta \cos \phi \sin \theta - 80 \cos^5 \theta \cos \phi \sin \theta + 80 \cos^7 \theta \cos \phi \sin \theta)$$

$$z = \frac{2}{15} r (3 + 5 \cos \theta \sin \theta) \sin \phi$$

A variável θ varia entre 0 e π , e ϕ varia entre 0 e 2π . Para a geração dos pontos da garrafa de Klein, os valores destas duas variáveis foram discretizados:

$$\begin{aligned} \phi_i &= i \cdot \frac{2\pi}{N_{\text{slices}}} & i \in \{0, 1, \dots, N_{\text{slices}}\} \\ \theta_j &= j \cdot \frac{\pi}{N_{\text{stacks}}} & j \in \{0, 1, \dots, N_{\text{stacks}}\} \end{aligned}$$

Após a geração dos vértices, constroem-se as faces da garrafa de Klein. Cada quadrilátero entre duas *stacks* sucessivas é dividido em duas faces triangulares. Ao contrário da fita de Möbius, decidiu-se que não era necessário renderizar o interior da garrafa, motivo pelo qual não se geram quatro faces triangulares para cada quadrilátero.

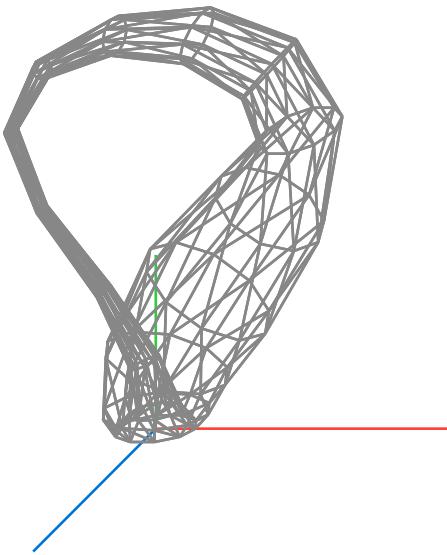


Figura 3: Garrafa de Klein

Para usar o generator para gerar uma garrafa de Klein, a seguinte linha de comando deve ser utilizada:

```
./generator kleinBottle <radius> <slices> <stacks> <file>
```

1.4 Roda Dentada

Também foi implementada a geração de rodas dentadas, que apresentam uma natureza muito semelhante à de um cilindro: o seu interior é cilíndrico, e o seu exterior também o é, no entanto com variações do raio nos dentes da roda dentada. A construção desta primitiva tem por base coordenadas cilíndricas, onde a posição de cada ponto é definida em função do raio interno (r_i), do raio externo (r_e), da altura do dente (h_t), do ângulo azimutal (ϕ) e da altura da roda dentada (h).

A parametrização de um ponto sobre uma roda dentada pode ser definida pelas seguintes equações:

$$x = r \cos \phi$$

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

$$z = r \sin \phi$$

No entanto, ao contrário do que acontece num cilindro, r possui três valores distintos: para a parte interna da roda, é utilizado r_i , para a parte externa, r_e , e nos dentes da roda, $r_e + h_t$.

Para discretizar a superfície da roda dentada, divide-se o intervalo de ϕ em *slices*, e o intervalo de y em *stacks*. Assim, os incrementos destas variáveis são calculados da seguinte forma:

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

É de notar que o número de fatias deve ser um múltiplo de 4 do número de dentes da roda dentada. Caso contrário, irregularidades no modelo gerado podem ocorrer. Para evitar este problema, o número de *slices* é obrigatoriamente o $4N_{\text{dentes}}$, que o utilizador especifica ao **generator** juntamente com os outros parâmetros através da seguinte linha de comando:

```
./generator gear <majorRadius> <minorRadius> <height> <stacks> <teeth> <toothHeight> <file>
```

A construção dos vértices segue os seguintes passos:

- Superfície lateral externa: Para cada *stack*, os valores de ϕ são iterados, determinando a posição dos pontos ao longo da circunferência externa. Nos intervalos correspondentes aos dentes, o raio é aumentado em h_t , a altura do dente, passando a ser $r_e + h_t$.
- Superfície lateral interna: De forma similar à superfície lateral externa, é gerada a superfície interna do cilindro, mas o raio mantém-se constante em r_i .

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

- Superfície lateral externa: Os pontos adjacentes de duas *stacks* consecutivas são ligados, formando quadriláteros que são divididos individualmente em dois triângulos.
- Superfície lateral interna: É aplicada a mesma estratégia ao cilindro interno, com a ligação dos pontos entre diferentes *stacks*.
- Bases superior e inferior: Os pontos da última *stack* são ligados aos vértices centrais correspondentes, de modo a formar triângulos radiais.

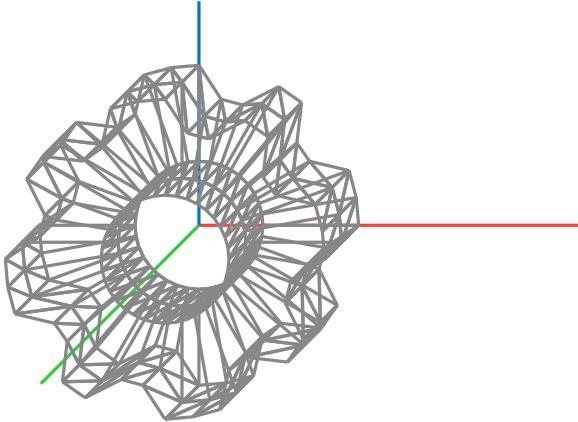


Figura 4: Representação da estrutura da roda dentada. Note que a orientação dos eixos foi trocada para ser possível melhor observar os detalhes desta figura.

2 *Engine*

2.1 Transformações

Um dos objetivos desta fase do trabalho prático é a possibilidade de aplicar transformações de translação, rotação e escala às entidades na cena. Estas podem ser especificadas no ficheiro XML como se apresenta no exemplo abaixo:

```
<group>
    <transform>
        <translate          x="1"  y="2"  z="3" />
        <rotate    angle="90" x="0"  y="0"  z="1" />
        <scale            x="1"  y="2"  z="1" />
    </transform>
    <models>
        <model file="modelo.3d" />
    </models>
</group>
```

Neste exemplo, a todos os modelos em `<models>` devem ser aplicadas, por esta ordem, uma translação pelo vetor $(1, 2, 3)$, uma rotação de 90° pelo eixo z , e uma escala vertical por um fator de duas vezes. Para calcular as matrizes associadas a cada uma destas transformações, as funções `glm::translate`, `glm::rotate`, `glm::scale` são utilizadas. [3] Estas são semelhantes às funções `glTranslate`, `glRotate` e `glScale` [4], respetivamente, mas devolvem a matriz de transformação calculada, em vez de modificarem as matrizes do estado interno do OpenGL. Outra diferença é que o ângulo dado à função `glm::rotate` deve ser passado em radianos,

sendo necessária uma conversão do valor em graus presente na descrição XML da cena.

Para compor estas operações, as matrizes geradas pela `glm` são multiplicadas pela ordem em que as transformações surgem no ficheiro XML. No exemplo anterior, onde T é a matriz da translação, R a matriz de rotação, e S a matriz de escala, as coordenadas no mundo dos pontos do modelo devem ser multiplicadas pela matriz W abaixo, originando as coordenadas no mundo do modelo transformado:

$$W = TRS$$

Para desenhar as entidades no ecrã, é necessário também ter em conta a matriz da câmara, C , o produto entre a matriz de projeção e a matriz de vista. Ademais, como os grupos na cena formam uma hierarquia, é necessário considerar as transformações dos ascendentes de um grupo para o desenhar. Assim, para desenhar a cena, usa-se uma *stack* de matrizes, inicializada com a matriz da câmara. Depois, para cada grupo aninhado, calcula-se o produto entre a matriz no topo do *stack* e a matriz de transformação do grupo, e esta matriz calculada é adicionada ao topo da *stack*. Quando se termina o desenho do grupo, é removida a matriz do topo da *stack*. No exemplo abaixo, mostra-se a *stack* de matrizes durante o desenho de um grupo na hierarquia da cena:

$C \cdot W_1 \cdot W_3 \cdot W_4$
$C \cdot W_1 \cdot W_3$
$C \cdot W_1$
C

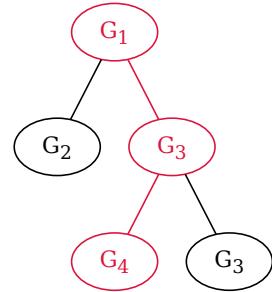


Figura 5: *Stack* de matrizes associada ao desenho de um grupo numa cena hierárquica.

2.2 UI (Interface Gráfica)

Como funcionalidade adicional, foi implementada uma interface gráfica, com o auxílio da biblioteca `Dear ImGui`. [5] Esta biblioteca simplifica a gestão e atualização dos componentes, pois os elementos gráficos são recriados a cada *frame*.

A classe responsável pela configuração e renderização da interface é a `UI`. No processo de

inicialização realizado no seu construtor, a criação do contexto `ImGui` é realizada, bem como a configuração do estilo da interface (o estilo *Dark* foi escolhido por preferência dos autores deste trabalho) e é efetuada a inicialização das implementações `OpenGL3` e `GLFW` da biblioteca, algo que é necessário, visto que a biblioteca suporta diversos outros *backends*.

A interface é composta por um menu, sendo este renderizado na função `UI::render` com os seguintes elementos:

- Acompanhamento do Desempenho:
 - FPS (*Frames Per Second*): apresenta o número de *frames* renderizados por segundo.
 - Apresenta o número de entidades renderizadas em relação ao número total de entidades existentes. Esta métrica é útil para testar a eficiência do *frustum culling*, pois permite avaliar se apenas as entidades dentro da visão da câmara estão a ser renderizadas.
- Configurações:
 - Preenchimento de polígonos: permite alternar entre o preenchimento (*fill polygons*) e o modo *wireframe*.
 - *Back-face culling*: ativa ou desativa o *back-face culling*.
 - Exibir eixos: ativa ou desativa a visualização dos eixos.
 - *Bounding spheres*: ativa ou desativa a visualização das esferas encapsuladoras das entidades (ver *frustum culling*).
- Câmara: permite modificar as coordenadas da posição da câmara.

Após a definição dos elementos gráficos, a interface é renderizada através das funções `ImGui::Render` e `ImGui_ImplOpenGL3_RenderDrawData`.



Figura 6: Interface Gráfica.

2.3 Câmara Orbital

A câmara orbital é uma câmara virtual que permite observar uma cena a partir de um ponto que orbita à volta de ou de um objeto ou de um ponto fixo (`lookAt`). Na implementação atual, este ponto de interesse está fixado na origem do espaço tridimensional. Esta abordagem é útil para observar um objeto de diversos ângulos possíveis com o foco num ponto em específico.

A câmara orbital é definida por 3 parâmetros:

- Raio (r): determina a distância do ponto de interesse à câmara;
- Ângulo azimutal (ϕ): define a rotação da câmara em torno do eixo vertical, o que permite uma visualização de 360º em redor do objeto;
- Ângulo polar (θ): controla o ângulo de elevação da câmara.

A atualização da posição da câmara é dada pela soma do ponto de interesse (`lookAt`) com as coordenadas cartesianas obtidas a partir das coordenadas esféricas. As fórmulas utilizadas para a conversão são as seguintes, tendo como ponto de interesse a origem do referencial:

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

$$y = r \cos \theta$$

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

De forma a garantir um comportamento estável e consistente, alguns valores devem ser restringidos a intervalos pré-definidos. O raio encontra-se limitado entre 0.5 e 100, para evitar que a câmara se aproxime demasiado ou se afaste demasiado do ponto de interesse. O ângulo azimutal varia entre 0 e 2π , sendo reajustado quando a câmara completa uma volta. Impedir que este valor fique demasiado alto ou demasiado baixo faz com que não sejam causados erros de precisão de vírgula flutuante caso se deem milhões de voltas à câmara. Por fim, o ângulo polar encontra-se entre 0.01 e $\pi - 0.01$ para evitar que a câmara alcance os polos e cause instabilidade no movimento. Sem esta restrição, caso $\theta > \pi$, o vetor \vec{up} da câmara deixaria de ser válido e a movimentação vertical da câmara ficaria invertida, tornando a navegação confusa e pouco intuitiva.

Para controlar a câmara orbital, o utilizador utiliza o teclado, sendo que as teclas associadas a cada ação são:

- W/S: inclinar a câmara para cima e para baixo, respetivamente;
- A/D: rodar a câmara para a esquerda e para a direita, respetivamente;
- F: aproximar a câmara do ponto de interesse;
- B: afastar a câmara do ponto de interesse.

2.4 Câmara Livre

A câmara livre permite que o utilizador se desloque de forma intuitiva, sendo semelhante à de jogos de exploração na primeira pessoa, como simuladores de voo, e ambientes de realidade virtual.

A câmara pode deslocar-se e rodar de forma contínua no espaço tridimensional, sendo os seus movimentos definidos pelos seguintes vetores:

- Vetor \vec{d} : Direção do olhar da câmara.
- Vetor \vec{r} : Direção perpendicular ao vetor \vec{d} , ao longo do plano horizontal.
- Vetor \vec{up} : Aproximação do eixo vertical da câmara.

A posição da câmara, P , é atualizada de acordo com a seguinte equação:

$$P' = P + v \cdot \vec{d}',$$

onde v representa a velocidade da câmara e \vec{d}' a direção do movimento, que depende do movimento escolhido pelo utilizador:

- Translação horizontal: O movimento para a frente e para trás ocorre ao longo do vetor \vec{d}' ($\vec{d}' = \pm \hat{d}$), enquanto o movimento lateral ocorre ao longo do vetor \vec{r} ($\vec{d}' = \pm \hat{r}$).
- Translação vertical: O movimento vertical acontece ao longo do vetor \vec{up} ($\vec{d}' = \pm \hat{up}$), permitindo ao utilizador subir e descer a câmara.

Para as transformações de rotação, é importante saber que a orientação da câmara é definida pelos ângulos θ (*yaw*), responsável pela rotação horizontal, e ϕ (*pitch*), responsável pela

inclinação vertical. Ambos estes ângulos determinam a direção do vetor \vec{d} . A atualização da orientação da câmara obtém-se através da conversão dos ângulos θ e ϕ para um vetor de direção tridimensional. A relação entre os ângulos e os componentes do vetor \vec{d} é dada por:

$$d_x = \cos \phi \cos \theta$$

$$d_y = \sin \phi$$

$$d_z = \cos \phi \sin \theta$$

Após serem calculados os componentes do vetor, este é normalizado, uma vez que é necessário garantir que este mantém um comprimento unitário. Esta normalização é fundamental para garantir que a câmara mantém uma velocidade constante em qualquer direção.

Além disso, foram tomadas medidas para prevenir o *gimbal lock*. Este é um problema que ocorre quando dois eixos de rotação se alinham, ou seja, quando ϕ atinge 90° ou -90° . Além disso, no caso de $|\phi|$ exceder 90° , tal como na câmara livre, o vetor \vec{up} deixa de ser válido e a movimentação vertical da câmara fica invertida. Para resolver estes dois problemas, foi imposto o seguinte limite:

$$-89^\circ \leq \phi \leq 89^\circ$$

Este impede que a câmara olhe completamente para cima ou para baixo, garantindo que o vetor \vec{up} mantenha uma orientação bem definida e evitando a perda de controlo sobre a rotação horizontal.

Sempre que a câmara sofre alterações, é necessário recalcular os seus vetores \vec{r} e \vec{up} . O vetor \vec{r} é obtido como o produto externo entre a direção do olhar da câmara, \vec{d} , e o vetor de referência vertical, $(0, 1, 0)$:

$$\vec{r} = \vec{d} \times \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

Depois, o vetor \vec{up} é recalculado para ser ortogonal aos outros dois vetores:

$$\vec{up} = \vec{r} \times \vec{d}$$

Isto assegura que os movimentos laterais, verticais e de rotação permanecem coerentes em qualquer orientação da câmara, garantindo um comportamento previsível para o utilizador. Ademais, como a função `glm::lookAt` leva como argumento um ponto para o qual a câmara aponta, e não o vetor \vec{d} , é necessário calcular as coordenadas de um ponto possível para gerar a nova matriz de visualização:

$$L = P + \vec{d}$$

O utilizador pode controlar a câmara de forma intuitiva, utilizando o teclado para movimentação e as setas direcionais para rotação:

- W/S: Mover a câmara para a frente / trás, respetivamente;
- A/D: Mover a câmara para os lados esquerdo e direito, respetivamente;
- Espaço: Sobir a câmara;
- *Shift* Esquerdo: Descer a câmara;
- Setas Esquerda / Direita: Rodar a câmara para a esquerda/direita;
- Setas Cima / Baixo: Inclinar a câmara para cima/baixo.

2.5 *Frustum Culling*

Cada *draw call* tem um custo elevado para o desempenho da aplicação. Logo, procurando reduzir o número de *draw calls*, foi implementado *frustum culling*, para apenas ser requisitada à GPU a renderização das entidades totalmente ou parcialmente no *view frustum* da câmara.

Seria muito computacionalmente intensivo verificar se a geometria de um modelo se encontra dentro ou fora do *view frustum*, pelo que se encapsulam todos os modelos em esferas que, devido à sua geometria simples, permitem uma verificação rápida da sua visibilidade no *view frustum*. No entanto, visto que estas esferas podem ser um pouco maiores do que os modelos em si, é possível que algumas entidades fora do ecrã sejam desenhadas, visto que parte das suas esferas ainda podem intersestar os planos do *view frustum*. Para mitigar este problema, formas geométricas encapsuladoras mais complexas poderiam ser utilizadas, visto que estas se poderiam adaptar melhor à geometria dos modelos. No entanto, o uso destas formas complexas conduziria a testes de visibilidade mais caros, possivelmente anulando os benefícios de desenhar um menor número de entidades.

Quando um modelo é carregado, é necessário calcular a esfera que o encapsula. Em primeiro lugar, o seu centro é calculado como o centro de massa de todos os pontos, como mostra a expressão abaixo, onde M é o modelo, uma sequência de pontos tridimensionais:

$$C = \frac{1}{|M|} \sum_{p \in M} p$$

Depois, o raio da esfera pode ser determinado como a distância entre o centro da esfera e o ponto mais longínquo do mesmo, como mostra a expressão abaixo, onde d é a função de distância cartesiana entre dois pontos:

$$r = \max \{d(C, p) \mid p \in M\}$$

É também necessário saber como uma esfera encapsuladora é afetada quando o objeto que encapsula sofre uma transformação no mundo. Considere-se uma matriz de transformação aplicada ao objeto (em coordenadas do mundo), originada através da aplicação de translações, rotações, e escalas. A matriz será 4x4 e terá o seguinte aspecto:

$$T = \begin{bmatrix} \vec{i} & \vec{j} & \vec{k} & \vec{t} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Para calcular o centro da esfera após a transformação da entidade, basta multiplicar a matriz de transformação pela posição do centro da esfera:

$$C' = TC$$

Depois, para calcular o novo raio da esfera, não é necessário ter em conta as transformações de rotação, visto que as esferas são simétricas em todos os eixos possíveis. No entanto, é necessário ter em conta a escala aplicada ao modelo. Por exemplo, na matriz T , a escala da entidade pelo eixo x é $\|\vec{i}\|$, e o mesmo se tem para o eixo y e $\|\vec{j}\|$, e para z e $\|\vec{k}\|$. Logo, o raio da esfera transformada é:

$$r' = \max (\|\vec{i}\|, \|\vec{j}\|, \|\vec{k}\|) \cdot r$$

É possível tirar proveito da estrutura hierárquica da cena para otimizar o processo de *frustum culling*. Por exemplo, caso um grupo contenha várias entidades ou subgrupos, pode construir-se uma esfera que encapsula a totalidade do grupo. Caso essa esfera não esteja no *view frustum*, pode-se evitar fazer os testes de visibilidade para as esferas dos objetos individuais que compõem o grupo. Caso contrário, é na mesma necessário realizar esses testes.

O processo para determinar as características de uma esfera que encapsula todos os objetos de um grupo é semelhante ao da construção de esferas com base no conjunto de pontos de um modelo. Em primeiro lugar, para determinar o centro da esfera, calcula-se o centro de massa do conjunto de pontos formado pelos centros de todas as esferas, C . De seguida, para cada subesfera, calcula-se a sua distância máxima a C , o raio da subesfera adicionado à distância entre C e o centro da subesfera. Depois, a maior destas distâncias é escolhida para ser o raio da nova esfera, como mostra a expressão abaixo, onde S representa o conjunto de subesferas:

$$r' = \max \{d(C', C) + r \mid (C, r) \in S\}$$

Foi adicionada à `engine`, para depuração, a possibilidade de visualizar as esferas encapsuladoras dos grupos e das entidades na cena. Abaixo, apresenta-se uma captura de ecrã da `engine` com esta funcionalidade ativa:

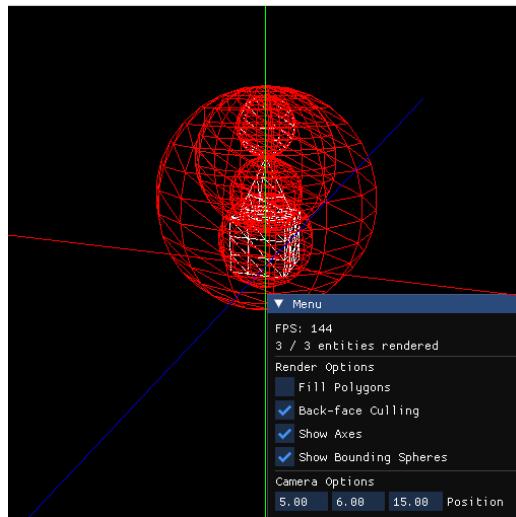


Figura 7: Esferas encapsuladoras de grupos e entidades.

Depois de saber como determinar as esferas encapsuladoras das entidades, é necessário determinar o *view frustum* da câmara, para se poder verificar a posição das esferas em relação aos planos do *view frustum*. Para determinar o *view frustum*, é necessário obter alguns vetores importantes da câmara [6], os mesmos necessários para o funcionamento da câmara livre, \vec{d} , \vec{r} e \vec{u} . É importante também notar que o vetor \vec{u} utilizado dado na descrição XML da cena pode não ser adequado para o cálculo do *frustum*: é necessário garantir que $\vec{u} \perp \vec{d}$. Para tal, o vetor

\vec{up} pode sofrer a seguinte transformação:

$$\vec{up} \leftarrow (\vec{d} \times \vec{up}) \times \vec{d}$$

Além destes vetores, é também necessário conhecer as dimensões dos planos *near* e *far*. Apesar de, matematicamente, planos não terem uma altura e uma largura, utiliza-se esta linguagem para se referir às dimensões dos retângulos nestes planos que constituem o *view frustum*. Na expressão abaixo, mostra-se como se pode calcular a altura (H_{near}) e a largura (W_{near}) do plano *near*. O processo para o plano *far* é semelhante. Com o FOV da câmera (θ), a distância ao plano *near* (d_{near}) e o *aspect ratio* (A), podem calcular-se as dimensões deste plano [7]:

$$H_{\text{near}} = 2d_{\text{near}} \tan\left(\frac{\theta}{2}\right) \quad W_{\text{near}} = H_{\text{near}} A$$

Com estes valores, é possível determinar as coordenadas dos pontos dos retângulos do *view frustum*. Os quatro pontos do plano *near* (a amarelo na figura abaixo) podem ser calculados do seguinte modo, e o método utilizado para o plano *far* é semelhante [6]:

$$F = P + d_{\text{near}} \hat{d} \pm \frac{H_{\text{near}}}{2} \hat{up} \pm \frac{W_{\text{near}}}{2} \hat{r}$$

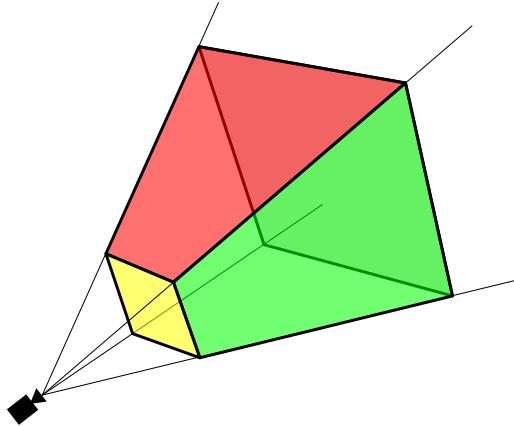


Figura 8: *View Frustum*.

Com os oito pontos do *view frustum*, é possível determinar as equações cartesianas de cada um dos seus planos, considerando três pontos da sua superfície. Em primeiro lugar, determina-se o vetor normal ao plano. Com os três pontos, P_1 , P_2 e P_3 , determinam-se dois vetores, a partir dos quais se calcula um produto externo, determinando-se assim um vetor perpendicular ao plano [8]:

$$n = \overrightarrow{P_1P_2} \times \overrightarrow{P_1P_3}$$

Depois, com este vetor normalizado, é possível determinar a constante na equação cartesiana do plano com base nas coordenadas de um dos três pontos dados [8]:

$$\begin{aligned} \alpha x + \beta y + \gamma z + \delta &= 0 \\ \Leftrightarrow \hat{n} \cdot P + \delta &= 0 \\ \Leftrightarrow \delta &= -\hat{n} \cdot P \end{aligned}$$

Durante a renderização de cada *frame*, verificam-se que esferas estão totalmente ou parcialmente dentro do *view frustum*. Para uma esfera estar no *view frustum* F , é necessário que a distância assinada entre o centro da esfera e cada um dos planos do *view frustum* não seja inferior ao simétrico do seu raio, ou seja [9]:

$$\forall_{p \in F} (\hat{n} \cdot C + \delta \geq -r)$$

Uma vez que é necessário calcular distâncias assinadas, é imperativo que os vetores normais de todos os planos do *view frustum* apontem para o seu interior, para que o conteúdo no seu interior (e não no seu exterior) seja desenhado. [6] Por este motivo, a ordem em que os pontos P_1 , P_2 e P_3 são especificados é relevante.

3 Resultados Obtidos

Nesta secção, procuram-se apresentar algumas capturas de ecrã da `engine` das novas cenas criadas e das cenas fornecidas pela docência da UC.

3.1 Figuras Geométricas Geradas

Os seguintes modelos foram criados utilizando o `generator`:

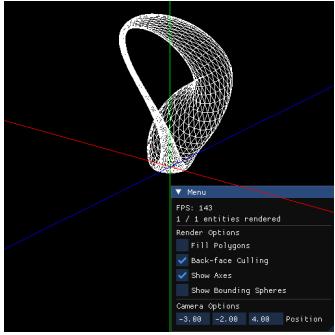


Figura 9: Garrafa de Klein.

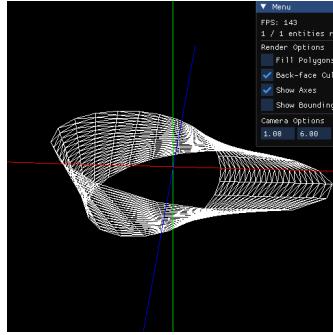


Figura 10: Fita de Möbius.

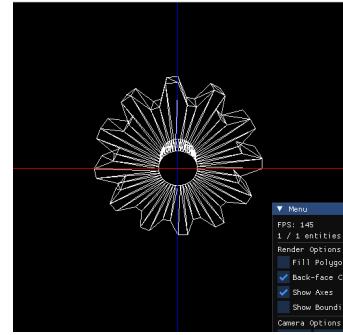


Figura 11: Roda dentada.

3.2 Cenas fornecidas pela docência da UC

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

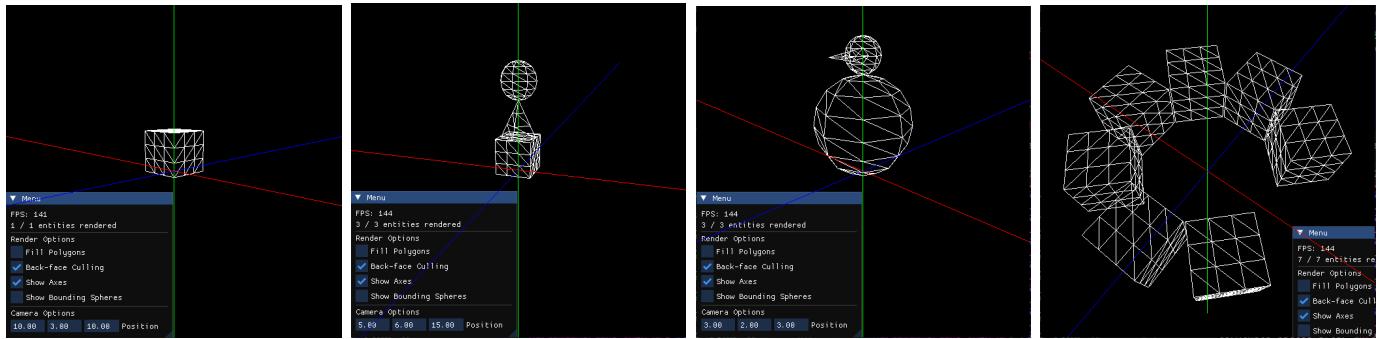


Figura 12: Renderização das cenas de teste fornecidas pela docência da UC.

4 Conclusão e Trabalho Futuro

Consideramos que esta segunda fase deste trabalho prático foi concluída com sucesso. Em primeiro lugar, não tivemos qualquer problema com a implementação das funcionalidades obrigatórias: as transformações dos objetos nas cenas e a geração do Sistema Solar. Em relação aos objetivos que tínhamos traçado na conclusão da última fase, conseguimos representar cenas numa estrutura hierárquica, e implementar dois novos tipos de câmara, a câmara orbital e a câmara livre. No entanto, não nos focámos na implementação da câmara em terceira pessoa, preferindo investir o nosso tempo em outras funcionalidades que nos despertaram mais curiosidade. Todavia, ainda pretendemos implementar este tipo de câmara, mas isso será algo para as próximas fases deste trabalho.

Além do que tínhamos planeado, implementámos diversas outras funcionalidades. Em primeiro

lugar, do lado do `generator`, implementámos diversas novas figuras, a roda dentada e alguns objetos conhecidos da topologia, a fita de Möbius e a garrafa de Klein. Ademais, do lado da `engine`, implementámos uma interface gráfica, para ser possível controlar vários parâmetros possíveis da `engine`. Por último, implementámos também *frustum culling*, para melhorar o desempenho da `engine` através da redução do número de *draw calls*.

Ao longo desta fase, a nossa principal dificuldade foi a implementação do *frustum culling*. Qualquer pequena falha (um sinal errado, uma troca entre duas variáveis, *etc.*) pode fazer com que este não funcione, pelo que foi necessário o desenvolvimento de uma boa infraestrutura para depuração (como a apresentação das *bounding spheres*) para encontrar e corrigir os erros que fomos cometendo durante a implementação desta funcionalidade.

Para a próxima fase deste trabalho, desejamos implementar as funcionalidades obrigatórias pedidas, a geração de modelos com base em *patches* de Bézier, e a animação dos objetos da cena com translações e rotações que variam ao longo do tempo. Além disso, como funcionalidades adicionais, desejamos implementar a já mencionada câmara em terceira pessoa, bem como *object picking* e *instanced rendering*.

5 Bibliografia

- [1] “Möbius Strip.” Wolfram MathWorld. Accessed: Mar. 25, 2025. [Online.] Available: <https://mathworld.wolfram.com/MoebiusStrip.html>
- [2] A. Norman and G. Newcomb, “Klein surfaces and real algebraic function fields.”, *Bulletin of the American Mathematical Society*, vol. 75, no. 4, pp. 869-872, Apr. 1969, doi:10.1090/S0002-9904-1969-12332-3.
- [3] “GLM_EXT_matrix_transform”. GLM 0.9.9 API documentation. Accessed: Mar. 27, 2025. [Online.] Available: <https://glm.g-truc.net/0.9.9/api/a00247.html>
- [4] “OpenGL (R) 2.1, GLX, and GLU Reference Pages ”. Khronos Registry. Accessed: Mar. 27, 2025. [Online.] Available: <https://registry.khronos.org/OpenGL-Refpages/g12.1>
- [5] “Dear ImGui.” GitHub. Accessed: Mar. 2, 2025. [Online.] Available: <https://github.com/ocornut/imgui>
- [6] “Geometric Approach – Extracting the Planes.”. Lighthouse3d.com. Accessed: Mar. 29, 2025. [Online.] Available: <https://www.lighthouse3d.com/tutorials/view-frustum-culling/geometric-approach-extracting-the-planes/>

- [7] “View Frustum’s Shape.”. Lighthouse3d.com. Accessed: Mar. 29, 2025. [Online.] Available: <https://www.lighthouse3d.com/tutorials/view-frustum-culling/view-frustums-shape/>
- [8] “Plane.”. Lighthouse3d.com. Accessed: Mar. 29, 2025. [Online.] Available: <https://www.lighthouse3d.com/tutorials/math/planar/>
- [9] “Geometric Approach – Testing Points and Spheres.”. Lighthouse3d.com. Accessed: Mar. 29, 2025. [Online.] Available: <https://www.lighthouse3d.com/tutorials/view-frustum-culling/geometric-approach-testing-points-and-spheres/>