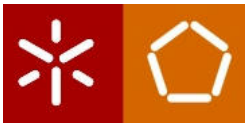

Universidade do Minho
Departamento de Informática

Computação Gráfica

Trabalho Prático - Fase 4

Grupo 5

João Coelho - A100596

José Rodrigues - A100692

Duarte Araújo - A100750

2024-05-26

Índice

Lista de Figuras

1. Indrodução

Nesta fase do projeto o foco principal recaiu sobre a implementação de normais e coordenadas de textura para cada vértice dos modelos 3D, de forma a que fosse possível a utilização de iluminação e texturas nas nossas cenas, com o objetivo de conseguir criar cenários mais completos e realistas.

Ao longo deste relatório, detalharemos não apenas os aspectos técnicos da implementação, mas também os desafios enfrentados durante o processo de desenvolvimento. Além disso, exploraremos os resultados obtidos e discutiremos como essas melhorias contribuem para o amadurecimento do nosso motor gráfico.

2. Desenvolvimento

Primeiramente, abordamos a necessidade de calcular normais para cada vértice dos nossos modelos 3D. As normais são vetores perpendiculares à superfície dos modelos e são cruciais para o correto cálculo da iluminação, permitindo que as superfícies respondam realisticamente às fontes de luz. A geração precisa de normais foi um passo fundamental para garantir que as texturas e os efeitos de luz fossem aplicados de maneira coerente e visualmente atraente.

De seguida, implementamos coordenadas de textura para os modelos. As coordenadas de textura determinam como uma imagem (textura) é mapeada sobre a superfície de um modelo. Este mapeamento é vital para que as texturas sejam aplicadas corretamente, evitando distorções e garantindo uma aparência natural. O desenvolvimento de algoritmos para gerar coordenadas de textura adequadas para diferentes tipos de primitivas (como esferas, caixas e cones) foi um desafio significativo.

Com a geração de normais e coordenadas de textura concluída, passamos a integrar a iluminação e a texturização no motor gráfico. Implementamos diferentes tipos de luzes, incluindo luzes pontuais, direcionais e spotlights. Cada tipo de luz requer configurações específicas e afeta as superfícies de maneiras distintas, adicionando profundidade e realismo às cenas.

Para suportar essas novas funcionalidades, fizemos ajustes no parser XML do motor gráfico, permitindo que os ficheiros de configuração especifiquem parâmetros de iluminação e texturização. Isso inclui a definição de propriedades de material, como componentes de cor difusa, especular, emissiva e ambiente, além da aplicação de texturas através de ficheiros de imagem.

Finalmente, a demonstração prática dessas funcionalidades foi realizada na criação de uma cena dinâmica do sistema solar, onde os planetas são texturizados e iluminados de forma realista, e um cometa segue uma trajetória definida por uma curva Catmull-Rom, com a superfície do cometa sendo modelada usando patches de Bezier.

Nas subseções a seguir, detalharemos os processos de implementação de cada uma dessas funcionalidades, discutindo os desafios enfrentados e as soluções adotadas para superá-los.

2.1. Implementação de Normais e Coordenadas de Textura

Como já foi referido anteriormente, um dos principais objetivos desta fase resumia-se na implementação de normais e coordenadas de textura para cada vértice dos modelos desenvolvidos na primeira fase. Assim sendo, nas próximas subseções, será explicado como foram calculadas as normais e as texCoords para cada uma das primitivas.

2.1.1. Plano

No caso do plano, as normais e as coordenadas de textura foram calculadas da seguinte forma:

2.1.1.1. Normais

Num plano horizontal situado no eixo XZ, todas as normais são vetores perpendiculares à superfície do plano, apontando diretamente para cima. Isso ocorre porque a superfície do plano é, tal como indica o nome, plana e não possui variações na direção normal ao longo da sua extensão. Portanto, para qualquer ponto no plano, a normal é a mesma, já que a direção perpendicular à superfície não muda.

Assim, podemos definir a normal do plano como: (0.0f, 1.0f, 0.0f) Indicando que a direção é positiva ao longo do eixo Y. Esta normal é constante para todos os vértices do plano, pois a superfície do plano é uniforme e não há curvatura ou inclinação, tal como já foi referido anteriormente. Assim sendo, esta primitiva apresenta o cálculo das normais mais simples e direto daquelas que serão apresentadas neste relatório.

2.1.1.2. Coordenadas de Textura

O plano, como uma superfície bidimensional, é definido por uma grade de vértices distribuídos ao longo dos seus eixos. Para cada vértice deste plano, é necessário determinar as coordenadas de textura (s, t) que correspondem à sua posição relativa na grade, garantindo assim que a textura seja aplicada de maneira adequada e uniforme.

A abordagem adotada para o cálculo das coordenadas de textura segue uma lógica simples e direta:

1. **Divisão do Plano numa Grade:** O plano é subdividido numa grade de células, processo que já foi utilizado no cálculo das coordenadas de posição do plano nas fases anteriores, onde cada célula contém quatro vértices que formam dois triângulos. Essa grade é definida com base no número de divisões especificado para o plano.
2. **Cálculo das Coordenadas (s, t):** Para cada vértice do plano, as coordenadas de textura são calculadas com base na sua posição na grade. Isso é alcançado dividindo-se as coordenadas do vértice ao longo dos eixos X e Z pelo número total de divisões ao longo desses eixos.

```
glm::vec2 t1(j / ((float)divisions), i / ((float)divisions));
glm::vec2 t2(j / ((float)divisions), (i+1) / ((float)divisions));
glm::vec2 t3((j+1) / ((float)divisions), i / ((float)divisions));
glm::vec2 t4((j+1) / ((float)divisions), (i+1) / ((float)divisions));
```

De seguida, estas coordenadas precisam de ser inseridas no ficheiro com a ordem correta que, neste caso, tem de ser equivalente à ordem em que são colocadas as coordenadas de posição (vértices).

2.1.2. Box

De seguida, a implementação das normais e coordenadas de textura na Box já demonstrou necessitar de um processo mais complexo do que o Plano.

2.1.2.1. Normais

Para cada face da caixa, as normais são calculadas de acordo com a orientação da face. Por exemplo, para a face frontal, todas as normais são definidas como (0.0f, 0.0f, 1.0f), indicando que estão apontando para fora da face na direção do eixo Z positivo. O mesmo princípio se aplica às outras faces da caixa, onde as normais são ajustadas de acordo com a orientação de cada face.

As normais são consistentes para todos os vértices de cada face, pois todas as faces da caixa são planas e não têm variações significativas na direção normal ao longo da sua extensão.

Assim sendo, as normais utilizadas para as diferentes faces da caixa foram as seguintes:

```
glm::vec3 normal_front(0.0f, 0.0f, 1.0f);
glm::vec3 normal_back(0.0f, 0.0f, -1.0f);
glm::vec3 normal_top(0.0f, 1.0f, 0.0f);
glm::vec3 normal_bottom(0.0f, -1.0f, 0.0f);
glm::vec3 normal_left(-1.0f, 0.0f, 0.0f);
glm::vec3 normal_right(1.0f, 0.0f, 0.0f);
```

Como é possível observar, à semelhança do plano, o cálculo das normais para esta primitiva é bastante simples e direto.

2.1.2.2. Coordenadas de Textura

Para garantir que a textura seja aplicada uniformemente em toda a superfície da caixa, seguimos um processo semelhante ao utilizado para o plano, porém, precisamos repeti-lo para cada uma das seis faces da caixa. Isso ocorre porque a caixa possui múltiplas faces, e cada face precisa de ter as suas próprias coordenadas de textura calculadas independentemente das outras.

Assim como no plano, dividimos a superfície de cada face numa grade de células e atribuímos coordenadas de textura para cada vértice com base na sua posição relativa nessa grade.

À semelhança do plano, as coordenadas de textura necessitam de ser inseridas no ficheiro *.3d* de forma a garantir a consistência com a ordem dos vértices.

2.1.3. Cone

No caso do cone, já foram encontradas algumas dificuldades durante o cálculo das normais e das coordenadas de textura, devido aos métodos mais complexos que precisam de ser utilizados para obter os resultados desejados.

2.1.3.1. Normais

Para cada face do cone, as normais são calculadas de maneira diferente para garantir uma representação precisa da curvatura da superfície. Vamos examinar como as normais são calculadas para cada parte:

1. **Base do Cone:**
   - As normais para a base do cone são todas apontadas para baixo, ao longo do eixo negativo Y, já que esta face é plana e a sua orientação é constante.
   - Cada vértice na base do cone recebe a mesma normal, apontando para baixo.

```
glm::vec3 normal_bottom(0.0f, -1.0f, 0.0f);
```

2. **Lados do Cone:**

Para simplificar a explicação do cálculo das normais para as laterias do cone, vamos reduzir o problema para duas dimensões.

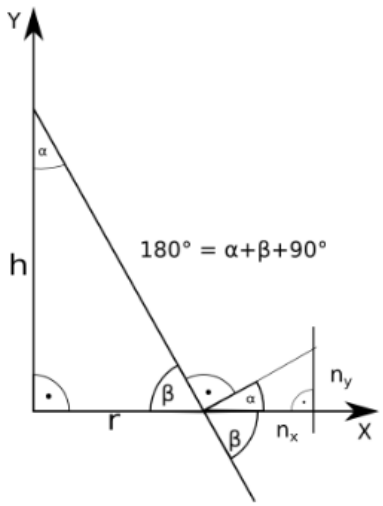

Figura 1: Normal da hipotenusa de um triângulo

Observando a imagem podemos facilmente verificar que a altura do cone seria representado pelo lado `h` do triângulo, e o raio da base do mesmo será representado pelo lado `r`. O objetivo é descobrir os cálculos para obter a perpendicular à hipotenusa deste triângulo. É de notar ainda que esta perpendicular é a hipotenusa do triângulo mais pequeno, semelhante ao maior.

Desta forma, através da semelhança de triângulos, podemos obter os valores de n_x e n_y. Primeiro, seja `c` o valor da hipotenusa do triângulo principal, dado por: c = sqrt(r^2 + h^2)

De seguida, os valores de n_x e n_y são dados por: n_x = h / c n_y = r / c

Com efeito, a componente horizontal da normal da hipotenusa é o próprio n_x e a vertical o n_y.

Voltando ao problema a três dimensões, as normais são calculadas com ajuda ainda da posição radial do vértice em relação ao centro do cone.

Tal como vimos nas fases anteriores, a posição `x` dos vértices é calculada com o seno do ângulo e a `z` com o cosseno. Com efeito, para o cálculo das componentes horizontais das normais, a mesma estratégia é utilizada, multiplicando estes valores trigonométricos pelo n_x e n_y. Com efeito, podemos observar de seguida o código em que as normais são calculadas, em que os valores n_x e n_y já foram previamente calculados, e theta representa o ângulo atual do *slice* e attach(theta, br:next) representa o valor do próximo ângulo.

```
glm::vec3 normal = glm::normalize(glm::vec3(nx * sin(theta), ny, nx * cos(theta)));
glm::vec3 normal_next = glm::normalize(glm::vec3(nx * sin(theta_next), ny, nx *
cos(theta_next)));
```

De notar que apenas duas normais são calculadas. Isto deve-se ao facto de que as normais dos vértices do topo de cada *stack* são exatamente iguais às normais dos vértices imediatamente abaixo da base dessa mesma *stack*.

2.1.3.2. Coordenadas de Textura

As coordenadas de textura são atribuídas de maneira a mapear a textura uniformemente sobre a superfície do cone. No entanto, devido à natureza curva do mesmo, é necessário ajustar os cálculos das coordenadas de textura para garantir uma representação adequada da mesma ao longo das curvas laterais e na base plana.

1. **Base do Cone:**
   - Adotamos uma abordagem semelhante à utilizada para um plano, no entanto, devido à curvatura da superfície, é necessário ajustar os cálculos para garantir uma representação adequada da textura.
   - Ao considerar a base do cone como um círculo, podemos calcular as coordenadas de textura de forma simples.

```
texCoords.push_back(glm::vec2(v1.x / radius + 0.5f, v1.z / radius + 0.5f));
texCoords.push_back(glm::vec2(0.5f, 0.5f));
texCoords.push_back(glm::vec2(v2.x / radius + 0.5f, v2.z / radius + 0.5f));
```

2. **Lados do Cone:**
   - Para determinar as coordenadas de textura das laterais do cone, utilizamos um método baseado na projeção cilíndrica das coordenadas cartesianas dos vértices no mesmo. Isso significa que as coordenadas de textura são calculadas com base na posição dos vértices ao longo da circunferência do cone e da altura ao longo de sua superfície.
   - Ao dividir o cone em slices e stacks, calculamos as coordenadas de textura *(s, t)* para cada vértice com base na sua posição relativa em relação ao número total de slices e stacks. Isso é feito para garantir que a textura seja aplicada de forma uniforme em todas as laterais do cone, independentemente da sua geometria.

```
texCoords.push_back(glm::vec2(s1, -t1));
```

Aqui, `s1` e `t1` representam as coordenadas de textura do vértice atual, calculadas com base na fatia e pilha atual do *loop* (segue a mesma lógica que já foi apresentado nas primitivas anteriores).

2.1.4. Sphere

De seguida, o cálculo das normais e das coordenadas de textura na esfera demonstraram ser relativamente fáceis, apesar da estrutura já ser mais complexa.

2.1.4.1. Normais

Para a esfera, as normais são calculadas com base na posição de cada vértice em relação ao centro da mesma. Isso é essencial para garantir que as normais apontem para fora da esfera, independentemente da sua posição. Dessa forma, as normais são calculadas como a normalização das coordenadas dos vértices, tornando assim o processo de cálculo das normais na esfera bastante simples.

```
glm::vec3 n1 = glm::normalize(v1);
glm::vec3 n2 = glm::normalize(v2);
glm::vec3 n3 = glm::normalize(v3);
glm::vec3 n4 = glm::normalize(v4);
```

2.1.4.2. Coordenadas de Textura

O cálculo das coordenadas de textura para a esfera envolve uma projeção das coordenadas esféricas para coordenadas de textura planares. Isso é feito atribuindo valores de s e t com base na latitude e longitude dos vértices (novamente recorrendo a stacks e slices).

```
glm::vec2 t1 = glm::vec2(((float)i) / slices, ((float)j) / stacks);
glm::vec2 t2 = glm::vec2(((float)(i + 1)) / slices, ((float)j) / stacks);
glm::vec2 t3 = glm::vec2(((float)i) / slices, ((float)(j + 1)) / stacks);
glm::vec2 t4 = glm::vec2(((float)(i + 1)) / slices, ((float)(j + 1)) / stacks);
```

Assim como nas outras primitivas, as coordenadas de textura são inseridas no arquivo .3d seguindo uma ordem consistente com a dos vértices, garantindo uma texturização uniforme em toda a superfície da esfera.

2.1.5. Patch

O cálculo das normais e coordenadas de textura para o *patch* é uma operação complexa que envolve a interpolação de curvas e superfícies de Bezier. Este processo é essencial para garantir a renderização adequada do *patch*, permitindo a sua correta iluminação e mapeamento de texturas.

2.1.5.1. Normais

No caso de um patch de Bezier, as normais precisam ser calculadas em cada ponto da superfície para garantir uma representação visualmente precisa. Assim sendo, para calcular as normais, duas técnicas principais foram empregadas: a interpolação das derivadas parciais e o produto vetorial.

1. **Interpolação das Derivadas Parciais:**

Para calcular as normais num ponto específico *(u,v)* da superfície de Bezier, primeiro calculamos as derivadas parciais da superfície em relação a u e v. Isso nos dá as direções tangentes à superfície nas direções u e v.

As derivadas parciais são calculadas usando a fórmula da derivada de uma curva de Bezier:

$$\frac{\mathrm{dP}}{\mathrm{du}} = \sum_{i=0}^{3} P_i.B'(i,u)$$

$$\frac{\mathrm{dP}}{\mathrm{dv}} = \sum_{j=0}^{3} P_j.B'(j,v)$$

onde *B´(i,u)* e *B´(j,v)* são as derivadas do polinômio de Bernstein de grau i e j em relação a u e v, respectivamente.

2. **Produto Vetorial:**

Com as derivadas parciais em mãos, calculamos a normal à superfície usando o produto vetorial:

$$N^{\rightarrow} = \frac{\mathrm{dP}}{\mathrm{du}} * \frac{\mathrm{dP}}{\mathrm{dv}}$$

Este produto nos dá uma normal que é perpendicular tanto à direção uu quanto à direção vv, fornecendo assim a orientação correta da superfície nesse ponto. Finalmente, normalizamos o vetor resultante N para garantir que tenha comprimento unitário:

$$N^{\rightarrow}_{\mathrm{norm}} = \frac{N^{\rightarrow}}{|| \ N^{\rightarrow} \ ||}$$

Isso é importante para garantir que a normal represente adequadamente a direção da superfície e seja útil para cálculos de iluminação.

2.1.5.2. Coordenadas de Textura

O cálculo das coordenadas de textura para o patch de Bezier envolve uma parametrização da superfície em termos de u e v, semelhante à abordagem usada para calcular a posição dos vértices. As coordenadas de textura são calculadas atribuindo valores de s e t com base nos valores dos parâmetros u e v. Tornando assim o processo de cálculo das coordenadas de textura bastante direto, o único cuidado a

ter é garantir que a ordem de inserção das coordenadas de textura segue a mesma ordem de inserção dos vértices.

```
texCoords.push_back(glm::vec2(u0, v0));
texCoords.push_back(glm::vec2(u1, v0));
texCoords.push_back(glm::vec2(u1, v1));

texCoords.push_back(glm::vec2(u0, v0));
texCoords.push_back(glm::vec2(u1, v1));
texCoords.push_back(glm::vec2(u0, v1));
```

2.1.6. Torus

Apesar da estrutura complexa que é o Torus, o processo de cálculo das suas normais e coordenadas de textura não é muito complexo.

2.1.6.1. Normais

As normais são calculadas para cada vértice do torus com base na direção radial em relação ao centro do torus. Isso é feito subtraindo a posição do vértice do vetor que aponta para o centro do torus e normalizando o resultado. Essa abordagem garante que as normais apontem para fora do torus em todos os pontos da sua superfície.

```
glm::vec3 n1 = glm::normalize(v1 - glm::vec3(radius * cos(curr_phi), 0, radius *
sin(curr_phi)));

glm::vec3 n2 = glm::normalize(v2 - glm::vec3(radius * cos(curr_phi), 0, radius *
sin(curr_phi)));

glm::vec3 n3 = glm::normalize(v3 - glm::vec3(radius * cos(next_phi), 0, radius *
sin(next_phi)));

glm::vec3 n4 = glm::normalize(v4 - glm::vec3(radius * cos(next_phi), 0, radius
* sin(next_phi)));
```

2.1.6.2. Coordenadas de Textura

Tal como já foi feito diversas vezes nas primitivas anteriores, os parâmetros de s e t no Torus são calculados tendo em conta o número de slices e stacks do mesmo, garantindo uma distribuição uniforme das coordenadas de textura em toda a sua superfície.

```
glm::vec2 t1 = glm::vec2((float)i / slices, (float)j / stacks);
glm::vec2 t2 = glm::vec2((float)i / slices, (float)(j + 1) / stacks);
glm::vec2 t3 = glm::vec2((float)(i + 1) / slices, (float)(j + 1) / stacks);
glm::vec2 t4 = glm::vec2((float)(i + 1) / slices, (float)j / stacks);
```

2.2. Integração de Iluminação e Texturização na Engine

Com o cálculo das normais e coordenadas de textura nos diversos modelos concluído, foi possível passar para a próxima fase do trabalho, a integração das novas *tags* dos ficheiros de *input*, ou seja, iluminação e texturas.

2.2.1. Estruturas

Antes de qualquer coisa, foi necessário criar e adaptar estruturas para guardar os novos valores inseridos nos ficheiros.

2.2.2. Material

Em primeiro lugar, temos a criação de uma nova estrutura `Material` para guardar os valores dos materiais dos modelos.

```
struct Material {
    float diffuse[4] = {200.0f / 255.0f, 200.0f / 255.0f, 200.0f / 255.0f,1.0f};
    float ambient[4] = {50.0f / 255.0f, 50.0f / 255.0f, 50.0f / 255.0f ,1.0f};
    float specular[4] = {0.0f, 0.0f, 0.0f,1.0f};
    float emissive[4] = {0.0f, 0.0f, 0.0f,1.0f};
    float shininess = 0.0f;
};
```

Podemos observar que alguns valores são preenchidos por *default*. Estes valores são alterados posteriormente consoante os presentes no *input*.

2.2.3. Light

Esta estrutura é bastante simples, guardando simplesmente o tipo da luz (guardando unicamente o primeiro carácter, o bastante para diferenciar cada tipo de luz), e os seus parâmetros, vetor de tamanho variável pois depende do tipo da luz.

```
struct Light {
    char type;
    float *params;
};
```

2.2.4. Model

Esta estrutura recebeu pequenas alterações.

```
struct Model {
    std::string filename;
    std::vector<Triangle> triangles;
    GLuint vbo;
    GLuint NormalVBO;
    GLuint TexCoordVBO;
    GLuint textureID;
    Material material;
};
```

Foram adicionados três novos parâmetros.

- **TexCoordVBO** : Id para o VBO das texturas do modelo;
- **TextureID** : Id referente à textura associada ao modelo;
- **material** : Estrutura para guardar os materiais associados ao modelo.

2.2.5. World

Por fim, estrutura `World` também recebeu uma adição.

```
struct World {
    Camera camera;
    Window window;
    std::map<std::string, Model> models;
    Tree tree;
    std::vector<Light> lights;
};
```

Esta adição é traduzida no novo atributo `lights`, vetor que guarda todas as iluminações especificadas no *input*.

2.3. Parsing

O *parsing* dos ficheiros *xml* foi ligeiramente alterado para refletir as adições dos novos atributos nos ficheiros.

Primeiramente, antes de ler a secção dos modelos, é verificada a existência do atributo "lights" no ficheiro. Se este existir, é então realizado o *parsing* referente a este atributo, preenchendo assim o vetor referido na Seção 2.2.5 com os valores das luzes.

A próxima mudança foi a implementação do reconhecimento da *tag* "color". Este atributo é reconhecido durante o *parse* do modelo. A estrutura referida na Seção 2.2.2 é então preenchida com os novos valores.

Por último, é ainda reconhecida a *tag* "texture" durante o *parse* do modelo. Nesta situação, o programa carrega no momento a textura com o nome presente no ficheiro com recurso à biblioteca stb_image. É ainda preenchida a estrutura do modelo com o identificador da textura.

2.4. Texturas

Para a renderização das texturas foi necessário definir vários parâmetros e funções para atingir os resultados pretendidos.

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
(...)
glGenerateMipmap(GL_TEXTURE_2D);
```

As duas primeiras linhas do excerto de código acima são usadas para definir o comportamento da textura quando as coordenadas de textura excedem os limites de 0 a 1 no eixo S(horizontal) e T(vertical), respetivamente. O valor *GL_REPEAT* faz com que a textura se repita, sendo que tal característica é particularmente útil para superfícies grandes onde a textura precisa de ser aplicado de forma contínua (como é o caso em vários dos nossos modelos).

De seguida, a terceira linha especifica o método de filtragem que será utilizado quando a textura precisar ser minificada, ou seja, quando for reduzida para se ajustar a um espaço menor na tela. A opção *GL_LINEAR* utiliza interpolação linear para suavizar a textura, resultando numa transição mais suave entre os pixeis e evitando o efeito de "pixelização".

Para além disso, a quarta linha, similar ao filtro de minificação, define o método de filtragem usado quando a textura precisa de ser ampliada, ou seja, quando for aumentada para se ajustar a um espaço maior na tela. A opção *GL_LINEAR* garante que a textura é ampliada de forma suave, preservando a qualidade visual.

Por fim, a última linha, define uma função que gera automaticamente um conjunto de *mipmaps* para a textura especificada. Mipmaps são uma sequência de texturas pre-calculadas, cada uma com uma resolução reduzida em comparação com a anterior. A principal vantagem de usar *mipmaps* é melhorar a performance de renderização e a qualidade visual. Quando uma textura é minificada, em vez de usar apenas a textura original, o sistema pode usar a versão de resolução mais baixa da textura apropriada para a escala atual. Isso reduz a carga de trabalho na GPU e minimiza artefatos visuais como *aliasing*. Além disso, o uso de *mipmaps* pode melhorar a cache de textura, resultando num melhor desempenho geral.

2.5. Iluminação

Nesta fase foi-nos pedido que a nossa *engine* suportasse 3 tipos diferentes de luzes.

Para cumprir tal desafio, antes de renderizar os modelos, e depois de posicionar a câmara, o programa itera por todas luzes guardadas aquando o *parsing*.

1. **Point Light**

```
float position[4] = { world.lights[i].params[0], world.lights[i].params[1],
world.lights[i].params[2], 1.0f };
glLightfv(lightID, GL_POSITION, position);
glLightf(lightID, GL_QUADRATIC_ATTENUATION, 0.0f);
```

Para definir as luzes do tipo `Point` basta criar um vetor com os valores da posição da luz e um quarto elemento com o valor 1.0, que simboliza que a luz é de facto pontual. Depois de criar a luz propriamente dita, a atenuação da luz é colocada a zero.

2. **Diretional Light**

```
float position[4] = { world.lights[i].params[0], world.lights[i].params[1],
world.lights[i].params[2], 0.0f };
glLightfv(lightID, GL_POSITION, position);
```

O código para as luzes direcionais é bastante semelhante à anterior. Uma diferença reside no quarto elemento do vetor da luz, que é colocado a 0.0, simbolizando uma luz direcional. Em adição, luz direcional não necessita da atenuação realizada na luz anterior.

3. **Spot Light**

```
float position[4] = { world.lights[i].params[0], world.lights[i].params[1],
world.lights[i].params[2], 1.0f };
glLightfv(lightID, GL_POSITION, position);
glLightfv(lightID, GL_SPOT_DIRECTION, world.lights[i].params + 3);
glLightf(lightID, GL_SPOT_CUTOFF, world.lights[i].params[6]);
glLightf(lightID, GL_SPOT_EXPONENT, 0.0f);
```

Na altura do *parsing* os valores para este tipo de luz foram todos guardados no mesmo vetor. As primeiras três posições referem a posição da luz, as 3 seguintes referem-se à direção e a última ao atributo *cutoff*. Desta forma, é então definida a luz direcional, seguida direção e o *cutoff*. Importante ainda de notar a existência da última linha, que coloca o exponente a zero. Este valor tem como objetivo uniformizar a intensidade da `spotlight` em todo o feixe de luz dentro do ângulo de `cutoff`.

2.6. Materiais

Para criar um modelo visual realista definimos diversos componentes de cor para cada material:

- **Diffuse:** Esta é a cor principal do material quando iluminado diretamente. Ela define como a luz é refletida em todas as direções a partir da superfície do objeto
- **Ambient:** Representa a cor do objeto sob luz ambiente, simulando a luz refletida de todas as direções.
- **Specular:** Define a cor dos reflexos especulares, que aparecem como pontos de luz brilhante na superfície do objeto.
- **Emissive:** Indica a cor que o objeto emite, simulando a luz proveniente do próprio objeto.
- **Shininess:** Controla a nitidez dos reflexos especulares. Valores altos resultam em reflexos mais nítidos e valores baixos resultam em reflexos mais suaves.

Estes componentes são utilizados para, em conjunto com a iluminação, permitiram uma coloração realista dos modelos numa cena aquando a presença de luz.

Tal como era pedido no enunciado, caso os ficheiros de configuração não especificassem os materiais, sendo que estes valores já foram apresentados anteriormente na Seção 2.2.2.

Quanto à implementação, os materiais foram introduzidos e corretamente implementados usando um pequeno número de linhas, considerando que para sua correta utilização basta utilizar a função `glMaterialfv`.

```
glMaterialfv(GL_FRONT, GL_DIFFUSE, model.material.diffuse);
glMaterialfv(GL_FRONT, GL_AMBIENT, model.material.ambient);
glMaterialfv(GL_FRONT, GL_SPECULAR, model.material.specular);
glMaterialfv(GL_FRONT, GL_EMISSION, model.material.emissive);
glMaterialf(GL_FRONT, GL_SHININESS, model.material.shininess);
```

3. Testes

Foram realizados vários testes com o intuito de verificar o bom funcionamento das técnicas implementadas nesta fase do projeto. Deste modo, podemos observar os resultados da utilização de vários tipos de iluminação e mapeamento de texturas em diversos modelos. Seguem-se os resultados dos testes:

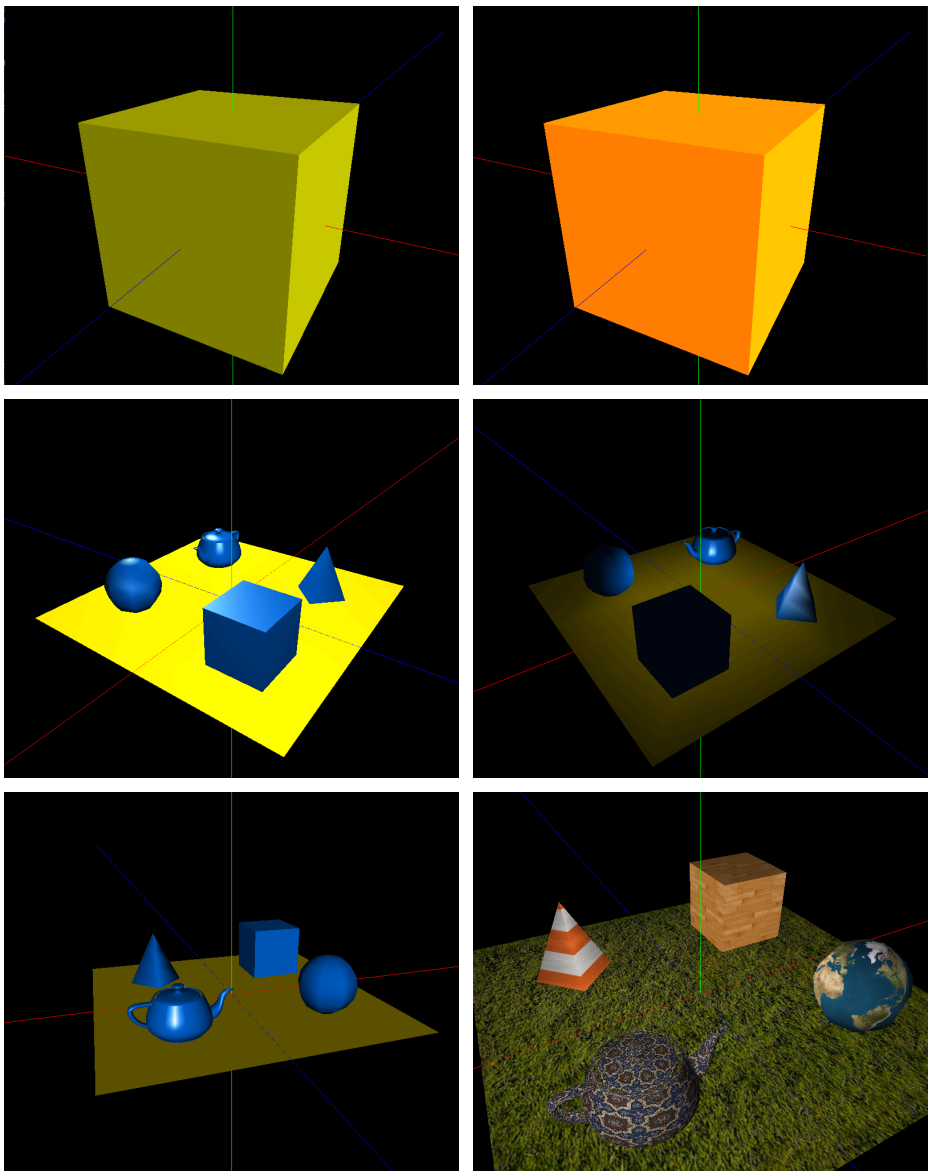

Figura 2: Resultados Obtidos nos Testes da Fase 4

Podemos então perceber que o sistema se comporta da maneira correta face a todos os cenários propostos.

4. Demo do Sistema Solar

A demo do Sistema Solar foi adaptada para utilizar as novas funcionalidades desenvolvidas nesta fase do projeto, incluindo o cálculo de normais, texCoords, bem como a implementação de texturização e iluminação. Essas transformações permitiram criar um ambiente mais realista do Sistema Solar.

Com o cálculo de normais e texCoords, foi possível melhorar a representação visual dos corpos celestes, garantindo que a iluminação fosse adequadamente aplicada em cada ponto da superfície. Além disso, a texturização adicionada proporcionou detalhes visuais adicionais, tornando os planetas, luas e o cometa mais realistas.

Agora, os planetas orbitam em torno do Sol, enquanto as luas giram em torno de seus planetas, mantendo uma iluminação e textura coerentes em todos os momentos. Todos os corpos celestes continuam a girar em torno dos seus próprios eixos, adicionando ainda mais realismo ao ambiente.

Segue uma imagem do modelo aprimorado:

Figura 3: Demo do Sistema Solar

5. Conclusões

Após a implementação do cálculo das normais e das coordenadas de textura nos diversos modelos, bem como a implementação de iluminação e texturização no lado da engine, estamos satisfeitos por termos alcançado os objetivos estabelecidos para a Fase 4 do projeto. Com estas funcionalidades implementadas, acreditamos que houve avanços significativos na capacidade de criação de cenários realistas e completos por parte da engine desenvolvida.

Para além disso, as funcionalidades implementadas contribuem para uma experiência de utilizador mais imersiva e envolvente e a iluminação e texturização melhoraram a qualidade visual dos cenários renderizados, proporcionando aos usuários uma experiência mais realista e cativante.

As novas adições, como o cálculo das normais e das coordenadas de textura, ampliaram as capacidades da engine para criar cenários mais complexos e detalhados. Agora, somos capazes de simular uma variedade maior de ambientes, desde paisagens naturais até sistemas planetários, com maior riqueza de detalhes e realismo.

A implementação dessas funcionalidades torna a engine mais flexível e versátil, sendo que agora podemos personalizar diferentes condições de iluminação e aplicar texturas específicas oferecendo aos desenvolvedores maior liberdade criativa na construção dos seus cenários.

Em suma, o grupo está satisfeito com o trabalho desenvolvido e achamos que fomos capazes de criar uma engine bastante completa e que fornece aos seus utilizadores diversas funcionalidades com que podem experimentar e desenvolver os seus cenários.

No entanto, reconhecemos que ainda há oportunidades para melhorias e refinamentos que, infelizmente, não poderam ser feitas no tempo limite, como por exemplo:

- **Melhorias na Interface do Utilizador:** Implementação do *Imgui* para aprimorar a interface do utilizador, tornando o sistema mais intuitivo e fácil de usar.
- **Otimização de Desempenho:** Re-renders desnecessários, bem como cálculo de vértices, normais e texCoords mais complexos do que poderiam ser são alguns dos diversos problemas que poderiam ser resolvidos e otimizados.
- **Maior diversidade de Demos:** Reconhecemos que poderiam ter sido desenvolvidas mais demos que permitissem demonstrar a capacidade total da engine criada, bem como as diversas funcionalidades implementadas.