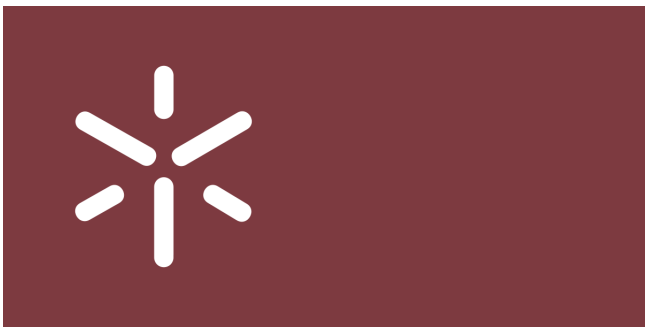


Universidade do Minho

Escola de Engenharia

Licenciatura em Engenharia Informática

Projeto da U.C. Computação Gráfica

4ª Fase

Ano letivo 2023/2024

Realizado por:

A91672 - Luís Ferreira
A93258 - Bernardo Lima
A100543 - João Pastore
A100554 - David Teixeira

Conteúdo

Lista de Figuras

1 Introdução

Nesta fase final do trabalho prático da disciplina de Computação Gráfica, incorporamos novas funcionalidades às duas aplicações principais: o *Generator* e o *Engine*, além de realizar uma remodelação e evolução do *Parser*.

As melhorias implementadas incluem a capacidade do *Generator* de calcular as normais e as coordenadas de textura para cada vértice das primitivas gráficas. Adicionalmente, o *Engine* agora utiliza esses dados gerados pelo *Generator* para aplicar texturas e iluminação às primitivas exibidas nas animações.

Neste relatório, apresentamos uma análise detalhada das decisões e metodologias adotadas, que possibilitaram a implementação eficiente das funcionalidades propostas.

2 Parser

2.1 Introdução

Em comparação com a fase anterior do nosso projeto, o novo *parser* apresenta uma eficiência significativamente maior e é mais leve, com diversas `structs` removidas (`struct LookAt`, `struct Up`, `struct Projection`), substituídas pela utilização de uma única `struct Ponto`. Adicionalmente, foi implementado o *parsing* de luzes, cores e texturas utilizando as novas `structs` mencionadas a seguir.

A principal motivação para a reconstrução do *parser* foi a resolução de problemas relacionados a *groups* e à gestão de memória. A nova implementação é mais robusta e escalável, proporcionando melhorias no desempenho geral e na manutenção do código.

2.2 Descrição das Estruturas

2.2.1 Window

A `struct Window` armazena as dimensões da janela de visualização, crucial para a configuração inicial do ambiente gráfico.

```
struct Window {
    int width = 0;
    int height = 0;
};
```

2.2.2 Point

A `struct Point` representa um ponto no espaço tridimensional, utilizado em várias outras estruturas para definir posições e vetores.

```
struct Point {
    float x = 0.0f;
    float y = 0.0f;
    float z = 0.0f;
};
```

2.2.3 Camera

A `struct Camera` armazena as configurações da câmera, incluindo posição, *lookAt*, *up* e parâmetros de projeção. Estas informações são essenciais para a definição da perspectiva e orientação da visualização da cena.

```
struct Camera {
    Point position;
    Point lookAt;
    Point up;
    Point projection;
};
```

2.2.4 Transform

A `struct Transform` representa uma transformação que pode ser aplicada a um objeto. As transformações podem ser de três tipos principais: translação, rotação e escala. Além disso, a estrutura suporta animações através da definição de uma sequência de pontos e um tempo de duração.

```
struct Transform {
    char type;
    Point point;
    float angle = 0.0f;
    float time = 0.0f;
    bool align;
    std::vector<Point> points;
};
```

2.2.5 Color

A `struct Color` armazena as informações de cor para materiais, incluindo valores RGB e um valor adicional para certas propriedades como *shininess*.

```
struct Color {
    std::string type;
    float r = 0.0f;
    float g = 0.0f;
    float b = 0.0f;
    float value = 0.f;
};
```

2.2.6 Model

A `struct Model` representa um modelo 3D, incluindo o nome do ficheiro do modelo e o nome do ficheiro de textura associado, bem como uma lista de cores. Esta estrutura facilita a integração de modelos complexos com texturas e materiais diversificados.

```
struct Model {
    std::string fileName;
    std::string textureName;
    std::vector<Color> colors;
};
```

2.2.7 Group

A `struct Group` representa um grupo de transformações e modelos. Pode conter subgrupos, permitindo uma estrutura hierárquica. Esta hierarquia é crucial para a organização de cenas complexas, permitindo a aplicação de transformações de forma recursiva.

```
struct Group {
    std::vector<Transform> transforms;
    std::vector<Model> modelFiles;
    std::vector<Group> children;
};
```

2.2.8 Light

A `struct Light` armazena as informações sobre as luzes na cena, incluindo tipo, posição, direção e *cutoff*. A correta definição das luzes é essencial para a renderização realista da cena, influenciando sombras, reflexos e a iluminação geral.

```
struct Light {
    std::string type;
    std::vector<float> position;
    std::vector<float> direction;
    float cutoff;
};
```

2.2.9 Parser

A `struct Parser` é a estrutura principal que contém todas as configurações da janela, câmera, grupo raiz de objetos e luzes. Esta estrutura centraliza todas as informações necessárias para a renderização da cena, facilitando a sua gestão e manipulação.

```
struct Parser {
    Window window;
    Camera camera;
    Group rootNode;
    std::vector<Light> lights;
};
```

2.3 Motivações para a Reconstrução

A decisão de reconstruir o *parser* do zero foi tomada devido a vários problemas críticos identificados nesta fase:

- **Problemas com *groups***: A estrutura anterior do parser apresentava dificuldades em lidar corretamente com grupos de transformações e modelos (nos testes da fase 4), resultando em erros na renderização e na organização hierárquica dos objetos.
- **Problemas de Memória**: Foram observados problemas significativos de gestão de memória, resultando no uso ineficiente dos recursos disponíveis.

2.4 Conclusão

Através da simplificação e otimização das estruturas do *parser*, foi possível resolver problemas de organização e eficiência de memória, facilitando a leitura e manipulação dos dados do ficheiro XML. A nova abordagem não só melhora o desempenho, como também a escalabilidade do *parser*, tornando-o mais robusto e adequado para projetos futuros. A melhoria na gestão de memória e na hierarquização de *groups* permite um processamento correto das informações cruciais do ficheiro XML para a renderização das cenas.

3 Generator

Cada uma das figuras geradas pelo *Generator* precisou ser redefinida, devido à necessidade de atribuir coordenadas de vetores normais (usados para aplicar a iluminação) e coordenadas de textura.

3.1 Esfera (Sphere)

3.1.1 Geração de Coordenadas

Para gerar uma esfera, utilizamos as variáveis `radius` (raio), `slices` (número de divisões ao redor do eixo vertical) e `stacks` (número de divisões ao longo do eixo vertical). O código divide a esfera em quadrados, cada um subdividido em dois triângulos. Para cada vértice, são calculadas as coordenadas cartesianas (x, y, z) usando as funções `sinf` e `cosf` baseadas em ângulos ϕ e θ.

3.1.2 Cálculo das Normais

As normais são calculadas a partir das coordenadas dos vértices. Para cada vértice (x, y, z), a normal é simplesmente a direção radial normalizada:

$$\text{normal} = \left(\frac{x}{r}, \frac{y}{r}, \frac{z}{r}\right)$$

3.1.3 Cálculo das Coordenadas de Textura

As coordenadas de textura são mapeadas usando os ângulos θ e ϕ:

$$u = \frac{\theta}{2\pi}, \quad v = \frac{\phi}{\pi}$$

3.2 Cubo (Box)

3.2.1 Geração de Coordenadas

Para um cubo, `size` define o comprimento da aresta e `divisions` define o número de subdivisões por aresta. Cada face do cubo é subdividida numa grade de pequenos quadrados, cada um dividido em dois triângulos.

3.2.2 Cálculo das Normais

As normais são constantes para cada face do cubo, pois cada face é plana e perpendicular aos eixos principais:

- Topo: $(0, 1, 0)$
- Base: $(0, -1, 0)$
- Frente: $(0, 0, 1)$
- Trás: $(0, 0, -1)$
- Direita: $(1, 0, 0)$
- Esquerda: $(-1, 0, 0)$

3.2.3 Cálculo das Coordenadas de Textura

As coordenadas de textura para cada vértice de um quadrado são interpoladas linearmente de acordo com a posição do vértice dentro da face:

$$u = \frac{\text{posição horizontal}}{\text{comprimento da face}}, \quad v = \frac{\text{posição vertical}}{\text{comprimento da face}}$$

3.3 Plano (Plane)

3.3.1 Geração de Coordenadas

Para um plano, `size` define o comprimento da aresta e `divisions` define o número de subdivisões. O plano é subdividido numa grade de quadrados, cada um dividido em dois triângulos.

3.3.2 Cálculo das Normais

A normal para todos os vértices é constante e aponta para cima, pois é um plano horizontal:

$$\text{normal} = (0, 1, 0)$$

3.3.3 Cálculo das Coordenadas de Textura

As coordenadas de textura são interpoladas de acordo com a posição do vértice dentro do plano:

$$u = \frac{\text{posição horizontal}}{\text{comprimento do plano}}, \quad v = \frac{\text{posição vertical}}{\text{comprimento do plano}}$$

3.4 Cone

3.4.1 Geração de Coordenadas

Para um cone, `radius` define o raio da base, `height` define a altura, `slices` define o número de subdivisões da circunferência da base e `stacks` define o número de subdivisões ao longo da altura. O cone é composto por triângulos que formam os lados e a base.

3.4.2 Cálculo das Normais

As normais para a base são simplesmente $(0, -1, 0)$. Para os lados, as normais são calculadas aproximando a superfície do cone utilizando a média dos ângulos.

3.4.3 Cálculo das Coordenadas de Textura

As coordenadas de textura para a base são mapeadas radialmente, enquanto que para os lados são interpoladas linearmente ao longo da altura e ao redor da circunferência.

3.5 Anel (Ring)

3.5.1 Geração de Coordenadas

Para um anel, `ir` define o raio interno, `er` define o raio externo e `slices` define o número de subdivisões ao redor da circunferência.

3.5.2 Cálculo das Normais

As normais para o anel são constantes para cada face, pois cada face é plana:

- Superior e Inferior: $(0, \pm 1, 0)$
- Internas e Externas: Perpendiculares ao raio.

3.5.3 Cálculo das Coordenadas de Textura

As coordenadas de textura são mapeadas radialmente para as faces superior e inferior e linearmente para as faces internas e externas.

3.6 Superfície de Bézier

3.6.1 Geração de Coordenadas

As superfícies de Bézier são geradas utilizando pontos de controlo. O ficheiro de *input* contém os *patches* e os pontos de controlo. A superfície é subdividida de acordo com o parâmetro de tesselação.

3.6.2 Cálculo das Normais

As normais são calculadas como o produto vetorial das derivadas parciais da superfície em relação aos parâmetros u e v.

3.6.3 Cálculo das Coordenadas de Textura

As coordenadas de textura são mapeadas diretamente dos parâmetros u e v. Estes parâmetros variam de 0 a 1 ao longo das direções horizontal e vertical da superfície:

$$u = \text{parâmetro } u, \quad v = \text{parâmetro } v$$

4 Engine

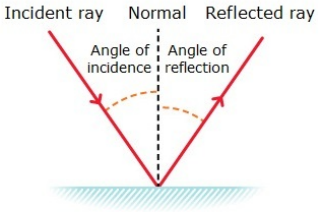
4.1 Iluminação

Para ser possível incorporar a iluminação, requisito desta fase do projeto, foi necessário familiarizar-nos com diversos contextos que iremos apresentar, um dos quais sendo o **Modelo de Reflexão de Phong**, modelo utilizado no *OpenGL*.

Também foram considerados três tipos de luz: luzes posicionais, direccionais e focos (*spotlights*). Para as luzes posicionais, é apenas necessário definir a posição. Para as luzes direccionais, fornecemos o vector de direcção correspondente. Já para os focos de luz, exige-se tanto o vector de direcção quanto a posição da luz juntamente com o parâmetro *cutoff*.

4.2 Normais

De forma a ser possível aplicar luz a objetos, é importante compreender como os objetos se comportam com a luz. Com base no modelo de *Phong* é possível compreender que para aplicar uma luz num objeto é necessário saber o ângulo que a luz faz com cada face desse.



No caso de uma relexão num espelho, a normal é contabilizada de uma maneira muito simples, onde o ângulo que a luz incide sobre o espelho será semelhante ao que é refletido, porque um espelho é uma superfície plana, a duas dimensões e considerado normalmente sem grande **textura**.

No contexto do *openGL* é importante referir que a normal tem necessariamente de ser representada como um **vetor unitário** e como tal, todas as normais são *normalizadas*. Isto é possível graças à utilização de um método `normalizePonto()`, que é invocado sempre que uma normal de um ficheiro *.3d* é detetada, dividindo as coordenadas pela sua **magnitude**.

```
void normalizePonto(ponto p) {
    float x = getX(p), y = getY(p), z = getZ(p);
    float l = sqrt(x*x + y*y + z*z;
    p->x /= l;
    p->y /= l;
    p->z /= l;
};
```

4.3 Material

Para representar a maneira como a luz se comporta em diferentes objetos foi utilizada em combinação com o **modelo de reflexão de Phong**, a noção de **brilho**.

4.3.1 Modelo de Reflexão de Phong

O modelo de reflexão de Phong foca na aplicação da luz em 3 formas. A **luz ambiente**, a **reflexão especular** e a **reflexão difusa**.

4.3.2 Luz Ambiente

A luz no mundo real é resultante de diferentes fontes que refletem em tudo que nos rodeia. De forma a simular este comportamento implementamos o **Ambient Lighting**, com recurso à implementação inicial de uma luz *branca*, usando o *GL_LIGHT_MODEL_AMBIENT* através da função `glLightModelfv()`.

4.3.3 Reflexão Especular

Para além da luz ambiente, é normal que algumas reflexões de luz se destaquem em relaxão a outras, seja pela proximidade da fonte da luz.

Figura 1: Reflexão Especular

4.3.4 Reflexão Difusa

Como nem todos os objetos têm uma textura "lisa", é importante perceber como a luz se comporta com estas superfícies irregulares.

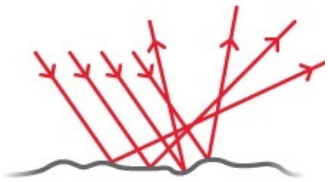


Figura 2: Reflexão Difusa

4.3.5 Material

Devido a cada material ter um comportamento diferente às propriedades da luz recentemente referidas, é necessário no contexto do *openGL* associar o **material** de que elas são feitas. Para isso cada figura é associada um conjunto de propriedades ao seu material usando o método `glMaterialfv()` para a difusão (*GL_DIFFUSE*), luz ambiente (*GL_AMBIENT*), especulação (*GL_SPECULAR*).

Para além destes, é necessário contabilizar se um objeto é emissor de luz, como o Sol na demo final, adicionando a propriedade emissora (*GL_EMISSION*). Estas propriedades são passadas ao método juntamente com um *vector* que possui as **cores** associadas à luz em formato *RGB*. Para além destes 4 tipos de materiais existe também outro, o nível de brilho, (*GL_SHININESS*), que não possui cor.

4.4 Texturas

No processo de importação de figuras, começamos por carregar todas as coordenadas das figuras, incluindo as coordenadas das normais e os pontos de textura, para os **VBOs** (*Vertex Buffer Objects*). Se uma figura tiver uma textura associada, carregamos a mesma utilizando a função `loadTexture()`[1].

Ao desenhar a figura, fazemos o *bind* do *buffer* das normais e utilizamos a função `glNormalPointer()`. Para as texturas, realizamos novamente o *bind* do *buffer* apropriado e utilizamos a função `glTexCoordPointer()`.

Este procedimento garante que todas as informações necessárias para a renderização das figuras, incluindo texturas e normais, estejam devidamente carregadas e vinculadas aos *buffers* corretos, permitindo uma renderização eficiente e precisa das cenas.

[1]Responsável pela formação do *path* da imagem, inicialização e ligação, carregamento, obtenção de propriedades (largura e altura), conversão para RGBA, obtenção de dados, criação e ligação da textura, definição de parâmetros, envio para GPU e geração de *mipmaps*.

5 Sistema Solar

A seguir, é apresentada a imagem da versão final do sistema solar:

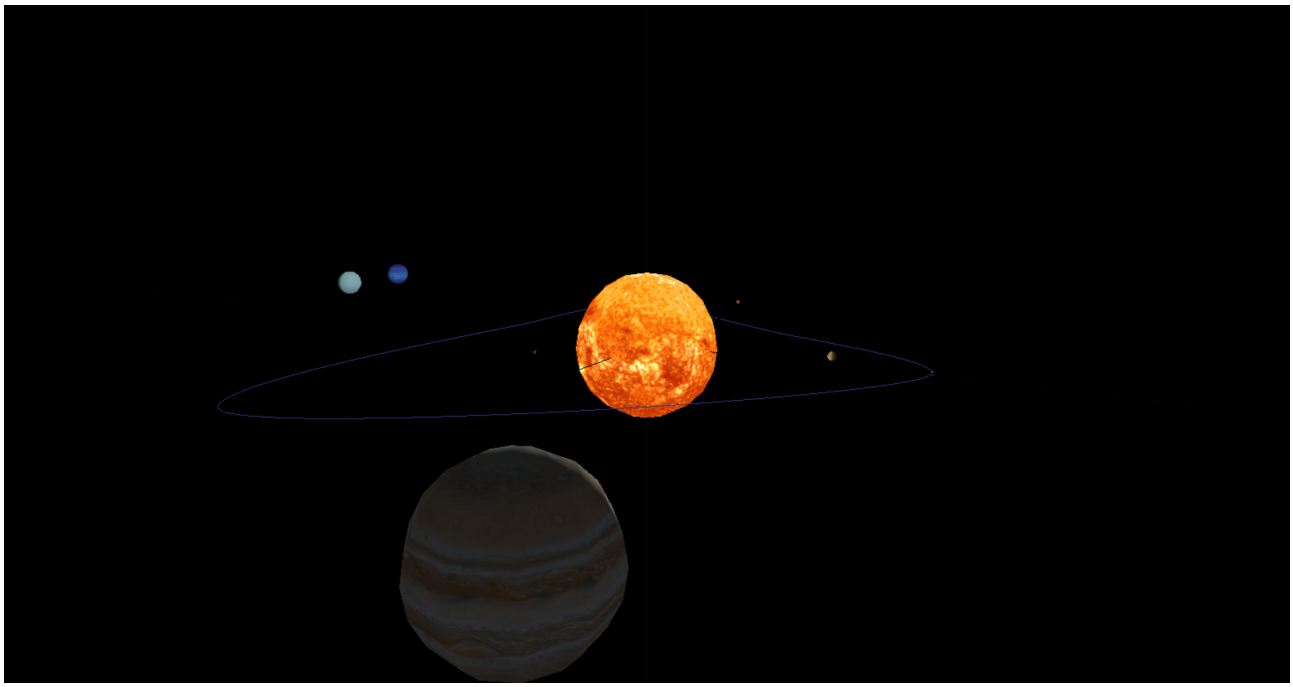


Figura 3: Representação do Sistema Solar

6 Conclusão

Durante a execução da quarta e última fase deste projeto, conseguimos consolidar os conceitos de textura e iluminação abordados nas aulas. Para isso, realizamos o cálculo das normais para todos os modelos previamente criados e determinamos as coordenadas de textura para cada um deles.

Estamos satisfeitos com o resultado alcançado, pois conseguimos implementar todas as funcionalidades solicitadas para esta fase.

Por fim, acreditamos que, ao longo desta fase, em conjunto com o trabalho realizado nas fases anteriores, reunimos todos os elementos necessários para apresentar um projeto robusto, consolidando de forma eficaz os conhecimentos adquiridos nas aulas de Computação Gráfica.