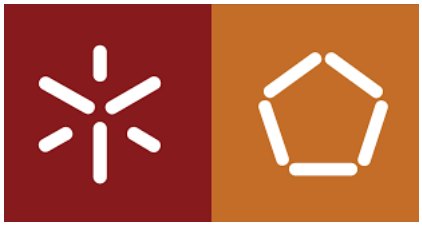

Universidade do Minho
Escola de Engenharia | Departamento de Informática
Licenciatura em Engenharia Informática

Computação Gráfica

Ano Letivo de 2023/2024

Fase 4 - Normals and Texture Coordinates

Ana Margarida Sousa Pimenta (A100830)
Miguel Tomás Antunes Pinto (A100815)
Inês Gonzales Perdigão Marques(A100606)
Pedro Miguel Costa Azevedo (A100557)

PL

Índice

1 Introdução

A quarta e última fase deste projeto de Computação Gráfica foca-se no aprimoramento da representação gráfica através da integração de normais e coordenadas de textura, essenciais para a ativação das funcionalidades de iluminação e texturização no engine desenvolvido. Este avanço é crucial para aumentar o realismo e a profundidade visual das cenas geradas, permitindo uma interação mais rica e imersiva com os modelos 3D.

Nesta fase, foi pedido para modificar o generator para que ele produza, além das informações de vértices e faces, as coordenadas de textura e as normais para cada vértice. Estes dados são fundamentais para o correto mapeamento de texturas e para a aplicação eficaz de técnicas de iluminação, que dependem das normais para calcular como a luz interage com as superfícies dos objetos.

Além disso, o engine será expandido para suportar a leitura e aplicação dessas novas informações provenientes dos arquivos de modelo.

Definimos ainda as fontes de luz dentro do arquivo de configuração XML, escolhendo entre luzes pontuais, direcionais e de foco, para iluminar as cenas de maneira controlada e efetiva.

Este projeto culmina na criação de uma cena animada do sistema solar, onde os planetas não só estarão corretamente texturizados, mas também iluminados.

2 Generator

A quarta fase deste projeto envolveu a introdução de normais e coordenadas de textura nos modelos geométricos gerados pelo nosso generator. Essa implementação teve como objetivo principal enriquecer a qualidade visual dos modelos 3D no engine, permitindo a utilização de técnicas avançadas de iluminação e texturização. Com essas melhorias, a renderização dos modelos não só ganha em realismo visual, mas também em profundidade e detalhe, essenciais para a criação de uma experiência gráfica mais imersiva.

2.1 Introdução de Normais e Coordenadas de Textura

As normais são vetores unitários que apontam para fora da superfície de um objeto geométrico. São cruciais para calcular o efeito da luz sobre as superfícies, influenciando diretamente o cálculo da iluminação difusa e especular conforme as leis de reflexão da luz. No nosso código, a normal de cada vértice é calculada e normalizada para garantir que o seu comprimento seja unitário, facilitando assim os cálculos de iluminação no shader que será usado no motor gráfico.

As coordenadas de textura, por outro lado, mapeiam uma textura 2D nos modelos 3D. Cada vértice do modelo recebe coordenadas (u, v), que são usadas para determinar que parte da textura é mapeada em cada face. Este processo é fundamental para adicionar detalhes visuais aos objetos sem a necessidade de aumentar a complexidade geométrica dos modelos.

2.2 Implementação no Código

O processo de escrita dos arquivos foi ajustado para incluir não apenas os vértices, mas também as normais e coordenadas de textura. A função writeToFile foi modificada para armazenar essas novas informações, formatando cada linha do arquivo com os valores de vértice, normal e coordenada de textura correspondente. Este formato estendido

permite que o engine leia e utilize estas informações diretamente, facilitando a integração entre a geração dos modelos e sua renderização.

Em termos de cálculo, as normais são geradas com base na orientação das faces para figuras geométricas simples, como planos e caixas, onde a direção é constante e conhecida. Para figuras mais complexas, como esferas e cones, as normais são derivadas das próprias posições dos vértices, garantindo que apontem na direção correta.

Nesta fase, atualizamos as nossas structs fazendo as seguintes alterações:

Vector3 : Esta struct representa um vetor 3D com coordenadas x, y e z. Ela inclui operações básicas de vetor, como subtração, adição, multiplicação por um escalar, cálculo do produto vetorial (cross product) e normalização (conversão para um vetor unitário).

Vector2 : Esta struct representa um vetor 2D com coordenadas u e v. É utilizada principalmente para armazenar coordenadas de textura.

Quanto a funções, alteramos as funções usadas previamente e acrescentamos novas, sendo essas as seguintes:

Lerp : Esta função realiza uma interpolação linear entre dois vetores 3D.

formatNumber : Esta função formata um número float como uma string, garantindo que, se o número for um inteiro, ele seja representado sem casas decimais, e, se for um float, ele seja representado com quatro casas decimais.

WriteToFile : Esta função escreve os vértices, normais e coordenadas de textura em um arquivo.

GeneratePlane : Esta função gera um plano e armazena seus vértices, normais e coordenadas de textura. A normal neste caso é pre-definida como 0,1,0 visto a normal ser igual para todo o plano.

GenerateBox : Esta função gera uma caixa com as faces subdivididas em divisões, calculando vértices, normais e coordenadas de textura para cada face. Da mesma forma que o plano, as normais também foram pré-definidas visto que temos uma normal por plano.

Theta : Esta função calcula o ângulo teta para uma determinada slice de uma esfera. É usada para determinar a posição angular de cada vértice ao longo da superfície da esfera.

generateSphere : Esta função gera uma esfera subdividida em slices e stacks. Ela calcula os vértices, normais e coordenadas de textura da esfera e escreve-os num arquivo. Primeiramente calculamos os vértices usando coordenadas esféricas, normalizamos os vértices para obter as normais e calculamos as coordenadas de textura baseadas nas posições angulares. Para o cálculo das normais primeiro, geramos os vértices da esfera

usando coordenadas esféricas, usando o ângulo theta e o ângulo polar (phi).

Para cada stack da esfera:

Calculamos o ângulo polar (phi) e seus senos (sin_phi) e cossenos (cos_phi). Dentro de cada stack, iteramos sobre os segmentos circulares (slices). Calculamos o ângulo theta e seus senos (sin_theta) e cossenos (cos_theta).

Para cada vértice:

Calculamos as coordenadas x, y, e z usando as fórmulas das coordenadas esféricas. Criamos o vetor posição do vértice Vector3(x, y, z). A normal é calculada normalizando o vetor posição (pois o centro da esfera é a origem).

generateCone : Esta função gera um cone com um dado raio e altura, subdividido em slices e stacks. Ela calcula os vértices, normais e coordenadas de textura do cone e escreve-os num arquivo.Calculamos os vértices da base do cone e as suas normais dividindo a altura do cone em stacks e calcula os vértices correspondentes a cada nível. Para o cálculo das normais primeiro, geramos os triângulos para a base circular do cone. As normais para estes triângulos são constantes e apontam diretamente para baixo ao longo do eixo y negativo (0, -1, 0), uma vez que a base é plana e horizontal.Para calcular as normais da superfície lateral do cone, seguimos os seguintes passos:

Dividimos a altura do cone em várias stacks. Cada camada é composta por slices. Para cada segmento (par de triângulos), calculamos as normais com base nos vértices que formam o triângulo.

readPatchesFile : Esta função lê um arquivo contendo patches de Bezier e devolve uma estrutura contendo os patches lidos.

multiplyMatrices : Esta função multiplica duas matrizes e armazena o resultado em uma terceira matriz.

surfacePoint : Esta função calcula um ponto numa superfície de Bezier para os parâmetros dados u e v. Primeiramente calcula os vetores U e V baseados nos parâmetros u e v e realiza multiplicações matriciais para obter as coordenadas do ponto na superfície e armazena as coordenadas calculadas no vetor "res".

buildPatches : Esta função gera uma malha de superfícies de Bezier baseada nos patches lidos de um arquivo. Ela calcula os vértices, normais e coordenadas de textura para os patches e escreve-os num arquivo. Esta função lê os patches do arquivo especificado, divide cada patch em tesselações menores, calcula os pontos na superfície usando surfacePoint para cada subdivisão, calcula as normais dos triângulos gerados, normaliza os vetores normais e por fim calcula as coordenadas de textura. As normais são calculadas usando as derivadas parciais da posição do ponto na superfície em relação às coordenadas u e v.

Para cada par de pontos calculados (A, B, C, D):

Calculamos as duas derivadas parciais: p/u e p/v. A normal em cada ponto é o produto vetorial dessas duas derivadas. Especificamente:

Para os pontos (A, B, C, D) calculados para as coordenadas (u, v), (u, v + delta), (u + delta, v), (u + delta, v + delta): Calculamos p/u como a diferença entre os pontos C e A e calculamos p/v como a diferença entre os pontos B e A. A normal é o produto vetorial desses dois vetores e depois normalizado.

3 Engine

3.1 Estruturas de Dados

No nosso *engine*, começamos por definir várias estruturas de dados, as quais são necessárias para o desenvolvimento e realização do trabalho realizado. Definimos então várias estruturas para representar os componentes de cada cena, como por exemplo as luzes, os materiais, os modelos, as transformações e grupos.

Estas estruturas permitem-nos então organizar e manipular os dados de forma mais eficiente.

De seguida, apresentaremos as seguintes estruturas.

Window: Representa a janela da aplicação.

Camera: Define os parâmetros da câmera, incluindo posição, direção de visualização (lookAt), vetor up e parâmetros de projeção.

Point3D, Vector2, Vector3: Representam pontos e vetores em 2D e 3D. Color: Define cores com componentes vermelho, verde e azul.

Material: Armazena as propriedades do material, como cores difusa, ambiente, especular e emissiva, além da shininess.

Light: Define as propriedades de uma luz, incluindo o tipo, posição, direção e cores.

Model: Representa um modelo 3D, incluindo vértices, normais, coordenadas de textura, material e textura.

Translate, Rotate, Transform: Armazenam informações de transformações, como translação, rotação e escala.

Group: Agrupa modelos e subgrupos, permitindo hierarquia de transformações.

World: Armazena todas as configurações, incluindo janela, câmera, grupos e luzes.

3.2 Processamento dos Modelos

Os modelos 3D são carregados a partir de ficheiros .3d. Para isso, utilizamos várias funções que trabalham em conjunto para ler e processar os dados do modelo.

Primeiro, a função ***readModel*** abre o ficheiro .3d, lê os vértices, normais e coordenadas de textura, e armazena esses dados na estrutura ***Model***. Este processo inclui a leitura do número total de pontos, a leitura dos dados dos pontos e o armazenamento desses dados na respetiva estrutura apropriada.

Depois de ler os dados, a função ***initializeVBO*** cria e configura os VBOs (Vertex Buffer Objects) e VAOs (Vertex Array Objects) para enviar os dados à GPU. Isto envolve a criação dos VBOs, a alocação de memória para os dados e o preenchimento dos *buffers* com os dados do modelo.

Além disso, a função ***parseModels*** lê os elementos *XML* ***<model>*** dentro de um grupo e carrega cada modelo utilizando a função ***readModel***.

Esta função também processa as cores e texturas associadas a cada modelo, utilizando a função ***parseColor*** para configurar as propriedades do material e carregar texturas, se especificadas.

Para além de termos funções de parse para elementos individuais, como a ***parseModels*** e a ***parseColor***, por exemplo, temos uma parseXML que é responsável por conjugar todas essas formando um parse "geral".

3.3 Renderização de Modelos e Grupos

Falando agora da renderização, esta consiste no processo de transformar os dados dos modelos 3D em imagens 2D.

Nas nossas funções de renderização, adotamos métodos específicos para garantir uma renderização eficiente e visualmente apelativa. Deste modo, iremos focar, de uma forma mais geral, nas funções ***renderModel*** e ***renderGroup***, que são cruciais para a renderização da cena em questão.

A função ***renderModel*** é responsável por desenhar um modelo 3D específico. Esta função aplica as transformações, materiais e texturas configurados para o modelo, e utiliza os comandos *OpenGL* para desenhar os vértices do modelo.

A função ***renderGroup*** é responsável por aplicar as transformações e renderizar cada grupo de modelos e os seus subgrupos recursivamente. Esta abordagem permite então garantir uma hierarquia de transformações, onde as transformações de um grupo pai

afetam todos os seus subgrupos.

Contudo, a função ***display*** é a função principal de renderização no programa, chamada repetidamente pelo *GLUT* para atualizar a cena.

Nesta função, configuramos a câmera e a projeção, e preparamos o ambiente de renderização através das funções auxiliares acima abordadas.

3.4 Aplicação de Transformações

Passando agora à transformações, este processo, que inclui translação, rotação e escala, é aplicado aos grupos e modelos. Aqui utilizamos a função de parse auxiliar, a ***parseTransform***, que é responsável por ler as configurações de transformações a partir de um ficheiro *XML*. Esta função armazena então os valores de translação, rotação e escala na estrutura *Transform*.

De modo a seguir o conteúdo lecionado e pedido na realização deste projeto, incorporamos também as curvas *Catmull-Rom*, que permitem interpolar entre pontos de controlo. A função *renderGroup* aplica essas transformações na cena, desenhando os modelos e subgrupos com as transformações apropriadas. Esta função é chamada recursivamente para aplicar transformações hierárquicas nos grupos e subgrupos, como falamos previamente.

3.5 Configuração da Câmera

Quanto às configurações da câmera, definimos a sua posição, direção de visualização (*lookAt*), *vetor up* e parâmetros de projeção. Utilizamos aqui a função ***parseCamera*** que é responsável por ler essas configurações a partir de um *XML*. Esta função lê os elementos correspondentes e armazena seus valores na estrutura *Camera*.

A configuração da câmera envolve definir a matriz de projeção e a matriz de visualização. A matriz de projeção é responsável por definir o campo de visão e os planos *near* e *far*, enquanto a matriz de visualização define a posição da câmera, a direção para onde ela está a olhar e o vetor up que orienta a câmera verticalmente.

Para além disso, é possível, na nossa implementação movimentar a câmara através da do teclado, o qual exemplificaremos posteriormente no **Manual de Utilização**.

3.6 Iluminação, Materiais e Texturas

3.6.1 Iluminação

De modo a implementar a ilumінção, projetamos a função ***initializeLighting*** que é responsável por configurar a iluminação global e as luzes individuais na cena.

Este processo envolve a ativação da iluminação, a normalização dos vetores normais e a configuração das propriedades das luzes.

3.6.2 Iluminação

A função ***initializeLighting*** é responsável por configurar a iluminação global e as luzes individuais na cena. Este processo envolve a ativação da iluminação, a normalização dos vetores normais e a configuração das propriedades das luzes.

Primeiro, utilizamos glEnable para o GL LIGHTING para ativar o sistema de iluminação no OpenGL e glEnable para o GL NORMALIZE para garantir que os vetores normais sejam normalizados, o que é importante para a correta aplicação da iluminação, especialmente após transformações de escala.

De seguida, definimos a iluminação global com glLightModelfv, onde globalAmbient é um *array* que define a cor da luz ambiente global, assegurando que a cena tenha uma iluminação básica, mesmo sem fontes de luz específicas.

Para configurar as luzes individuais, iteramos sobre a lista de luzes definida na configuração do mundo (worldConfig.lights) e configuramos cada luz individualmente.

Para cada luz, ativamos a luz correspondente utilizando glEnable para o lightId, onde lightId é calculado com base no índice da luz (GL LIGHT0 + i). Configuramos as propriedades das luzes (ambiente, difusa e especular) utilizando glLightfv para definir as cores dessas componentes.

Dependendo do tipo de luz (direcional ou pontual), configuramos a posição ou direção da luz com glLightfv(GL POSITION, position/direction). Este processo garante que cada luz na cena seja configurada corretamente, permitindo iluminações mais dinâmicas.

3.6.3 Materiais

A função **applyMaterial** é responsável por configurar as propriedades do material de um modelo antes da sua renderização.

Os materiais determinam como a superfície de um objeto reage à luz, afetando sua aparência final.

Deste modo, utilizamos **glMaterialfv** para definir as cores ambiente, difusa, especular e emissiva do material. Estes parâmetros são passados como arrays de valores (GLfloat), representando as componentes RGB das cores. Utilizamos também o **glMaterialf** para definir o coeficiente de brilho (shininess), que controla o tamanho e a intensidade dos reflexos especulares na superfície do modelo.

As propriedades específicas do material incluem:

Ambiente: Cor ambiente do material que reage com a luz ambiente da cena.

Difusa: Cor difusa do material que determina como a luz espalha na superfície.

Especular: Cor especular que define a cor dos reflexos brilhantes.

Emissiva: Cor emissiva que define a cor que o material parece emitir.

Shininess: Valor que controla o brilho especular, afetando a nitidez dos reflexos.

Esta função garante que cada modelo seja renderizado com as propriedades de material corretas, permitindo a criação de superfícies realistas que respondem de maneira adequada à iluminação da cena.

3.6.4 Texturas

A função ***loadTexture*** é responsável por carregar uma imagem de textura a partir de um ficheiro, configurar os parâmetros de textura e gerar um identificador de textura para uso no OpenGL.

Para executar a aplicação das texturas, primeiro, utilizamos a *biblioteca stb image* para carregar a imagem de textura. A função *stbi load* lê a imagem do ficheiro e retorna os dados da imagem, juntamente com suas dimensões (largura e altura) e o número de canais de cor. Se a imagem não puder ser carregada, é exibida uma mensagem de erro.

Para a aplicação deste processo, guiamo-nos pela informação reunida em vídeos na Internet, procurando assim garantir que as texturas sejam carregadas e configuradas corretamente, adicionando detalhes visuais às superfícies dos modelos e melhorando a aparência geral da cena.

4 Demonstração

4.1 Manual de utilização

De forma a tornarmos o nosso sistema mais iterativo adicionamos os seguintes comandos:

- **<- :** rotação da câmara para a esquerda
- **-> :** rotação da câmara para a direita
- **v :** rotação da câmara para baixo
- ˆ rotação da câmara para cima
- **+ :** zoom in
- **- :** zoom out
- **q :** sair do programa

5 Resultados dos testes

Neste ponto, demonstramos os resultados de todos os testes disponibilizados na Black-Board.

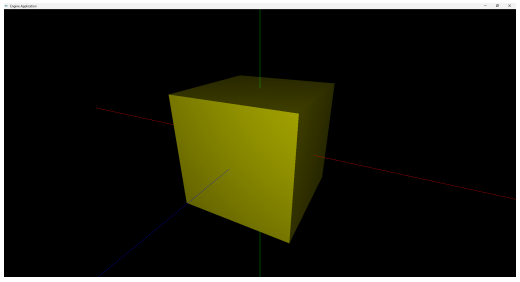

Figura 5.1: Teste 4.1

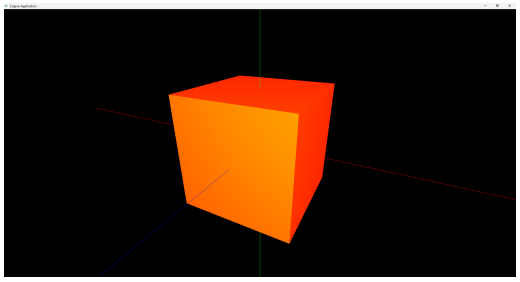

Figura 5.2: Teste 4.2

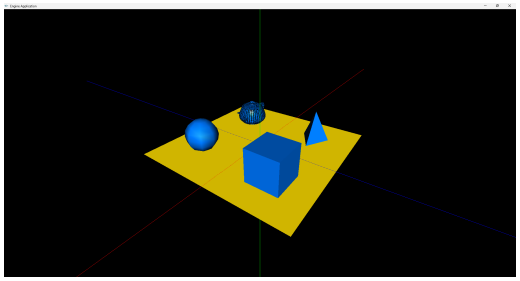

Figura 5.3: Teste 4.3

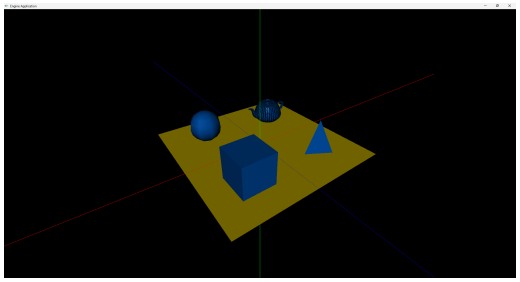

Figura 5.4: Teste 4.4

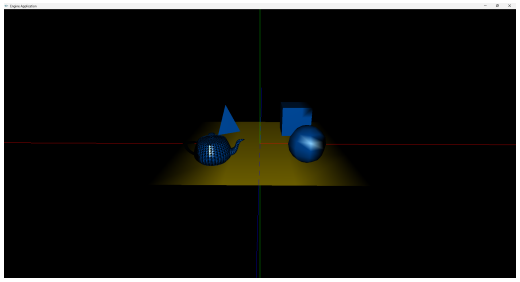

Figura 5.5: Teste 4.5

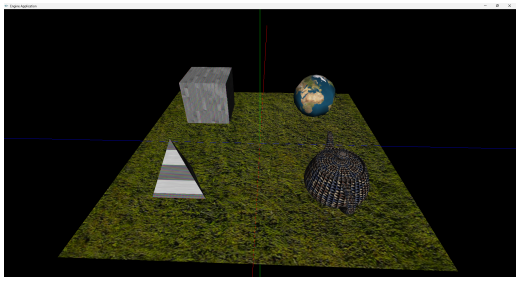

Figura 5.6: Teste 4.6

6 Sistema Solar

Para esta última fase, implementamos ainda um sistema solar dinâmico com texturas para os respetivos planetas, como é possível observar nas imagens abaixo.

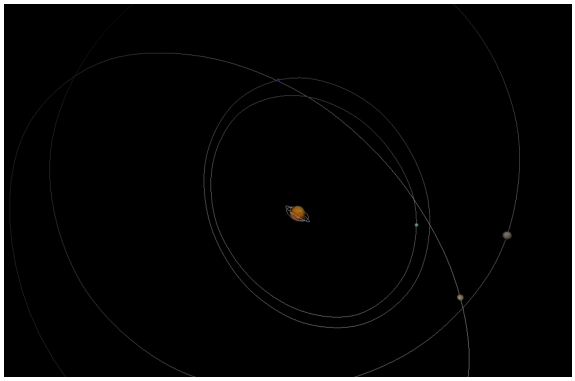

Figura 6.1: Solar System 1

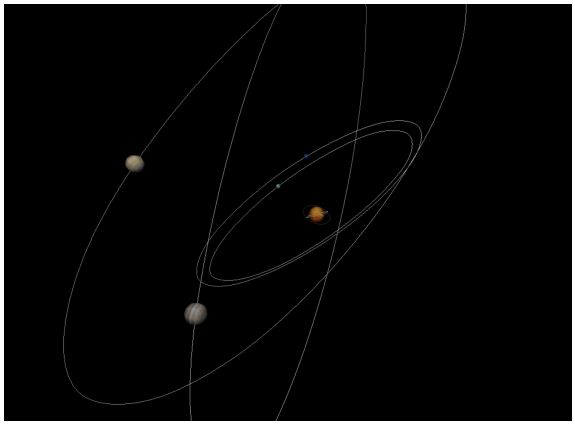

Figura 6.2: Solar System 2

7 Conclusão

A quarta fase do desenvolvimento do motor gráfico 3D representou um marco significativo no aprofundamento das nossas capacidades de renderização gráfica, através da integração de normais e coordenadas de textura aos modelos gerados. Esta fase não só fortaleceu a nossa compreensão teórica e prática sobre processamento gráfico avançado, mas também elevou a qualidade visual das simulações realizadas pelo motor.

Com a implementação das normais, conseguimos obter uma iluminação mais realista e detalhada nos modelos, permitindo que os efeitos de luz e sombra fossem representados de maneira mais fiel. A adição de coordenadas de textura possibilitou a aplicação de diversos materiais e acabamentos às superfícies dos modelos, enriquecendo significativamente a experiência visual sem comprometer a performance do sistema.

Embora o processo de implementação tenha apresentado desafios, particularmente na correção de cálculos das normais em superfícies curvas e na otimização do mapeamento de texturas, as soluções encontradas contribuíram para o aprimoramento do nosso conhecimento técnico.

Tal como podemos verificar na secção dos Resultados Obtidos nos testes, obtivemos sucesso em grande parte deles, no entanto, tivemos alguns problemas nos últimos testes no que toca à textura do cone e da caixa. Apesar de não conseguirmos detetar o problema a tempo da entrega, iremos trabalhar mesmo assim para encontrar a sua raiz e corrigir esta questão.

A colaboração entre os membros da equipa foi essencial para superar esses obstáculos, destacando a importância da comunicação efetiva e do trabalho em equipa no desenvolvimento de soluções inovadoras.

Além disso, esta fase permitiu a realização de testes mais rigorosos e a validação das funcionalidades implementadas, através dos quais identificámos e corrigimos várias inconsistências que, de outra forma, poderiam comprometer a funcionalidade do engine em cenários de uso real.

Em suma, esta quarta fase não apenas cumpriu com os requisitos técnicos estabelecidos, como também proporcionou uma plataforma robusta para futuras expansões do projeto. Através deste projeto, continuamos a demonstrar o nosso compromisso com a excelência técnica e a inovação para futuros desafios na área de computação gráfica.