Computação Gráfica

LEI - 3º ano - 2º semestre
Universidade do Minho

Trabalho Prático

Fase 4

Grupo 10

Trabalho realizado por:	*Número*
Duarte Moreira	A93321
Lucas Carvalho	A93176
Manuel Novais	A89512

Braga, 5 de junho de 2022

Índice

1. Introdução

O presente relatório foi realizado no âmbito da Unidade Curricular de Computação Gráfica e tem como objetivo expor detalhadamente as decisões e abordagens tomadas pelo grupo durante a quarta e última fase de desenvolvimento do projeto.

Este trabalho, dividido em 4 fases, tem como objetivo final a criação de um mini cenário baseado em gráficos 3D. Nesta etapa final é-nos pedido, mais uma vez, a alteração de ambas as aplicações, tanto do **generator** como do **engine**. Aqui, a nova aplicação **generator** deverá ser capaz de calcular vetores normais e coordenadas de textura. Já o **engine** deverá ser atualizado de forma a conseguir suportar e lidar com a leitura destes novos pontos de forma a reproduzir modelos com iluminação e texturas.

2. Gerador

2.1 Normais e Texturas

2.1.1 Plane

Para o plano o cálculo do vetor normal é direto, sendo este $(0, 1, 0)$.

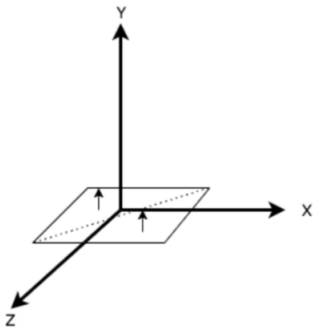

Figura 2.1: Plano xOz.

O mapeamento das texturas foi também muito simples e direto, para cada quadrado que é desenhado "dentro"do plano é aplicado o seguinte processo:

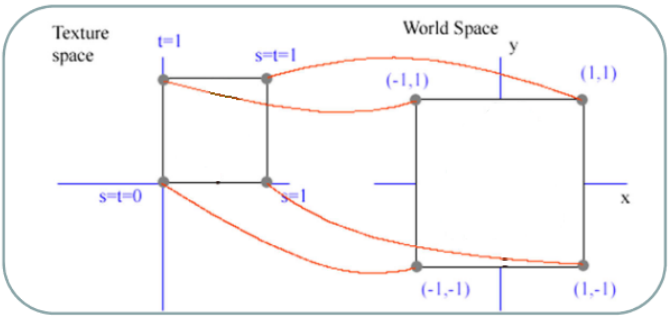

Figura 2.2: Aplicação das texturas.

2.1.2 Box

De forma semelhante ao plano obteve-se os vetores normais de cada um dos seis lados do cubo, tendo agora em atenção na face em que nos encontrávamos. Assim obtiveram-se os seguintes resultados:

- Topo: $(0, 1, 0)$
- Base: $(0, -1, 0)$
- Face frontal: $(0, 0, 1)$
- Face traseira: $(0, 0, -1)$
- Face lateral direita: $(1, 0, 0)$
- Face lateral esquerda: $(-1, 0, 0)$

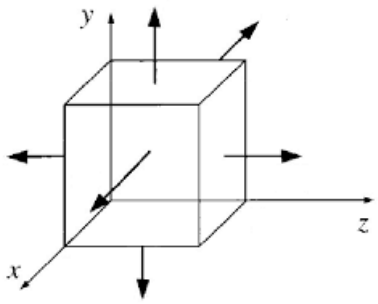

Figura 2.3: Vetores normais de um cubo.

Para o cálculo das texturas utilizados uma função auxiliar que normaliza valores numa escala de $[-size/2, size/2]$, onde *size* é o comprimento do cubo, para uma escala de $[0, 1]$.

```
float find_texture_plane(float value, float size) {
    // (value - min) / (max - min)
    float min = -size / 2.0f;
    float max = -min;
    float normalized = (value - min) / (max - min);
    return normalized;
}
```

2.1.3 Sphere

O cálculo dos vetores normais da esfera segue a seguinte fórmula:

$$(x/raio, y/raio, z/raio)$$

Ou seja, o vetor normal vai ser igual ao ponto mas normalizado.

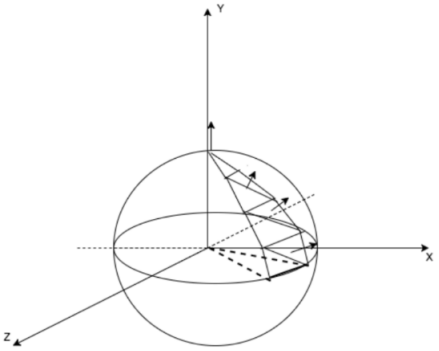

Figura 2.4: Vetores normais de uma esfera.

Para o cálculo das texturas foram aplicadas as seguintes fórmulas:

$$u = i/slices$$

$$v = (stacks - j)/slices$$

Onde i respresenta a "fatia"da esfera em que o ponto calculado se encontra, *slices* o número total de fatias da esfera, j respresenta a "camada"da esfera em que o ponto calculado se encontra e *stacks* o número total de camadas da esfera.

2.1.4 Cone

O cálculo do vetor normal da base do cone é direto, $(0, -1, 0)$. Já para o corpo do mesmo é usada a seguinte fórmula:

$$(\sin(\alpha), \pi - \pi/2 - \arctan(altura, raio), \cos(\alpha))$$

Onde α representa o ângulo da "fatia"onde se encontra o ponto em questão, e o valor de y representa a inclinação do cone que será igual em todos os pontos.

Para a aplicação das texturas foi seguido o seguinte algoritmo:

$$u = camada/altura$$

$$v = \sin(\alpha)$$

Onde a componente **u** é representada pela altura normalizada da camada onde se encontra o ponto em questão, e a componente α o ângulo da respetiva fatia.

2.1.5 Teapot

O cálculo dos vetores normais do *teapot* é realizado em simultâneo com o cálculo dos pontos uma vez que temos todas as variáveis necessárias para o cálculo dos vetores tangentes. Estes são calculados da seguinte forma:

$$\frac{\partial B(u,v)}{\partial u} = [3u^2 \quad 2u \quad 1 \quad 0]M \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} M^T V^T$$

$$\frac{\partial B(u,v)}{\partial v} = UM \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} M^T \begin{bmatrix} 3v^2 \\ 2v \\ 1 \\ 0 \end{bmatrix}$$

Figura 2.5: Vetores normais do teapot.

Depois de efetuados todos os cálculos necessitamos apenas de normalizar os vetores e realizar o *cross product* entre eles para assim obter o vetor normal.

Por outro lado, a obtenção das coordenadas de textura é direta. Estas são representadas pelas variáveis **u** e **v** encontradas na função *build_bezier* que estão relacionadas com o nível de tesselação fornecido.

3. Motor

3.1 Iluminação

Para a iluminação, uma vez que podemos ter três tipos de luz diferentes, recorreu-se ao conceito de hierarquização. Como tal, foi criada uma classe **Light** que apresenta três subclasses diferentes: **PointLight**, **DirectionalLight** e **SpotLight**.

A classe **Light** apresenta variáveis comuns a todas as outras uma vez que é a classe mais geral.

```
class Light {
public:
        GLenum i;
        float* diffuse;
        float* ambient;
        float* specular;
        Light(GLenum i);
        Light(GLenum i, float* diffuse, float* ambient, float* specular);
        void init();
        virtual void apply();
};
```

Onde *i* identifica o número da luz, sendo que podemos ter até um máximo de 8 simultâneamente. Os três vetores de *floats* apresentados representam as componentes da luz que são inicializados com valores *default*. A função *init* faz uso de *glEnable* e *glLightfv* para ligar e definir as propriedades da luz. Já a função *apply* é implementada pelas respetivas subclasses, uma vez que cada uma delas vai apresentar comportamentos distintos, e tem como objetivo posicionar a luz e aplicar atributos dependendo do seu tipo.

- **PointLight**: apresenta uma posição e um valor de atenuação;
- **DirectionalLight**: apresenta uma direção;
- **SpotLight**: apresenta uma posição, uma direção, um *cutoff* e um expoente;

Para coloração dos modelos podem ser fornecidos ou não valores **RGB** para as componentes, difusa, ambiente, especular e emissiva. Para além disso é também fornecido um valor de *shininess*. Para suportar esta mudança foram adicionadas novas variáveis de instância à classe **Model**. Assim, a quando do seu desenho estas propriedades são aplicadas com a utilização da função *glMaterialfv*.

3.2 Texturas

O tratamento das texturas levou também a um ajuste da classe **Model**. Foi necessário criar um construtor diferente para esta classe de maneira a permitir desenhar modelos com ou sem textura. Este novo construtor adiciona uma *string* para o nome da textura a ser carregada um **id** fornecido pela nova função *loadTexture*. Esta função foi criada, como o próprio nome indica, para carregar a textura e fazer os preparos iniciais para se fazer uso da mesma, tais como, definir certos parâmetros de textura.

Do lado da classe **VBO** esta tem agora a capacidade de ler e criar um *array* com as coordenadas de textura para posteriormente as copiar para a memória gráfica e mais tarde aplicá-las.

```
void VBO::prepareData(vector<Ponto> pontos) {
        vector<float> coords, normais, texturas;
        for (Ponto p : pontos) {
                (...)
                normais.push_back(p.nx);
                normais.push_back(p.ny);
                normais.push_back(p.nz);
                texturas.push_back(p.ti);
                texturas.push_back(p.tj);
        }
        // criar o VBO
        glGenBuffers(3, this->buffers);
        // copiar o vector para a memória gráfica
        (...)
        glBindBuffer(GL_ARRAY_BUFFER, this->buffers[1]);
        glBufferData(GL_ARRAY_BUFFER,sizeof(float) * normais.size(),
            normais.data() ,GL_STATIC_DRAW);

        if (this->texture) {
                glBindBuffer(GL_ARRAY_BUFFER, this->buffers[2]);
                glBufferData(GL_ARRAY_BUFFER, sizeof(float) * texturas.size(),
                    texturas.data(), GL_STATIC_DRAW);
        }
}

void VBO::draw(int textid) {
        (...)
        glBindBuffer(GL_ARRAY_BUFFER, this->buffers[1]);
        glNormalPointer(GL_FLOAT, 0, 0);

        if (this->texture && textid != -1) {
                glBindBuffer(GL_ARRAY_BUFFER, this->buffers[2]);
                glTexCoordPointer(2, GL_FLOAT, 0, 0);
        }
        if (this->texture) {
                glBindTexture(GL_TEXTURE_2D, textid);
                glDrawArrays(GL_TRIANGLES, 0, this->verticeCount);
                glBindTexture(GL_TEXTURE_2D, 0);
        }
        else glDrawArrays(GL_TRIANGLES, 0, this->verticeCount);
}
```

4. Resultados Obtidos

4.1 Testes

Apresentam-se a seguir os resultados obtidos com as nossas aplicações de acordo com os ficheiros de testes fornecidos pela equipa docente.

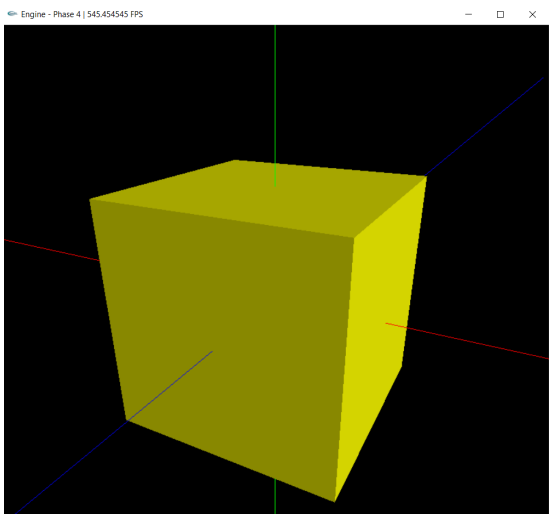

Figura 4.1: Resultado de *test_4_1.xml*

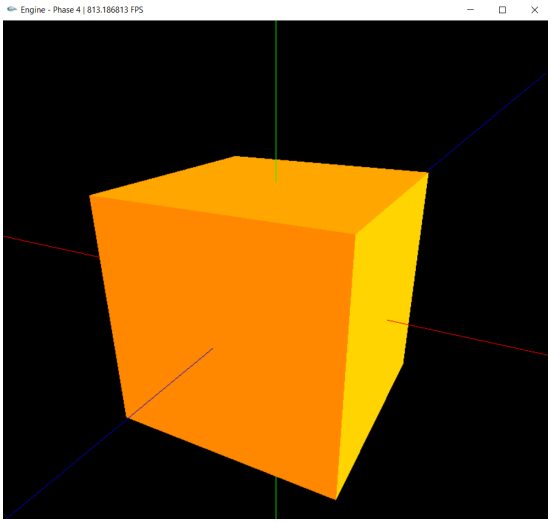

Figura 4.2: Resultado de *test_4_2.xml*

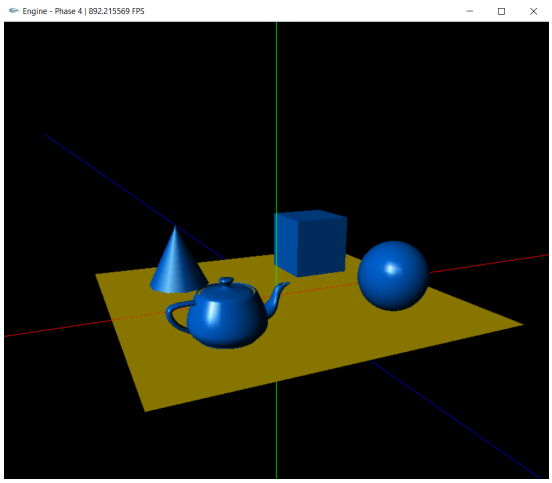

Figura 4.3: Resultado de *test_4_3.xml*

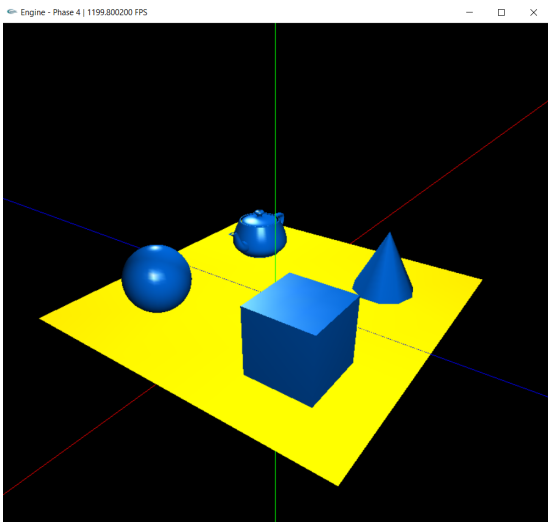

Figura 4.4: Resultado de *test_4_4.xml*

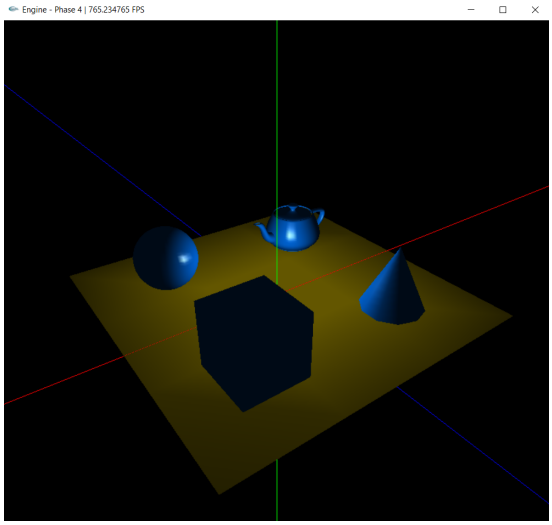

Figura 4.5: Resultado de *test_4_5.xml*

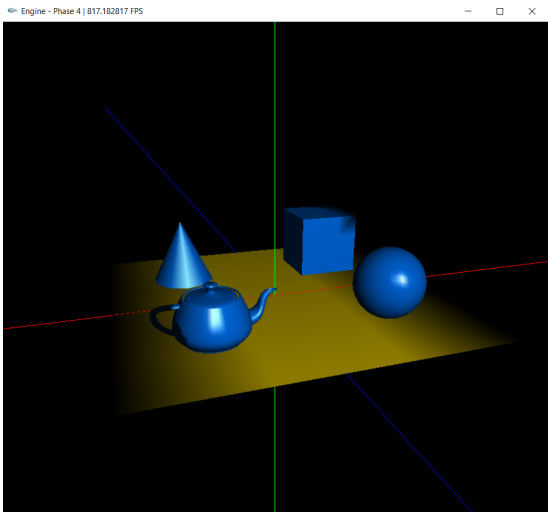

Figura 4.6: Resultado de *test_4_6.xml*

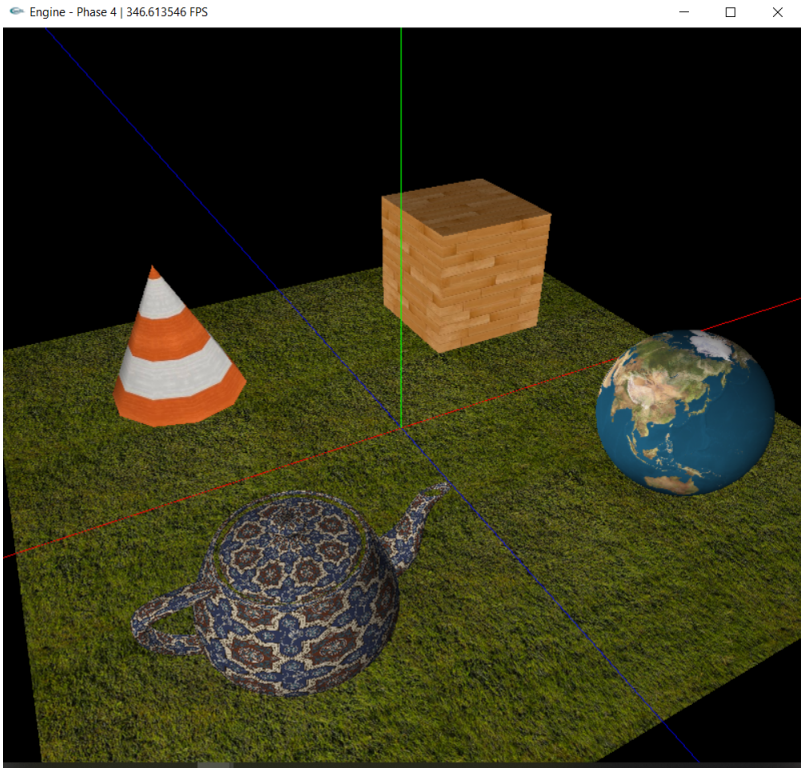

Figura 4.7: Resultado de *test_4_7.xml*.

4.2 Cenários de demonstração

De seguida expõem-se o cenário de demonstração pretendido para esta fase: um sistema solar com texturas e iluminação.

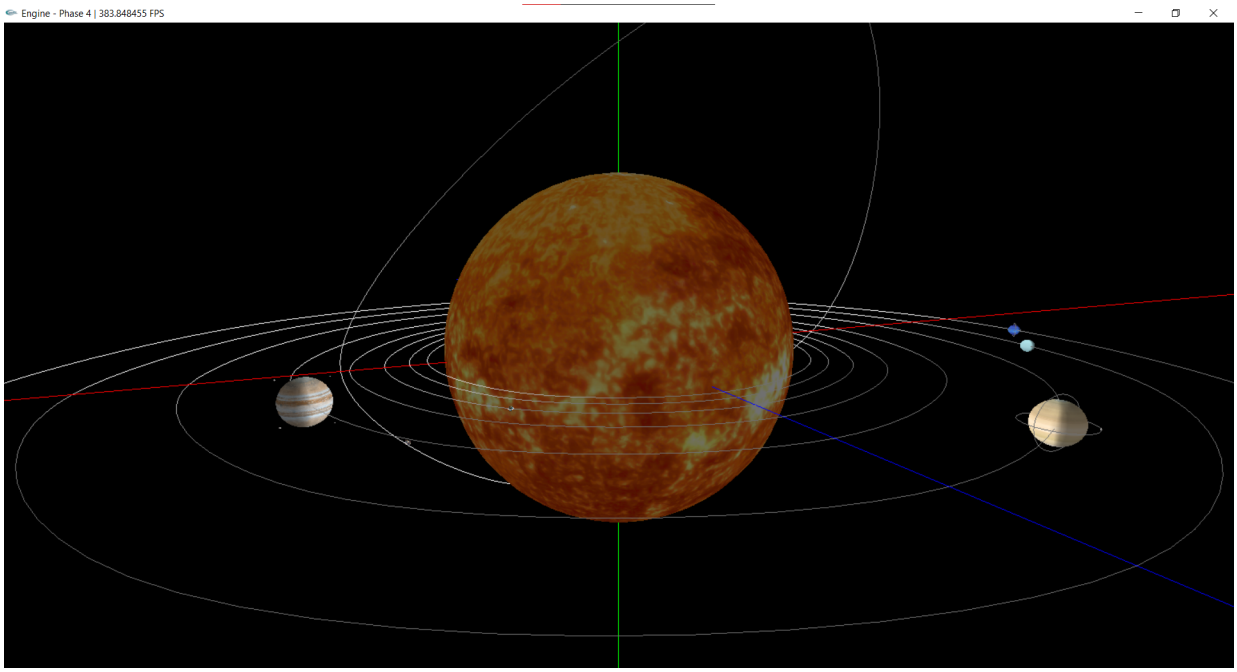

Figura 4.8: Sistema solar.

5. Conclusão

Esta última etapa caiu sobre a importância da iluminação e da textura que de facto fizeram toda a diferença no resultado final deste projeto. O trabalho realizado á volta destes tópicos permitiu consolidar a matéria aprendida no término do semestre e também reforçar o nosso conhecimento da ferramenta *OpenGl*.

Em jeito de conclusão e chegando ao final desta jornada, o grupo acredita ter conseguido atingir o objetivo inicialmente estabelecido que foi a modelação do nosso sistema solar.