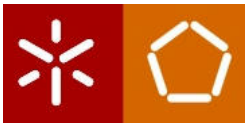

Universidade do Minho
Departamento de Informática

# Computação Gráfica

# Trabalho Prático - Fase 1

## Grupo 5

João Coelho - A100596

José Rodrigues - A100692

Duarte Araújo - A100750

2024-03-08

# 1. Introdução

No âmbito da Computação Gráfica, este projeto visa desenvolver um mini motor 3D baseado num grafo de cena. Como parte do curso de Computação Gráfica, o objetivo é aplicar os conceitos e técnicas aprendidos para criar um sistema capaz de renderizar modelos 3D simples em tempo real.

Na sua essência, o projeto gira em torno de dois componentes principais: o gerador e o motor.

- **Generator:** Este programa é responsável por gerar os vértices de modelos 3D baseados em diferentes primitivas gráficas, como planos, cubos, esferas e cones. Ele recebe parâmetros especificando o tipo de primitiva, dimensões e outras propriedades, e gera arquivos contendo as informações dos vértices num formato específico. Estes ficheiros gerados serão usados pela `engine` para renderização.
- **Engine:** É responsável por ler um ficheiro de configuração e renderizar a cena 3D em conformidade. Desempenha um papel central no processo de visualização e requer várias funcionalidades-chave, como a análise de XML, o carregamento de modelos, a configuração da câmara e a gestão de janelas.

# 2. Generator

Como mencionado anteriormente, o componente gerador desempenha um papel fundamental no cálculo dos vértices das primitivas selecionadas e na subsequente geração dos ficheiros .3d correspondentes. O gerador analisa os argumentos de entrada para discernir o tipo de primitiva e outros dados essenciais necessários para o cálculo dos vértices.

Só depois de recolher esta informação é que instancia um objeto que representa a primitiva especificada, passando à fase de cálculo. A estratégia utilizada durante esta fase varia consoante a primitiva em avaliação. No entanto, o processo global pode ser delineado em duas fases primárias: cálculo de vértices e criação de ficheiros .3d.

O processo de criação de ficheiros .3d permanece consistente em todas as primitivas, envolvendo a simples operação de escrever os vértices calculados no ficheiro. Por outro lado, a fase de cálculo dos vértices difere significativamente entre as primitivas. Como tal, a secção subsequente irá aprofundar os processos únicos envolvidos no cálculo de vértices para cada primitiva. Esta abordagem tem como objetivo fornecer uma explicação clara e detalhada do processo.

## 2.1. Plano

A primitiva plano representa a forma geométrica mais simples do nosso sistema. Para entender como são gerados os seus vértices, é importante primeiro compreender o conceito de divisões.

### 2.1.1. Disivões do Plano

As divisões de um plano representam a subdivisão da sua superfície em pequenos segmentos. Cada divisão cria uma seção retangular que compõe a grelha sobre o plano. Quanto maior o número de divisões, mais detalhada será a representação do plano.

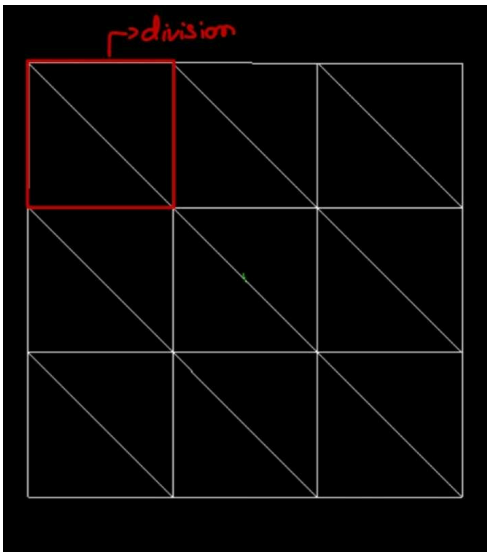

Figure 1: Divisões do Plano

### 2.1.2. Cálculo das Posições dos Vértices

Com as divisões em mente, podemos explorar como os vértices são calculados para formar essa grelha.

Ao calcular as posições dos vértices, começamos por determinar o meio comprimento e o comprimento da divisão do plano. O meio comprimento é a distância do centro do plano até a sua borda ao longo de um eixo, garantindo simetria e equilíbrio no posicionamento dos vértices.

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

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

Esses cálculos são essenciais para garantir que os vértices estejam uniformemente espaçados dentro da grelha e corretamente alinhados em relação ao centro do plano.

### 2.1.3. Geração dos Vértices

Utilizamos loops aninhados para iterar sobre cada divisão na grelha. Dentro destes loops, calculamos as posições de quatro vértices (v1, v2, v3 e v4) que definem um quad representando um pequeno segmento do plano.

Para cada divisão, as coordenadas x e z dos vértices são determinadas com base nos índices de iteração actuais (i e j), no comprimento da metade e no comprimento da divisão. Este cálculo garante que os vértices estão uniformemente espaçados dentro da grelha, com o alinhamento correto relativamente ao centro do plano.

$$\text{v1} = (\text{halfLen} - j * \text{divLen}, 0.0f, \text{halfLen} - i * \text{divLen})$$

$$\text{v2} = (\text{halfLen} - j * \text{divLen}, 0.0f, \text{halfLen} - \text{divLen} - i * \text{divLen})$$

$$\text{v3} = (\text{halfLen} - \text{divLen} - j * \text{divLen}, 0.0f, \text{halfLen} - i * \text{divLen})$$

$$\text{v4} = (\text{halfLen} - \text{divLen} - j * \text{divLen}, 0.0f, \text{halfLen} - \text{divLen} - i * \text{divLen})$$

Para observar os resultados obtidos com os vértices gerados através deste método, consulte a Figure 5.

## 2.2. Cone

De seguida, a próxima primitiva a ser desenvolvida foi o cone que, ao contrário da última, começou a usar ângulos para que os pontos podessem ser calculados, aumentando assim a sua complexidade. O cone é constituido por vários *slices* verticais e *stacks* horizontais, gerando assim vários trapézios que são construídos por dois triângulos.

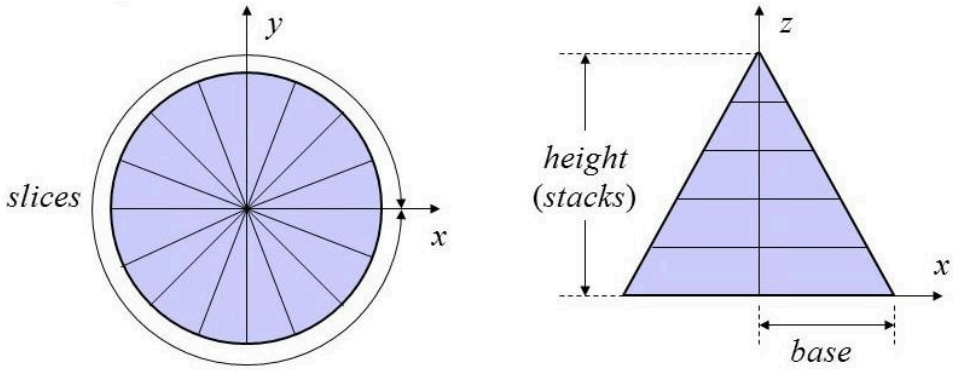


Figure 2: Slices e Stacks do Cone

Desta forma, o cone é construído de *stack* em *stack*, de baixo para cima. Primeiramente é calculado os valores comuns aos vértices dessa stack, ou seja, a altura dos pontos da base do trapézio e também a altura dos vértices do topo, bem como os respetivos raios nas duas alturas. Ou seja, sendo $i$ o número da stack a ser desenhada, *height* o valor da altura total do cone, *radius* o raio da base e *height_stack* o valor da altura de cada stack, os valores são computados da seguinte forma:

Valores da base:

$$\text{stack\_height\_base} = i * \text{height\_stack}$$

$$\text{radius\_base} = \text{radius} * \left(1 - \frac{\text{stack\_height\_base}}{\text{height}}\right)$$

Valores do topo:

$$\text{stack\_height\_top} = (i+1) * \text{height\_stack}$$

$$\text{radius\_top} = \text{radius} * \left(1 - \frac{\text{stack\_height\_top}}{\text{height}}\right)$$

A cada nível constrúido são realizados os seguintes cálculos para descobrir as restantes coordenadas dos pontos. Desta forma, seja *v1* e *v2* os vértices de baixo e *v3* e *v4* os vértices de cima. Para além disso, seja $\alpha$ o ângulo associado ao slice a desenhar e $\beta$ o ângulo do próximo slice.

$$\text{v1} = (\text{radius\_base} * \sin(\alpha), \text{stack\_height\_base}, \text{radius\_base} * \cos(\alpha))$$

$$\text{v2} = (\text{radius\_base} * \sin(\beta), \text{stack\_height\_base}, \text{radius\_base} * \cos(\beta))$$

$$\text{v3} = (\text{radius\_top} * \sin(\beta), \text{stack\_height\_top}, \text{radius\_top} * \cos(\beta))$$

$$\text{v4} = (\text{radius\_top} * \sin(\alpha), \text{stack\_height\_top}, \text{radius\_top} * \cos(\alpha))$$

Para visualizar o resultado obtido através dos vértices calculados com este método, consulte a Figure 7

## 2.3. Cubo

A primitiva do cubo é uma forma geométrica fundamental em gráficos 3D e serve como base para muitas estruturas e modelos mais complexos, sendo representada por seis faces quadradas.

### 2.3.1. Divisões do cubo

O cubo é definido pelas suas dimensões e pelo número de divisões ao longo de cada eixo. As divisões determinam a quantidade de detalhes e a suavidade das superfícies do cubo e são idênticas às divisões apresentadas durante a explicação do plano.

### 2.3.2. Cálculo das Posições dos Vértices

Para calcular os vértices do cubo, dividimos cada face do mesmo em segmentos com base no número de divisões especificado.

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

Aqui, *step* representa o tamanho do passo ao longo de cada eixo para criar as divisões. Com base nesses passos, calculamos os vértices para cada face do cubo.

### 2.3.3. Geração dos Vértices

Iteramos sobre cada face do cubo e, dentro de loops aninhados, calculamos os vértices para os segmentos dessa face.

Os vértices são gerados para cada face, começando pela face frontal e repetindo o processo para o resto das faces.

Os vértices de cada face são adicionados à lista de vértices do cubo, formando uma malha que representa a superfície do mesmo.

Para visualizar o cubo gerado por este método, consulte a Figure 6.

## 2.4. Esfera

A esfera é uma forma geométrica comum em gráficos 3D, representando uma superfície esférica. Para além disso, é das formas mais complexas desta fase do projeto.

### 2.4.1. Stacks e Slices da Esfera

A representação de uma esfera em gráficos 3D é frequentemente dividida em *stacks* e *slices*. As slices representam as fatias verticais da esfera, enquanto as stacks representam as fatias horizontais. Quanto maior o número de stacks e slices, mais detalhada será a representação da esfera.

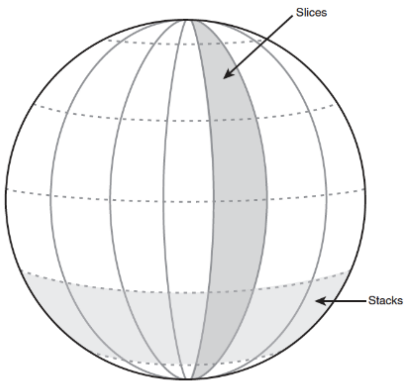


Figure 3: Slices e Stacks da Esfera

#### 2.4.2. Cálculo das Posições dos Vértices

Para calcular os vértices da esfera, utilizamos trigonometria esférica para determinar as coordenadas dos pontos. Começamos dividindo a esfera em stacks e slices.

$$\theta^1 = i * 2 * \frac{\pi}{\text{slices}}$$

$$\theta^2 = (i+1) * 2 * \frac{\pi}{\text{slices}}$$

$$\varphi^1 = j * \frac{\pi}{\text{stacks}}$$

$$\varphi^2 = (j+1) * \frac{\pi}{\text{stacks}}$$

Aqui, "$\theta$" representa a longitude e "$\varphi$" a latitude da esfera. Com base nessas coordenadas, calculamos os vértices da esfera.

#### 2.4.3. Geração dos Vértices

Dentro de loops aninhados que iteram sobre as slices e stacks, calculamos as posições dos vértices. Para cada ponto da esfera, utilizamos as coordenadas esféricas para calcular as coordenadas cartesianas.

$$\text{v1} = (\text{radius} * \sin(\varphi^1) * \cos(\theta^1), \text{radius} * \cos(\varphi^1), \text{radius} * \sin(\varphi^1) * \sin(\theta^1))$$

$$\text{v2} = (\text{radius} * \sin(\varphi^1) * \cos(\theta^2), \text{radius} * \cos(\varphi^1), \text{radius} * \sin(\varphi^1) * \sin(\theta^2))$$

$$\text{v3} = (\text{radius} * \sin(\varphi^2) * \cos(\theta^1), \text{radius} * \cos(\varphi^2), \text{radius} * \sin(\varphi^2) * \sin(\theta^1))$$

$$\text{v4} = (\text{radius} * \sin(\varphi^2) * \cos(\theta^2), \text{radius} * \cos(\varphi^2), \text{radius} * \sin(\varphi^2) * \sin(\theta^2))$$

Esses vértices formam quatro pontos que definem um quadrilátero na esfera. Repetindo esse processo para todas as slices e stacks, geramos uma malha de vértices que representa a superfície da mesma.

Para visualizar a esfera gerada por este método, consulte a Figure 8.

## 2.5. Torus

Por fim, como forma complementar, desenvolvemos ainda a forma geométrica chamada de "torus", ou mais vulgarmente denominada, "donut".

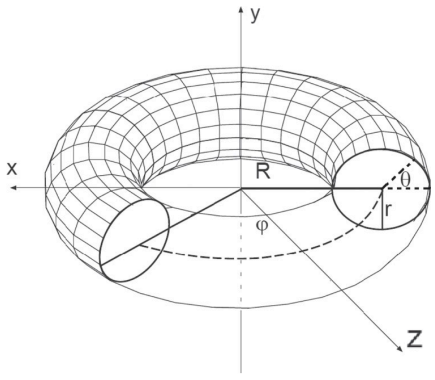


Figure 4: Slices e stacks do Torus

Como podemos ver na imagem acima, o sólido é dividio em vários *slices* e *stacks*, tal como a esfera. Para além disso, o objeto é ainda definido pelo raio principal (R), que é a distância do centro do tube até ao centro do torus, e o próprio raio do tubo (r). Cada ponto do torus tem em conta ainda 2 ângulos, $\varphi$, que é o ângulo do slice, e $\theta$, que é o ângulo da stack. Desta forma, as coordenadas dos pontos são dadas por:

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

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

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

Com as expressões definidas podemos calcular todos os vértices a ser desenhados à selhança das outras permitivas, com um ciclo a iterar pelos *slices* e outro, encadeado, a iterar pelas *stacks*. A cada iteração calcula-se o próximo ângulo para permitir o processamento dos 4 vértices de cada face.

Para visualizar a esfera gerada por este método, consulte a Figure 9.

# 3. Engine

A engine é uma estrutura de software projetada para facilitar a criação e manipulação de objetos tridimensionais num ambiente virtual. Ela oferece um conjunto de ferramentas e funcionalidades para renderização de gráficos 3D em tempo real, usando OpenGL como interface principal.

## 3.1. Parse de Ficheiros XML

O componente de parsing de ficheiros XML desempenha um papel crucial na configuração da cena 3D. Ele analisa um arquivo XML de configuração que define os parâmetros da câmara, janela e modelos a serem renderizados. O processo de parsing envolve a utilização da biblioteca TinyXML2 para extrair as informações necessárias do arquivo e inicializar os parâmetros da cena na engine. Esta abordagem permite uma configuração flexível e dinâmica sem a necessidade de recompilar o código-fonte.

## 3.2. Controlo da Câmara com Teclado

O controlo da câmara com o teclado permite aos utilizadores interagir dinamicamente com a cena 3D. As teclas das setas são utilizadas para mover a câmara para frente, para trás, para a esquerda e para a direita. Além disso, as teclas de página para cima e página para baixo controlam o movimento vertical da câmara. O controlo da câmara é implementado utilizando as funções de teclado do OpenGL, permitindo uma experiência de utilizador intuitiva e responsiva.

# 4. Resultados

Esta secção apresenta os resultados e conclusões do nosso projeto, centrando-se nas principais métricas, medidas de desempenho e observações notáveis. Discutimos os resultados de renderização obtidos com a engine, incluindo os resultados visuais das cenas 3D renderizadas. Além disso, são fornecidas informações sobre a exatidão e a fidelidade dos modelos processados em comparação com as suas representações esperadas. Através destas análises, pretendemos demonstrar a eficácia e as capacidades do nosso mini motor e gerador 3D na produção de gráficos 3D realistas e interactivos.

## 4.1. Plano

Os resultados da renderização do plano demonstram a capacidade da engine em representar primitivas geométricas simples de forma precisa e fiel. A figura mostra um plano perfeitamente definido, com arestas nítidas e uma superfície uniforme. A renderização do plano atende às expectativas em termos de exatidão e fidelidade, proporcionando uma base sólida para a construção de cenas mais complexas.

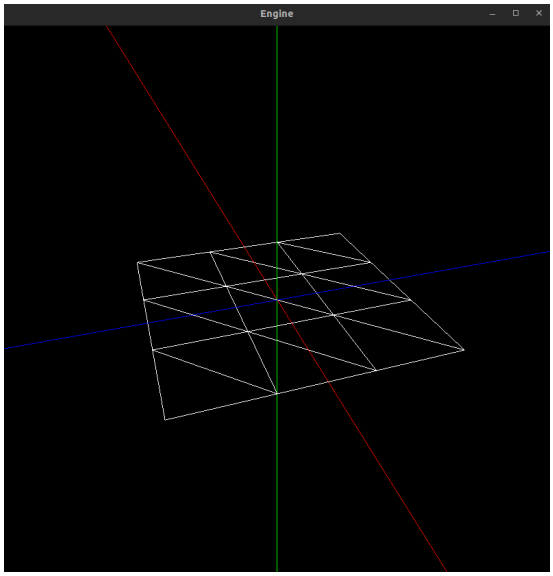


Figure 5: Plano Renderizado

## 4.2. Cubo

Os resultados da renderização do cubo destacam a capacidade da engine em manipular objetos tridimensionais complexos. A figura mostra um cubo com todas as faces claramente definidas e ângulos precisos entre elas. A renderização do cubo demonstra uma representação visual fiel às suas especificações, com uma geometria sólida e detalhes precisos em cada face. Este resultado valida a capacidade da engine em lidar com objetos tridimensionais mais elaborados.

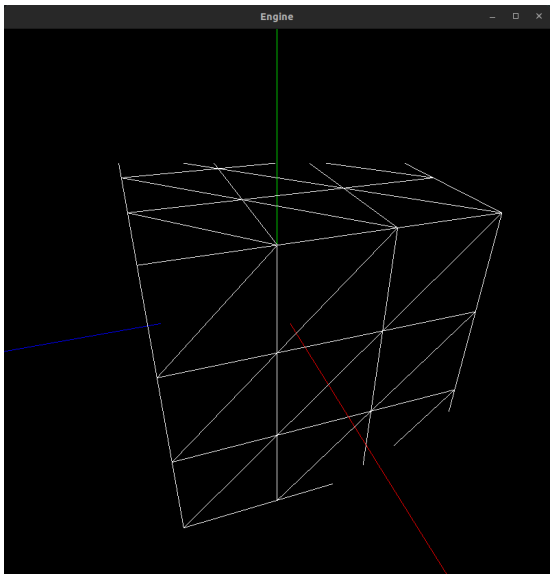


Figure 6: Cubo Renderizado

## 4.3. Cone

Os resultados da renderização do cone ilustram a capacidade da engine em lidar com formas tridimensionais que possuem geometria mais complexa, como o cone. A figura apresenta um cone com uma estrutura consistente e bem definida, composta por várias fatias verticais e horizontais. A renderização do cone demonstra uma representação visual precisa da forma cônica, com superfícies suaves e transições naturais entre os segmentos.

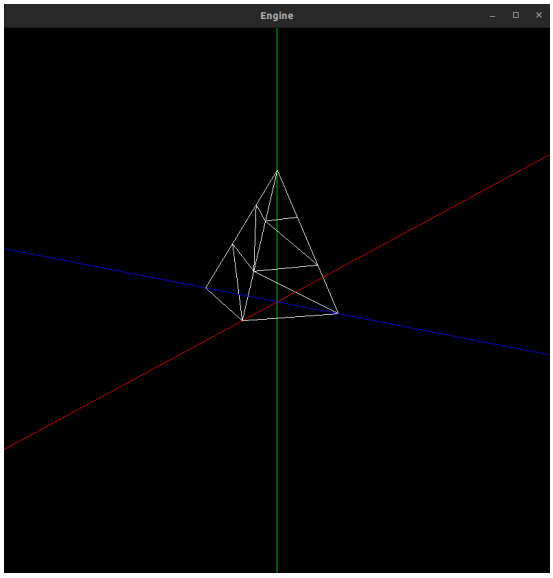


Figure 7: Cone Renderizado

## 4.4. Esfera

A renderização da esfera destaca a habilidade da engine em lidar com formas curvas e superfícies suaves. A figura mostra uma esfera com curvaturas suaves e uma distribuição uniforme de vértices ao longo de sua superfície. A renderização da esfera apresenta uma representação realista da forma esférica. Este resultado demonstra a capacidade da engine em produzir gráficos 3D realistas e interativos, mesmo para formas complexas como uma esfera.

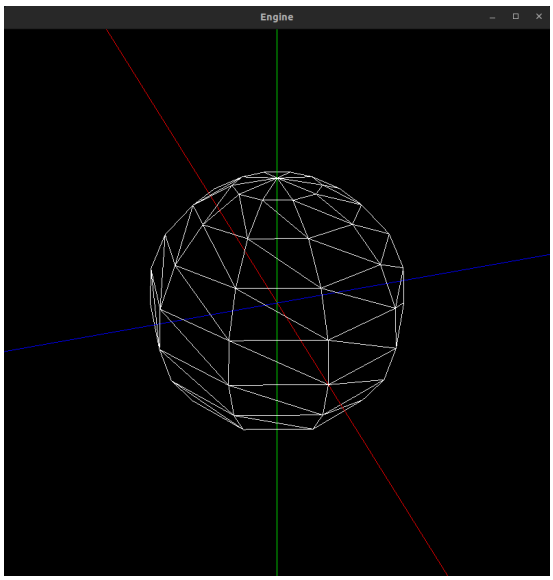


Figure 8: Esfera Renderizada

### 4.5. Torus

A renderização do torus destaca a versatilidade da engine em manipular formas geométricas complexas com precisão e detalhes. A figura exibe um torus com curvaturas suaves e uma distribuição uniforme de vértices ao longo da sua superfície circular. A renderização do torus oferece uma representação visual fiel à sua forma característica de "donut". Este resultado demonstra mais uma vez a capacidade da engine de produzir gráficos 3D realistas e interativos, mesmo para formas geométricas complexas e distintas como o torus.

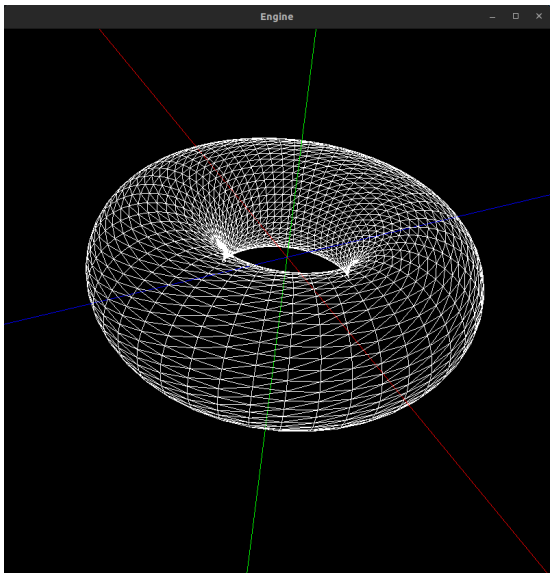


Figure 9: Torus Renderizado

## 5. Conclusões

Ao completar este projeto do mini motor 3D e gerador de primitivas, alcançamos uma compreensão mais profunda dos desafios e complexidades envolvidos no desenvolvimento de sistemas de renderização 3D em tempo real. Este trabalho reflete o nosso compromisso em criar um sistema eficiente e modular para produzir gráficos 3D realistas e interativos.

Estamos satisfeitos com os resultados alcançados e orgulhosos por termos cumprido todos os requisitos do projeto. No entanto, reconhecemos que ainda há espaço para melhorias em várias áreas:

- **Eficiência da Renderização:** Observamos que a eficiência da renderização dos modelos poderia ser melhorada. Embora o sistema seja capaz de renderizar modelos 3D com precisão, há espaço para otimizações que poderiam melhorar o desempenho, especialmente ao lidar com modelos mais complexos ou cenas com um grande número de objetos.
- **Funcionalidades da Câmara:** Embora a funcionalidade básica de controle da câmara esteja implementada, reconhecemos que a câmara poderia oferecer recursos mais avançados para melhorar a experiência do utilizador.
- **Interface Gráfica Aprimorada:** A interface gráfica do sistema poderia ser mais completa, oferecendo informações adicionais aos utilizadores, como o número de frames por segundo (FPS), o ângulo atual da câmara e outros dados relevantes da cena. Essas informações adicionais ajudariam os utilizadores a monitorar o desempenho do sistema e a entender melhor a configuração da cena em tempo real.