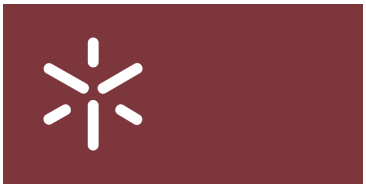


Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Computação Gráfica

Ano Letivo de 2024/2025

Texturas, Normais e Iluminação
Fase 4

Afonso Pedreira A104537
André Pinto A104267
Fábio Magalhães A104365

Abril, 2025

CG

Índice

1. Introdução

Este relatório tem como objetivo descrever o trabalho desenvolvido na quarta fase do projeto, no âmbito da unidade curricular de Computação Gráfica. Nesta fase, foram introduzidas várias funcionalidades adicionais, tanto ao nível do **engine** como do **generator**, permitindo uma evolução significativa da aplicação em termos de realismo visual e complexidade técnica.

No que diz respeito ao **generator**, foi necessário adaptá-lo para gerar, além da geometria básica, os vetores normais e as coordenadas de textura associadas a cada vértice. Esta alteração tornou os ficheiros de modelo mais ricos em informação, o que permite ao **engine** representar superfícies com iluminação mais realista e aplicar texturas de forma precisa.

Por sua vez, no **engine**, foram realizadas várias modificações para que este fosse capaz de interpretar e utilizar os novos dados gerados. Passou a ser possível processar vetores normais e coordenadas de textura, bem como aplicar materiais e texturas aos modelos. Além disso, foram também introduzidas luzes na cena, permitindo assim simular diferentes tipos de iluminação e dar mais profundidade e realismo ao ambiente renderizado.

No geral, esta fase representou um passo importante na aproximação do *engine* a um **pipeline** de renderização mais completo e moderno, abrindo portas para funcionalidades futuras mais avançadas.

2. Estrutura do Projeto

O projeto está dividido em dois programas principais: ***generator*** e ***engine***, que trabalham em conjunto para gerar e renderizar figuras gráficas. A estrutura do projeto é organizada de forma a facilitar a modularidade e a reutilização de código.

A arquitetura do projeto é composta pelas seguintes pastas principais:

- **engine**: Esta pasta contém os componentes responsáveis pela renderização das cenas. Inclui elementos essenciais como:
  - **camera**: Define a perspetiva e a posição da visualização da cena.
  - **configuration**: Contém as configurações gerais do sistema, como a `window`, as definições da câmara e informação sobre a cena.
  - **window**: Responsável pela gestão dos parâmetros da janela de visualização.
  - **ModelGroup**: Responsável por armazenar todos os modelos pertencentes a uma cena e respetivas transformações.
  - **Model**: Responsável pelo armazenamento dos pontos dos respetivos modelos, incluindo toda a lógica de **VBOs**
  - **filesParser**: Este módulo é responsável por fazer o `parsing` dos ficheiros de entrada (`.xml`) e configurar os dados necessários para a renderização.
  - **catmullCurves**: Responsável por armazenar toda a lógica relacionada com translações ao longo de uma curva, rotações, cálculo de posições na curva, entre outros.
- **generator**: Esta pasta contém os módulos responsáveis pela geração das figuras geométricas. Cada figura é gerada a partir de parâmetros específicos e guarda num ficheiro `.3d`. Estes ficheiros, posteriormente, serão embutidos em ficheiros `.xml`, que a *engine* utiliza para renderizar as cenas.
- **common**: Esta pasta contém componentes compartilhados entre os dois programas principais. Inclui definições essenciais, como a definição de **Ponto** bem como funções auxiliares para tarefas comuns, como:
  - **salvar em ficheiro**: Funções para escrever os dados gerados em ficheiros de diferentes formatos.
  - **ler de ficheiro**: Funções para ler dados de ficheiros de entrada, como `.xml`,`.obj` e `.patch`.
  - **criar ficheiro 3D**: Funções que permitem a criação e manipulação de ficheiros no formato `.3d`.

A organização modular do projeto permite uma gestão eficiente de cada componente, garantindo a escalabilidade e uma boa base para as próximas fases.

3. Alterações no Generator

Um dos requisitos para esta quarta e última fase foi a inclusão da geração dos vetores normais e das coordenadas de textura para todas as figuras geométricas desenvolvidas na primeira fase do projeto. Assim, foi necessário modificar a lógica de geração de cada uma das figuras para que estas passassem a conter essa informação adicional.

As figuras afetadas por estas alterações foram:

- Plano
- Cubo
- Cone
- Donut (Torus)
- Esfera
- Cilindro
- Curvas de Bézier

Além destas modificações nas figuras, a estrutura dos ficheiros `.3d` gerados também foi atualizada. Inspirámo-nos em formatos comuns como o `.obj`, que podem ser encontrados facilmente em sites como Free3D ou o TurboSquid, onde os modelos incluem não só as coordenadas dos vértices, mas também as normais e as coordenadas de textura.

A partir desta fase, cada linha dos ficheiros `.3d` passou a conter a seguinte informação, separada por espaços:

`x, y, z  nx, ny, nz  u, v`

Ou seja, para cada ponto, guardamos:
- `x, y, z`: coordenadas do vértice
- `nx, ny, nz`: vetor normal
- `u,v`: coordenadas de textura

```
0.59  1.72  -3.48  0.11  0.88  -0.46  0.22  0.77
-2.45  -0.64  1.89  -0.31  0.41  0.85  0.94  0.13
3.26  -1.11  -0.58  0.68  -0.72  0.13  0.06  0.59
-0.36  2.80  -1.94  -0.57  0.12  0.81  0.88  0.44
1.09  -3.33  0.75  0.91  -0.29  0.26  0.33  0.97
-1.76  0.48  2.14  -0.22  -0.93  0.29  0.71  0.16
0.48  -2.22  -2.49  0.12  0.38  -0.92  0.57  0.08
2.85  1.61  -0.73  0.76  0.13  0.63  0.19  0.39
-3.14  -1.85  1.05  -0.42  0.64  0.64  0.02  0.50
0.73  -0.93  2.66  0.34  -0.82  0.45  0.80  0.27
1.98  0.10  -1.38  0.07  0.58  -0.81  0.49  0.92
-2.69  2.55  -0.42  -0.87  -0.14  0.47  0.11  0.66
```

Listing 1: Nova estrutura .3d

3.1. Alterações nas Figuras Geométricas

Esta alteração foi, talvez, a mais complexa enfrentada pelo grupo, uma vez que não foi imediatamente óbvio quais coordenadas seriam necessárias. Contudo, após analisar o código da fase anterior, percebemos que não seria necessário fazer grandes alterações, mas sim expandir a informação presente nos modelos, adicionando os novos dados de forma a integrar as coordenadas de textura e os vetores normais.

3.1.1. Plano

3.1.1.1. Coordendas Normais

O plano foi, sem dúvida, a figura mais simples de modificar, pois está situado no plano **XZ**, com $\mathbf{Y = 0}$. Assim, todos os pontos do plano têm vetores normais iguais, ou seja, (0, 1, 0), o que torna o cálculo direto e sem grande complexidade.

3.1.1.2. Coordendas Texturas

Quanto às coordenadas de textura, estas foram aplicadas de forma a repetirem-se ao longo do plano em cada subdivisão. Para calcular as coordenadas de textura, considerámos o número total de subdivisões do plano, bem como a subdivisão em que nos encontrávamos durante a iteração. Dado que o número de cada subdivisão nunca ultrapassa o número total de divisões, as coordenadas de textura variam entre 0 e 1, como esperado, garantindo que a aplicação da textura seja homogénea e sem distorções visíveis.

```
float u1 = static_cast<float>(col) / divisions;
float u2 = static_cast<float>(col + 1) / divisions;
float v1 = static_cast<float>(row) / divisions;
float v2 = static_cast<float>(row + 1) / divisions;
```

Listing 2: Coordendas Textura Plano

3.1.2. Cubo

A par do plano, o cubo também foi relativamente simples de modificar, variando apenas numa pequena nuance nas coordenadas das normais, dependendo da face em que o vértice se encontrava. As coordenadas de textura seguiram um fluxo semelhante, com a diferença de que agora utilizamos a coordenada do vértice somada ao tamanho do lado do cubo, dividida pela sua comprimento total, de forma a garantir que as coordenadas fiquem entre 0 e 1.

3.1.2.1. Coordenadas Normais

As coordenadas normais no cubo variam de acordo com a face em que o vértice está localizado. Cada face tem uma normal própria, conforme descrito a seguir:

- A face frontal terá uma normal (0, 0, 1).
- A face traseira terá uma normal (0, 0, −1).
- A face superior terá uma normal (0, 1, 0).
- A face inferior terá uma normal (0, −1, 0).
- A face lateral direita terá uma normal (1, 0, 0).
- A face lateral esquerda terá uma normal (-1, 0, 0).

Assim, cada vértice do cubo recebe a normal correspondente à sua face, garantindo a orientação correta para os cálculos de iluminação.

3.1.2.2. Coordenadas de Textura

As coordenadas de textura são calculadas com base na posição do vértice e no tamanho do cubo. A coordenada do vértice é ajustada somando o tamanho do lado do cubo e dividindo pela sua largura total, o que assegura que a textura seja mapeada de forma correta e uniforme entre os valores 0 e 1. Este método aplica-se a todas as faces do cubo, garantindo que a textura se repita de forma consistente ao longo da superfície.

```
// Front face texture coordinates
     float texA = (posA + cubeSide) / length;  // Normalize to [0,1]
     float texB = (posB + cubeSide) / length;
     float texNextA = (nextA + cubeSide) / length;
     float texNextB = (nextB + cubeSide) / length;
```

Listing 3: Coordendas Textura Cubo

3.1.3. Cone

Como já era de prever, a geração das coordenadas das normais tornou-se mais complexa à medida que as figuras geométricas aumentaram de complexidade, e no caso do cone isso confirmou-se rapidamente, tanto nas normais como nas texturas.

3.1.3.1. Coordenadas Normais

A base do cone foi a parte mais simples de calcular — como se encontra no plano **XZ**, todas as suas normais são iguais a $(0, -1, 0)$, apontando diretamente para baixo.

Já nas faces laterais, o processo é mais elaborado. Como o cone se vai afunilando até ao vértice, os vetores normais não são verticais nem horizontais, mas sim inclinados, apontando para fora e ligeiramente para cima. Para os calcular, foi utilizada uma aproximação geométrica: para cada ponto lateral, a normal resulta da combinação do vetor radial (em função do ângulo) com uma componente vertical proporcional ao raio na stack atual. O vetor resultante é depois normalizado.

Além disso, para otimizar, as normais dos pontos de cada slice foram pré-computadas na primeira stack e reaproveitadas para as restantes, garantindo consistência visual e eficiência computacional.

3.1.3.2. Coordenadas de Textura

As coordenadas de textura na base seguem uma abordagem radial: o centro da base recebe (0.5, 0.5), e os restantes pontos são mapeados em torno do centro com funções trigonométricas (seno e cosseno) para garantir uma distribuição circular das coordenadas entre [0,1].

Nas faces laterais, foi usado um mapeamento linear. O eixo horizontal das texturas (**u**) representa a posição angular em torno do cone (de 0 a 1 ao longo de um círculo completo), enquanto o eixo vertical (**v**) representa a altura ao longo do cone (também de 0 a 1, do fundo ao topo). Desta forma, a textura é esticada suavemente ao longo da superfície lateral, acompanhando a geometria do cone.

3.1.4. Cilindro

3.1.4.1. Coordenadas Normais

Tal como para o cone, também no cilindro as coordenadas normais das bases são triviais — se for a base superior, então as normais são `(0, 1, 0)`, e se for a base inferior, são `(0, -1, 0)`.

Para a face lateral, utilizámos uma abordagem simples e eficaz. Cada normal corresponde a um vetor radial, que aponta diretamente do centro do cilindro para o ponto na superfície lateral. Isto é feito calculando `(sin(θ), 0, cos(θ))`, onde `θ` é o ângulo atual do slice. Esta direção é perpendicular

à superfície curva do cilindro, o que permite uma iluminação correta e suave. Normalizamos este vetor para garantir que a sua magnitude é 1.

3.1.4.2. Coordenadas Textura

A textura usada para o cilindro junta, num único ficheiro, as imagens do topo, da base e da parte lateral. Isto exigiu um mapeamento cuidado das coordenadas para cada região da figura. As decisões sobre os valores a usar foram discutidas e aplicadas numa das aulas práticas, onde analisámos a estrutura da imagem e como alinhar corretamente cada secção do modelo 3D com a respetiva área da textura.

Para as tampas circulares (topo e base inferior), determinámos os centros das respetivas áreas na imagem:

- O **centro da base inferior** foi definido como `Point(0.8125, 0.1875)`
- O **centro do topo** ficou em `Point(0.4375, 0.1875)`
- O **raio** das zonas circulares foi considerado como `0.1875`, aplicado tanto ao eixo `x` como ao `y` através das funções `sin` e `cos`, garantindo uma distribuição uniforme dos pontos de textura ao redor do centro

Estes valores permitem posicionar corretamente os vértices da malha nas regiões circulares da textura que lhes correspondem.

Já para a lateral do cilindro, como esta forma pode ser "desenrolada" num retângulo, as coordenadas foram ajustadas nesse sentido:

- O eixo horizontal da textura (`u`) varia de `0.0` a `1.0` com base na slice atual
- O eixo vertical (`v`) vai de `0.375` (base da lateral) até `1.0` (topo da lateral)

Também estes valores foram definidos com base no exemplo prático apresentado na aula, que ajudou a perceber como cada porção da textura se encaixa visualmente nas diferentes partes do cilindro.

3.1.5. Esfera

O cálculo das normais na esfera segue uma lógica semelhante à usada nas faces laterais do cilindro, o que acabou por tornar o processo bastante direto.

3.1.5.1. Coordenadas Normais

Como o centro da esfera está na origem (0, 0, 0), calcular as normais para cada ponto torna-se simples: basta fazer o vetor entre o centro e o ponto atual e depois normalizar. Na prática, isso significa que a normal de cada ponto corresponde exatamente às suas coordenadas, mas normalizadas. Este método garante que todas as normais apontem para fora da superfície da esfera, como é esperado.

3.1.5.2. Coordenadas Textura

A projeção da textura na esfera baseia-se numa conversão de coordenadas esféricas para coordenadas 2D, algo que abordámos numa das aulas práticas ao discutir como aplicar texturas em superfícies curvas.

Cada ponto da esfera é definido por dois ângulos:

- `θ` (*theta*), que representa a inclinação vertical do ponto (de 0 a π)
- `φ` (*phi*), que representa a rotação horizontal em torno do eixo vertical (de 0 a 2π)

Com estes ângulos, conseguimos calcular as coordenadas de textura `(u, v)` da seguinte forma:

- `u = φ / (2π)` — corresponde à posição horizontal da textura
- `v = θ / π` — indica a posição vertical da textura

Desta forma, conseguimos mapear corretamente a imagem sobre a superfície da esfera, garantindo uma transição suave entre as diferentes faixas da textura.

3.1.6. Donut (Torus)

Para o **torus**, a abordagem foi muito semelhante à do cilindro e da esfera, usando o ponto central como referência.

3.1.6.1. Coordenadas Normais

As normais são calculadas de forma a apontar para o exterior da superfície do torus. Para tal, subtrai-se o vetor posição do centro do tubo (definido pelo círculo maior) à posição de cada vértice. Esse vetor resultante é perpendicular à superfície e representa corretamente a orientação da normal nesse ponto da geometria. Como cada face é composta por dois triângulos, são calculadas e armazenadas as normais correspondentes a cada um dos vértices desses triângulos.

3.1.6.2. Coordenadas de Textura

As coordenadas de textura (UVs) são mapeadas de forma proporcional à posição dos vértices nas divisões do torus. O valor `u` está associado ao índice do lado do tubo (**sides**) e `v` ao índice do anel (**rings**), ambos normalizados para o intervalo [0, 1]. Esta abordagem permite um mapeamento contínuo da textura ao longo da superfície, facilitando a aplicação de imagens ou padrões gráficos no torus de forma visualmente coerente e sem distorções abruptas.

3.1.7. Curvas *Bézier*

As curvas foram sem dúvida o maior desafio do grupo, uma vez que implicaram um grau de complexidade matemática e computacional superior ao das outras primitivas geométricas. Ainda assim, consideramos que obtivemos um resultado minimamente satisfatório neste quesito.

3.1.7.1. Coordenadas Normais

As normais em cada ponto da superfície *Bézier* são calculadas com base nas derivadas parciais da superfície em relação a `u` e `v`.

A normal é depois obtida através do produto vetorial `dv × du`, e normalizada. Este método permite obter normais suaves, o que melhora significativamente a iluminação.

3.1.7.2. Coordenadas de Textura

Para o mapeamento de texturas, foi feita uma correspondência direta entre os parâmetros `u` e `v` da superfície *Bézier* e as coordenadas da textura. Como `u` e `v` variam entre 0 e 1, são utilizados diretamente como coordenadas de textura `Point2D(u, v)`.

4. Iluminação

Um dos requisitos desta fase final foi permitir que as cenas tivessem fontes de luz com as suas características típicas, luz direcional, point light, spotlight e também que cada modelo tivesse propriedades de material associadas.

Agora, cada modelo tem componentes de material armazenadas: ambiente, difusa, especular e emissão, além do `shininess`, que define a intensidade do brilho especular. Estas propriedades estão guardadas na estrutura de dados de cada modelo, o que permite configurar o material antes de o desenhar.

Durante a renderização, usamos funções como `glMaterialfv` e `glMaterialf` para definir as propriedades visuais do modelo. As luzes são configuradas com os seus próprios parâmetros — como posição, direção, cutoff e suportam os diferentes tipos exigidos. Assim, conseguimos que cada objeto interaja com a luz de forma realista, dependendo do tipo de fonte e do seu material.

5. Configuração do Modelo

Para garantir que o modelo seja corretamente renderizado, agora incluímos as normais e as coordenadas das texturas além das posições dos vértices. Este processo envolve transferir essas informações para a GPU, de modo a otimizar a renderização dos modelos 3D.

Primeiro, o código começa por extrair os dados de posições, normais e coordenadas de textura a partir do **Vertex Buffer Object** (VBO). Estes dados são essenciais para o processo de renderização, pois permitem calcular a iluminação, as sombras e a aplicação de texturas corretamente.

A seguir, são criados os buffers para armazenar cada um desses dados na GPU:

1. **Buffer de Posições de Vértices**: A função `glGenBuffers` gera um identificador único para o buffer de posições e a função `glBindBuffer` vincula o buffer a uma operação de buffer de dados. O `glBufferData` é utilizado para transferir os dados de posição dos vértices para a GPU, utilizando a flag `GL_STATIC_DRAW`, que indica que os dados não serão frequentemente modificados.
2. **Buffer de Normais**: De forma semelhante, as normais são extraídas e armazenadas num buffer específico para as normais dos vértices. Essas informações são fundamentais para o cálculo de luz e sombreamento durante o processo de renderização. As normais ajudam a determinar a direção em que cada superfície está orientada em relação à fonte de luz, permitindo a iluminação correta.
3. **Buffer de Coordenadas de Textura**: As coordenadas de textura são usadas para mapear as texturas 2D aos modelos 3D. Após a extração das coordenadas de textura, os dados são armazenados em outro buffer para que possam ser usados durante o processo de mapeamento de textura na GPU.
4. **Buffer de Índices**: Finalmente, o código cria e configura um buffer de índices. Os índices são utilizados para indicar a ordem em que os vértices são desenhados, permitindo a criação de superfícies e triângulos a partir de vértices individuais. Este buffer também é configurado com a flag `GL_STATIC_DRAW`, já que, em geral, os índices não mudam com frequência durante a execução do programa.

5.1. Texturas

O método utilizado para carregar as texturas segue a abordagem vista nas aulas práticas. A principal tarefa deste processo é carregar a imagem de uma textura e transferi-la para a GPU, onde será utilizada na renderização do modelo 3D.

O processo começa com o uso da biblioteca **stb_image** para carregar a imagem a partir de um arquivo, recuperando as suas dimensões e o número de canais (por exemplo, RGBA). Se a imagem for carregada com sucesso, o próximo passo é criar uma textura no OpenGL e configurar os parâmetros adequados, como repetição (wrap) e o filtro de interpolação para diferentes distâncias (com mipmapping para melhorar a qualidade visual). Após a configuração, a textura é carregada na GPU, e a memória da imagem original é liberada, já que os dados já estão na GPU.

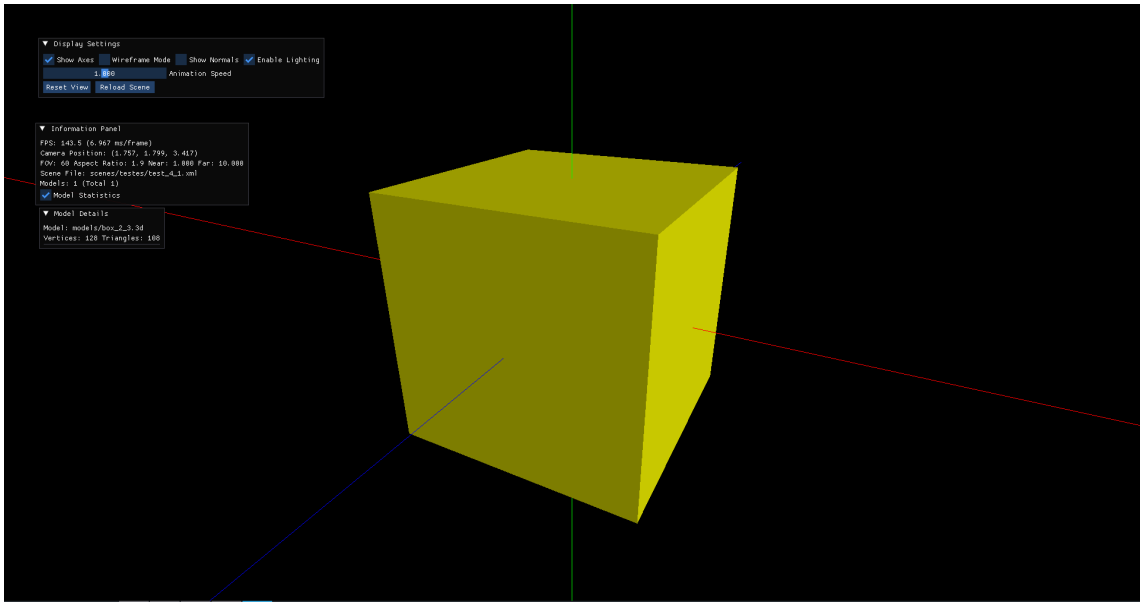
A seguir, o código mostra como tudo isso é feito:

```
// Generate and bind texture
glGenTextures(1, &this->_texture_id);
glBindTexture(GL_TEXTURE_2D, this->_texture_id);

// Set texture parameters
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

// Use mipmapping for better quality at different distances
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

// Ensure proper alignment when uploading
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

// Upload texture data to GPU
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA,
             GL_UNSIGNED_BYTE, imageData);
glGenerateMipmap(GL_TEXTURE_2D);

// Unbind the texture
glBindTexture(GL_TEXTURE_2D, 0);

// Free image data after uploading it
stbi_image_free(imageData);
```

Listing 4: Texturas

6. Interface Interativa

O nosso grupo desafiou-se a criar uma interface intuitiva, que permitisse ao utilizador interagir e alterar facilmente as opções do programa. Para isso, decidimos usar **ImGUI**, uma ferramenta que achamos que se encaixa bem com o estilo do projeto, oferecendo uma solução simples e eficaz. Para implementar esta interface, guiámo-nos por este tutorial, bastante prático e de fácil compreensão: ImGui + GLFW Tutorial - Install & Basics.



7. Testes Realizados

Após as alterações necessárias no ***engine*** e no ***generator***, procedemos à execução de uma série de testes disponibilizados pelos docentes com o objetivo de validar a correta interpretação e renderização dos modelos. Estes testes encontram-se disponíveis na pasta `scenes/testes`.

Em seguida, apresentam-se os resultados obtidos:

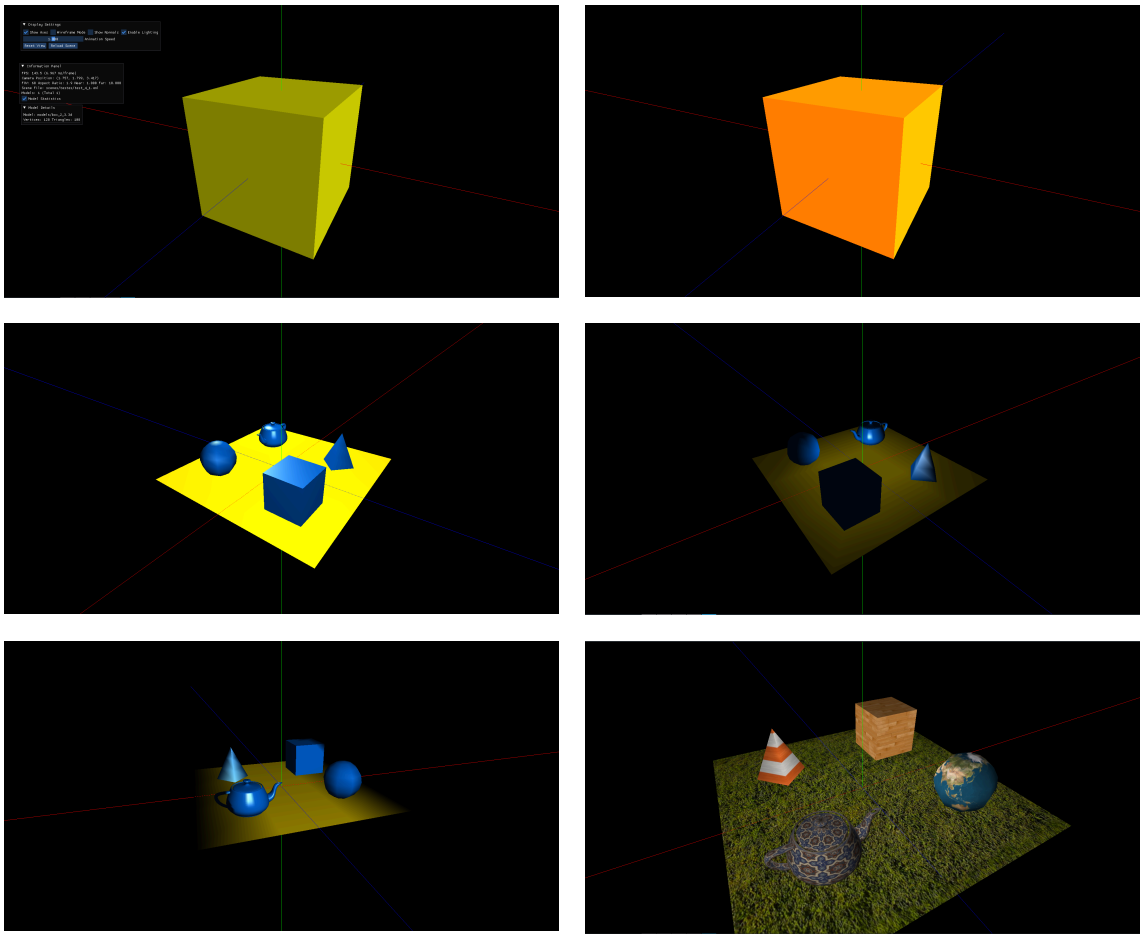


Figura 1: Resultados dos testes disponibilizados pelos docentes

8. Sistema Solar

Um dos requisitos para esta terceira fase foi dar um *upgrade* à cena referente ao sistema solar apresentada na fase anterior. Nesta fase, foi necessário escalar todo o Sistema Solar para possibilitar a aplicação de texturas nos planetas, a adição de luzes no cenário e a definição das características dos materiais de cada planeta e asteroide. Como resultado, a cena final apresenta um sistema solar mais realista, com texturas detalhadas, iluminação dinâmica e materiais específicos para cada corpo celeste.

8.1. Resultados Obtidos

Após a incorporação de toda a informação adicional no ficheiro `scenes/solar_system.xml`, foi possível observar uma representação significativamente mais realista e dinâmica do sistema solar.

Como demonstração do resultado final, incluímos um vídeo disponível em `scenes/videos`, onde é possível visualizar o movimento completo do sistema solar em tempo real.

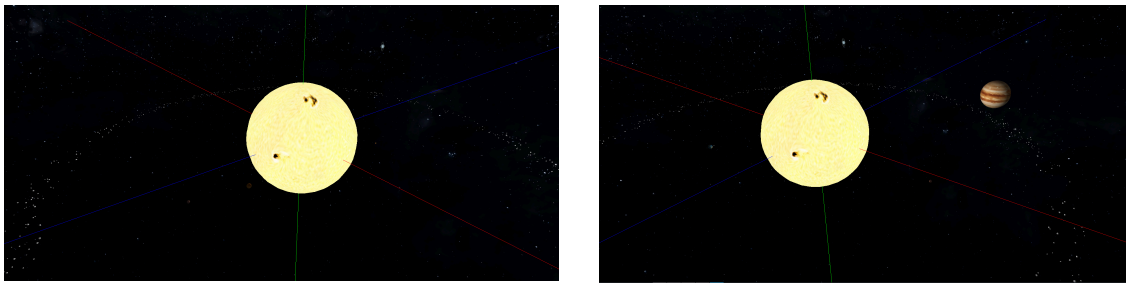


Figura 2: Sistema solar

9. Conclusões e Trabalho Futuro

O desenvolvimento desta quarta fase revelou-se bastante positivo, e o grupo conseguiu concluir com sucesso tudo o que tinha planeado.

Nesta fase final, consideramos que atingimos um resultado satisfatório, com a cena final a refletir o esforço colocado no projeto ao longo do tempo. No entanto, gostaríamos de ter tido mais tempo para explorar a implementação de **Frustum Culling**, de forma a otimizar ainda mais o desempenho do sistema,mas infelizmente não foi possível devido às limitações temporais.

Bibliografia

- [1] «vcpkg - Open source C/C++ dependency manager from Microsoft». Disponível em: https://vcpkg.io/en/
- [2] «CMake». Disponível em: https://cmake.org/
- [3] «OpenGL - The Industry Standard for High Performance Graphics». Disponível em: https://www.opengl.org/
- [4] «GitHub - Build software better, together». Disponível em: https://github.com/
- [5] Song Ho Ahn, «OpenGL Tutorials». Disponível em: https://www.songho.ca/opengl/
- [6] «Eyes on the Solar System». NASA. Disponível em: https://eyes.nasa.gov/apps/solar-system/#/home
- [7] «Devstronomy - Planetary Data Reference». Disponível em: https://devstronomy.martinovo.net/
- [8] «Bezier Interpolation». Medium. Disponível em: https://medium.com/@adrian_cooney/bezier-interpolation-13b68563313a
- [9] «Catmull-Rom Spline Algorithm». MVPs. Disponível em: https://www.mvps.org/directx/articles/catmull/
- [10] «Catmull-Rom Splines in Plain English». Andrew Hung Blog. Disponível em: https://andrewhungblog.wordpress.com/2017/03/03/catmull-rom-splines-in-plain-english/

Definição de Ponto

A estrutura `Ponto` representa uma coordenada num espaço tridimensional e é definida como:

```
typedef struct Point {
  float x;
  float y;
  float z;

  Point(float x_val = 0.0f, float y_val = 0.0f, float z_val = 0.0f)
      : x(x_val), y(y_val), z(z_val) {}

} Point;
```

ModelGroup

```
ModelGroup::ModelGroup(std::vector<Model> models,
                       std::vector<ModelGroup> modelSubGroup,
                       glm::mat4 transformations,
                       std::vector<Rotations> rotations,
                       std::vector<Translations> translations,
                       std::vector<TimeTransform> orderOfTransformations) {
  this->models = models;
  this->subgroups = modelGroup;
  this->transformations = transformations;
  this->rotations = rotations;
  this->translates = translations;
  this->order = orderOfTransformations;
}
```

Model

```
Model::Model(std::string modelFileName, std::vector<Point> controlPoints) {
  this->modelFileName = modelFileName;
  this->modelIdentifier = counter;
  this->vertexBufferObject = generateVBOs(controlPoints);
  this->indexBufferObject = generateIBOs(controlPoints, this->vertexBufferObject);
  this->isInitialized = false;
  this->controlPoints = controlPoints;
  counter++;
}
```