Universidade do Minho
Escola de Ciências

# Computação Gráfica

Ciências da Computação
Fase 4

Bruno Fernandes (A95972) | Tiago Silva (A97450) | Tomás Pereira (A97402)

2 de junho de 2023

# Índice

# Capítulo 1

# Introdução

Este relatório foi elaborado no âmbito da quarta fase do trabalho prático da unidade curricular de Computação Gráfica, no qual foi proposto implementar iluminação e texturas, acrestando ao *generator* as normais e pontos de textura para cada modelo.

# Capítulo 2

# Generator

O *generator* passou a criar um ficheiro *.3d* em que a primeira linha indica o número total de coordenadas, sendo este número o mesmo para as normais e os coordenadas de textura. Depois dessa linha, aparecem as coordenadas da primitiva, seguido das normais e dos pontos de textura. Para tal, foi necessário alterar a função *wrtfile* para esta escrever no ficheiro toda a informação necessária.

```
void wrtfile(string objeto, string normal, string textura, string name) {
    ofstream file("..\\..\\engine\\Fich3d\\" + name);
    if (file.is_open()) {
        file << objeto << normal << textura << endl;
        file.close();
    }
    else {
        cout << "Nao foi possivel abrir o arquivo " << name << " para escrita." << endl;
    }
}
```

Figura 2.1: Função *wrtfile*

## 2.1 Vetores Normais

### 2.1.1 Plano

Como o plano está fixado no eixo xOz, a normal de todos os pontos do plano será o vetor com o sentido do eixo do y: **(0,1,0)**.

### 2.1.2 Cubo

Para o cubo, assumimos que é composto por seis planos, onde a normal de cada um é perpendicular ao mesmo.

- Face de cima: **(0,1,0)**;
- Face de baixo: **(0,-1,0)**;
- Face da esquerda: **(0,0,1)**;
- Face da direita: **(0,0,-1)**;
- Face da frente: **(1,0,0)**;
- Face de trás: **(-1,0,0)**.

### 2.1.3 Cone

Em primeiro lugar, calculamos as normais da base que, tal como o plano está fixada no eixo xOz mas a sua face exterior aponta no sentido contrário ao eixo do y: **(0,-1,0)**. Em seguida, calculamos as normais à superfície lateral, ou seja, normalizamos cada ponto (dividimos pelo raio).

```
s << atualR * sin(angulo) << ' ' << atualH << ' ' << atualR * cos(angulo) << '\n';
n << sin(angulo) << ' ' << height/stacks << ' ' << cos(angulo) << '\n';
```

Figura 2.2: `Exemplo da criação de uma normal para um ponto do cone`

### 2.1.4 Esfera

Para a esfera, tal como no cone, normalizamos todos os pontos, dividindo pelo raio.

### 2.1.5 Torus

Para o torus, normalizamos todos os pontos, dividindo pelo raio exterior.

### 2.1.6 Bezier

Infelizmente não conseguimos implementar as normais e as coordenadas de textur para as superfícies de Bezier a tempo da entrega.

## 2.2 Coordenadas de Textura

### 2.2.1 Plano

Para o plano, as coordenadas da textura foram calculadas tendo em conta que as extremidades do plano correspondem às extremidades da textura.

### 2.2.2 Cubo

Para o cubo, a textura será replicada para cada face. Sendo que cada face é um plano, foi usada uma estratégia semelhante à do plano.

### 2.2.3 Cone

Para a base do cone, fixamos o ponto do centro (0.5,0.5) e os restantes pontos são calculados com os valores das coordenadas de cada vértice. Para a superfície lateral dividimos, em cada ponto, o número da *slice* atual pelo número total de *slices* para o x, e dividimos o número da *stack* atual pelo número total de *stacks* para o y.

### 2.2.4 Esfera

Para a esfera, a coordenada x da textura é igual ao valor da *slice* a que o ponto pertence a dividir pelo número total de slices. A coordenada y é o valor da *stack* a que o ponto pertence a dividir pelo número de *stacks*.

### 2.2.5 Torus

Para o torus, o cálculo das coordenadas de textura é idêntico ao da esfera.

# Capítulo 3

# Engine

Nesta fase, para possibilitar a implementação da iluminação e das texturas criamos dois *buffers*, um para as normais e outro para as coordenadas de textura.

## 3.1 Iluminação

Na leitura do *XML* podem exister três tipos de iluminação: *point*, *directional* e *spot*. Para além disso, cada objeto tem valores de iluminação (difusa, ambiente, especular, emissiva e *shininess*). Para tal, criamos duas *structs*.

```
struct cor {
    int f = 0;
    float diffuse[4];
    float ambient[4];
    float specular[4];
    float emissive[4];
    float shininess;
};

struct iluminacao {
    string tipo;
    float pos[4];
    float dir[4];
    float cutoff;
};

struct objeto {
    vector <transformacao> transf;
    vector <float> model;
    vector <float> norm;
    vector <float> tex;
    cor color;
    GLuint tex_id = -1;
};
```

Figura 3.1: *Structs* "cor", "iluminacao"e "objeto"

A *struct* "cor"está presente nas características de cada objeto, ao contrário da *struct* "iluminacao", que é global e não tem referência a nenhum objeto.

Para implementar a iluminação, começamos por inicializá-la na função *main*.

```
if (luz.size() > 0) {
    // init
    float dark[4] = { 0.2, 0.2, 0.2, 1.0 };
    float white[4] = { 1.0, 1.0, 1.0, 1.0 };
    float amb[4] = { 0.2f, 0.2f, 0.2f, 1.0f };
    // controls global ambient light
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, amb);
    glEnable(GL_LIGHT0);

    for (int i = 0; i < luz.size(); i++) {
        if (i != 0) glEnable(GL_LIGHT0 + i);

        // light colors
        glLightfv(GL_LIGHT0 + i, GL_AMBIENT, dark);
        glLightfv(GL_LIGHT0 + i, GL_DIFFUSE, white);
        glLightfv(GL_LIGHT0 + i, GL_SPECULAR, white);
    }
}
```

Figura 3.2: `Inicialização da iluminação`

Depois, dependendo do tipo de luz, ativamo-la com as características especificadas.

```
for (int i = 0; i < luz.size(); i++) {

    if (luz[i].tipo == "point") {
        glLightfv(GL_LIGHT0 + i, GL_POSITION, luz[i].pos);
    }

    if (luz[i].tipo == "directional") {
        glLightfv(GL_LIGHT0 + i, GL_POSITION, luz[i].dir);
    }

    if (luz[i].tipo == "spot") {
        glLightfv(GL_LIGHT0 + i, GL_POSITION, luz[i].pos);
        glLightfv(GL_LIGHT0 + i, GL_SPOT_DIRECTION, luz[i].dir);
        glLightf(GL_LIGHT0 + i, GL_SPOT_CUTOFF, luz[i].cutoff);
        glLightf(GL_LIGHT0 + i, GL_SPOT_EXPONENT, 0.0);
    }
}
```

Figura 3.3: `Aplicação da iluminação`

Para implementar a componente dos materiais do objetos fazemos o seguinte:

```
if (objetos[i].color.f) {
    glMaterialfv(GL_FRONT, GL_DIFFUSE, objetos[i].color.diffuse);
    glMaterialfv(GL_FRONT, GL_AMBIENT, objetos[i].color.ambient);
    glMaterialfv(GL_FRONT, GL_SPECULAR, objetos[i].color.specular);
    glMaterialfv(GL_FRONT, GL_EMISSION, objetos[i].color.emissive);
    glMaterialf(GL_FRONT, GL_SHININESS, objetos[i].color.shininess);
}
```

Figura 3.4: `Implementação dos materiais`

## 3.2 Texturas

Para a implementação das texturas, começamos por carregá-las na leitura do *XML*. Neste processo, abrimos o ficheiro da imagem, guardamos o valor da largura e da altura, convertemos para RGBA, geramos um *id* e ativamos. De seguida, definimos os parâmetros de textura e enviamos a informação para o *OpenGL*.

Caso o objeto possua textura, será guardado na *struct* "objeto"o *id* da textura inicializada.

```
XMLElement* texture = n->FirstChildElement("texture");
if (texture) {
    string texfile = "..\\..\\test_files\\test_files_phase_4\\" + (string)texture->Attribute("file");
    const char* tex_file = texfile.c_str();

    unsigned int t, tw, th;
    unsigned char* texData;
    ilGenImages(1, &t);
    ilBindImage(t);

    ilLoadImage((ILstring)tex_file);
    tw = ilGetInteger(IL_IMAGE_WIDTH);
    th = ilGetInteger(IL_IMAGE_HEIGHT);
    ilConvertImage(IL_RGBA, IL_UNSIGNED_BYTE);
    texData = ilGetData();

    glGenTextures(1, &obj.tex_id);

    glBindTexture(GL_TEXTURE_2D, obj.tex_id);
    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_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, tw, th, 0, GL_RGBA, GL_UNSIGNED_BYTE, texData);

}
```

Figura 3.5: `Implementação das texturas`

A seguir, ativamos o *id* e desenhamos utilizando os *buffers*. Por fim, voltamos à textura *default*.

```
glBindBuffer(GL_ARRAY_BUFFER, buffers[i]);
glVertexPointer(3, GL_FLOAT, 0, 0);

glBindBuffer(GL_ARRAY_BUFFER, buffersN[i]);
glNormalPointer(GL_FLOAT, 0, 0);

glBindBuffer(GL_ARRAY_BUFFER, buffersT[i]);
glTexCoordPointer(2, GL_FLOAT, 0, 0);

glBindTexture(GL_TEXTURE_2D, objetos[i].tex_id);
glDrawArrays(GL_TRIANGLES, 0, vertexCount[i]);
glBindTexture(GL_TEXTURE_2D, 0);
```

Figura 3.6: `Desenho dos modelos`

## 3.3 Resultados Gerados

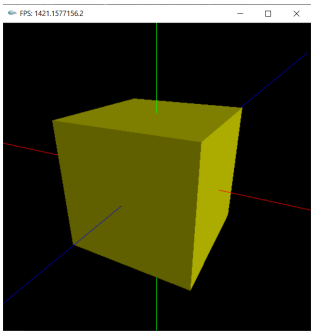

Figura 3.7: `Resultado do` *`test_4_1.xml`*

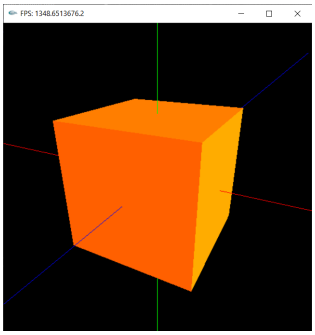

Figura 3.8: `Resultado do` *`test_4_2.xml`*

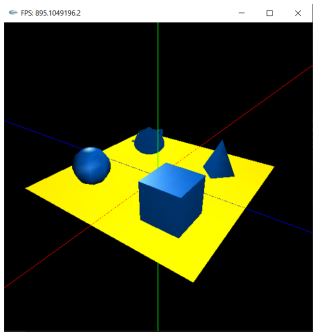

Figura 3.9: Resultado do *test_4_3.xml*

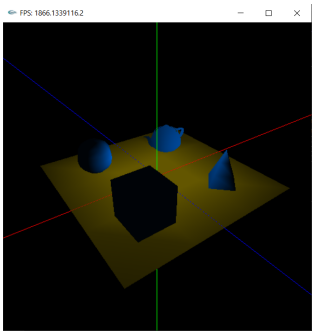

Figura 3.10: Resultado do *test_4_4.xml*

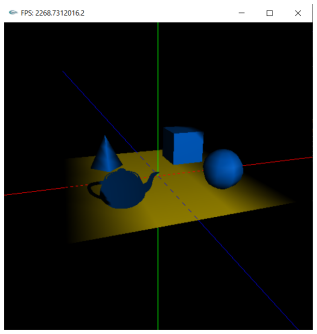

Figura 3.11: Resultado do *test_4_5.xml*

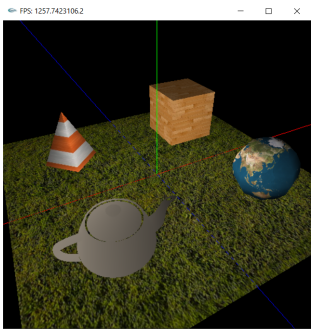

Figura 3.12: Resultado do *test_4_6.xml*

## 3.4 Modelo do Sistema Solar

Ao modelo da fase anterior, foram acrescentadas a iluminação e as texturas dos planetas e das luas.

```
<lights>
    <light type="point" posx="0" posy="0" posz="0" />
</lights>

<group>
    <group>                                                    <!-- SOL -->
        <transform>
            <scale x="20" y="20" z="20" />
            <rotate time="60" x="0" y="1" z="0"/>
        </transform>
        <models>
            <model file="sphere_1_20_20.3d" > <!-- generator sphere 1 8 8 sphere_1_20_20.3d -->
                <texture file="sun.jpg" />
                <color>
                    <diffuse R="255" G="255" B="255" />
                    <ambient R="255" G="255" B="255" />
                    <specular R="0" G="0" B="0" />
                    <emissive R="255" G="255" B="255" />
                    <shininess value="0" />
                </color>
            </model>
        </models>
    </group>
```

Figura 3.13: Excerto do documento *XML* para a geração do sistema solar

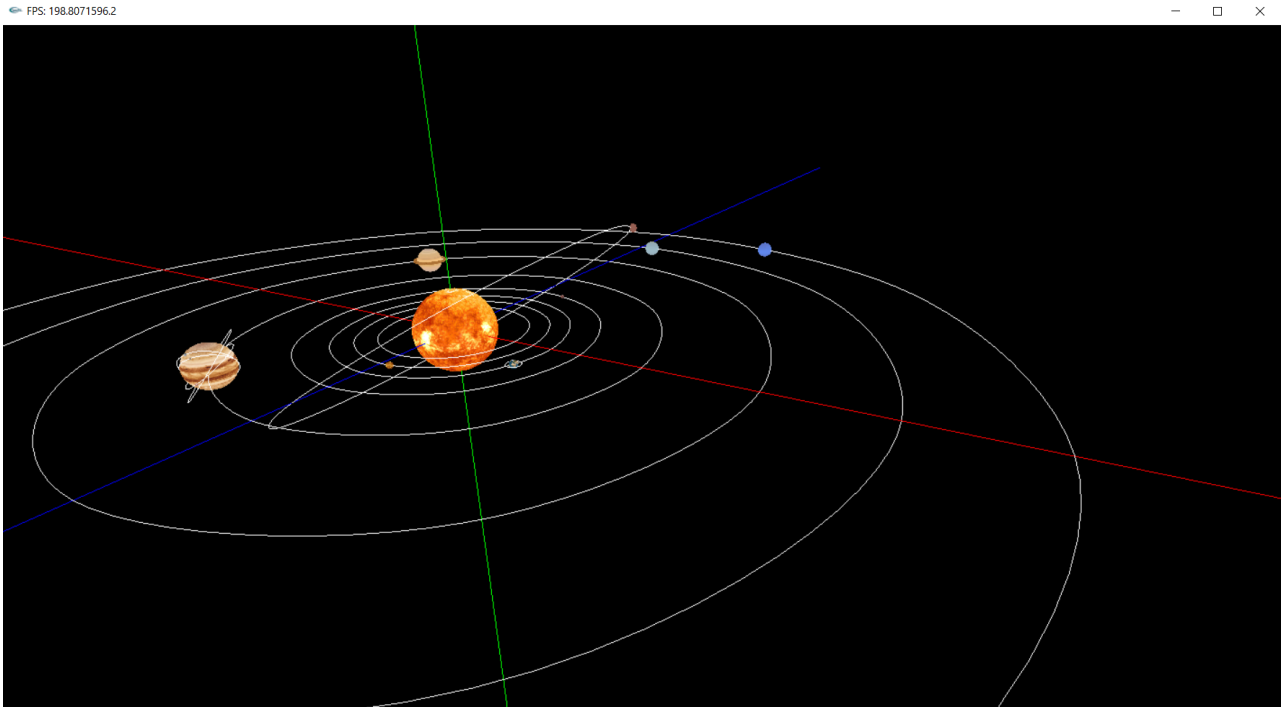


Figura 3.14: Modelo do sistema solar com iluminação uniforme

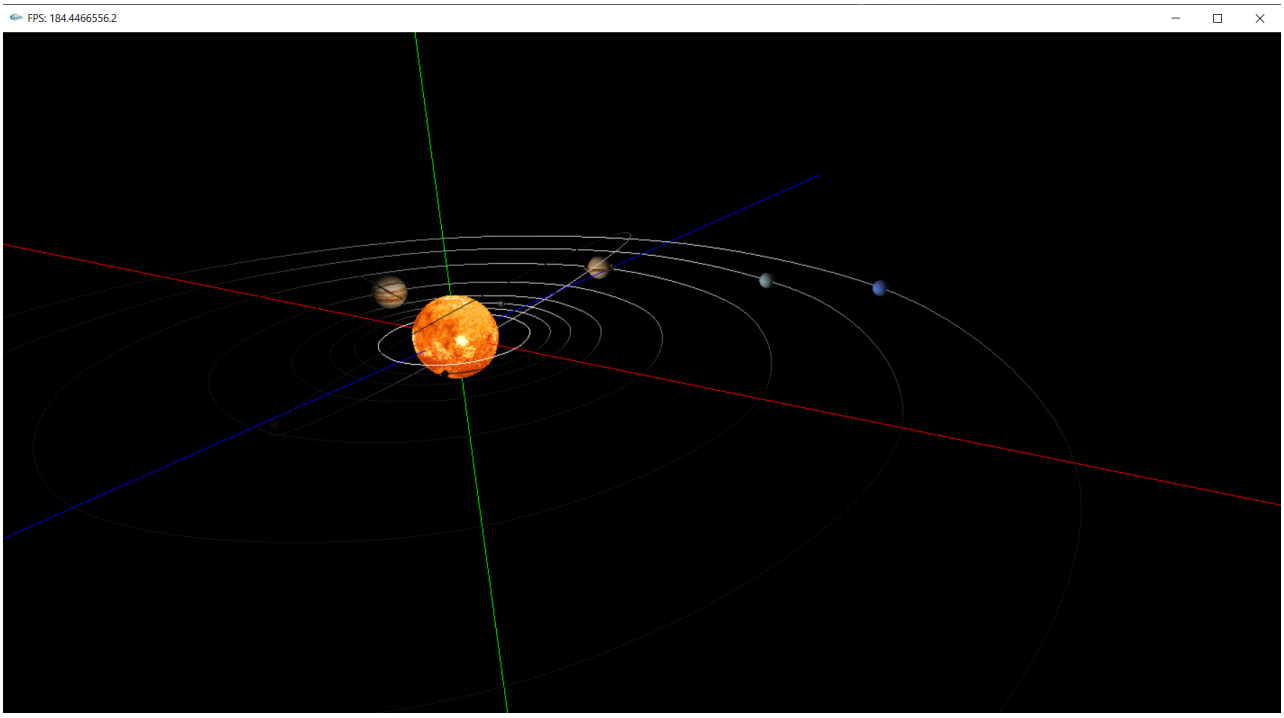


Figura 3.15: Modelo do sistema solar com iluminação realista

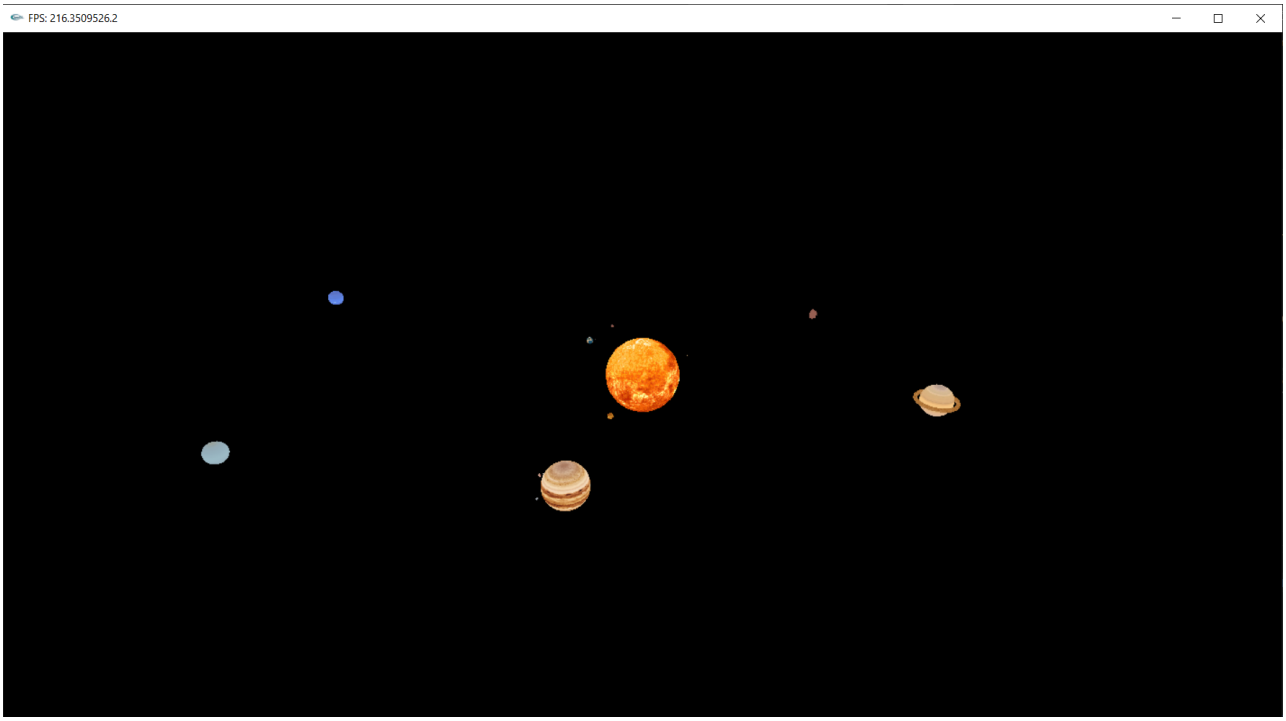


Figura 3.16: `Modelo sem eixos e curvas`

# Capítulo 4

# Conclusão

Nesta fase, a maior dificuldade encontrada foi o cálculo das normais e das coordenadas de textura para cada uma das primitivas. No entanto, a implementação da iluminação e das texturas na *engine* foi mais simples devido ao trabalho realizado nas fases anteriores.

Esta fase foi desafiante mas foi fundamental para consolidar-mos os nossos conhecimentos de iluminação e texturas. Deste modo, consideramos que cumprimos os objetivos e estamos satisfeitos com o resultado final do projeto.