



Universidade do Minho  
Departamento de Informática

## Computação Gráfica

Ano Letivo 2023/2024

# Trabalho Prático

## Fase 4 - Normais e Coordenadas de Textura

### **Grupo 13**

Ema Martins - A97678

Gonçalo Costa - A100824

Henrique Malheiro - A97455

Marta Rodrigues - A100743

# Índice

<b>1. Introdução .....</b>	<b>3</b>
<b>2. Generator .....</b>	<b>3</b>
2.1. Cálculo das Normais .....	3
2.1.1. Plano .....	3
2.1.2. Box .....	4
2.1.3. Sphere .....	4
2.1.4. Cone .....	5
2.1.5. Superfícies de Bezier .....	5
2.1.6. Torus .....	6
2.2. Cálculo das Coordenadas de Textura .....	6
2.2.1. Plano .....	6
2.2.2. Box .....	7
2.2.3. Sphere .....	7
2.2.4. Cone .....	8
2.2.5. Superfícies de Bezier .....	9
2.2.6. Torus .....	9
<b>3. Engine .....</b>	<b>9</b>
3.1. Parser XML .....	9
3.1.1. Estrutura Light .....	9
3.1.2. Estrutura Color e Model .....	10
3.2. Parser Ficheiros .3d .....	10
3.3. Extra (Câmera Terceira Pessoa) .....	10
3.4. Carregamento das diversas coordenadas .....	11
3.5. Setup das luzes .....	11
3.6. Setup das cores .....	12
3.7. Extra (Framerate) .....	12
<b>4. Conclusão .....</b>	<b>12</b>

# 1. Introdução

A quarta e última fase deste projeto, realizado no âmbito unidade curricular de Computação Gráfica, resumiu-se na progressão das tarefas implementadas nas fases anteriores, nomeadamente no *Generator* e no *Engine*.

As alterações introduzem no *Generator* a capacidade de gerar coordenadas de textura e normais para cada ponto das primitivas criadas previamente. O *Engine* deve ser capaz de ler e aplicar as normais e coordenadas de textura geradas pelo *Generator* para funcionalidades de iluminação e textura nos modelos. Assim, foi necessário alterar as estruturas de dados do *Engine* e o *parser* dos ficheiros utilizados para desenhar os modelos 3D. O parser para ficheiros xml será adaptado para que os modelos tenham textura ou cores de diferentes tipos e para que possa existir fontes de luz de vários tipos. Adicionalmente, criamos novas cenas de demonstração dinâmicas do Sistema Solar com luz e textura, de modo a demonstrar as novas atualizações.

As diferentes decisões e abordagens implementadas para o desenvolvimento das funcionalidades propostas serão detalhadas ao longo deste relatório.

## 2. Generator

Nesta fase do projeto, o *Generator* deve passar a gerar coordenadas de textura e normais para cada ponto (*vertex*) de uma primitiva. Assim, foi necessário adaptar a estrutura dos ficheiros .3d gerados para acomodar essa exigência.

A estrutura desses ficheiros passou a ser:

- A primeira linha possui o número total de pontos no ficheiro gerado;
- Uma linha por ponto que constroem a figura pedida. Cada ponto é agora constituído por 8 valores float, entre os quais, os 3 primeiros correspondem as coordenadas do ponto, os 3 seguintes são as coordenadas do vetor normal e os últimos 2 equivalem às coordenadas de textura do ponto. Os três grupos mencionados são separados por “;”, e as coordenadas dentro destes grupos são separadas por vírgula (“,”).

```
54
-0.5,0,-0.5;0,1,0;0,1
-0.5,0,-0.166667;0,1,0;0,0.666667
-0.166667,0,-0.166667;0,1,0;0.333333,0.666667
-0.166667,0,-0.166667;0,1,0;0.333333,0.666667
-0.166667,0,-0.5;0,1,0;0.333333,1
-0.5,0,-0.5;0,1,0;0,1
-0.166667,0,-0.5;0,1,0;0.333333,1
-0.166667,0,-0.166667;0,1,0;0.333333,0.666667
0.166667,0,-0.166667;0,1,0;0.666667,0.666667
0.166667,0,-0.166667;0,1,0;0.666667,0.666667
0.166667,0,-0.5;0,1,0;0.666667,1
-0.166667,0,-0.5;0,1,0;0.333333,1
```

Figura 1: Exemplo de um novo ficheiro .3d

### 2.1. Cálculo das Normais

#### 2.1.1. Plano

Partindo do pressuposto que o plano se encontra no plano XZ orientado no sentido positivo do eixo Y, a normal de todos os pontos da primitiva é (0, 1, 0).

### 2.1.2. Box

O cálculo das normais dos pontos de cada face da caixa segue uma estratégia semelhante à que foi utilizada no plano. Contudo, é preciso ter em atenção os eixos em que se enquadram e a sua orientação pois estes terão implicações nos vetores resultantes. Devem existir então 6 normais diferentes, tal como podemos observar na seguinte figura:

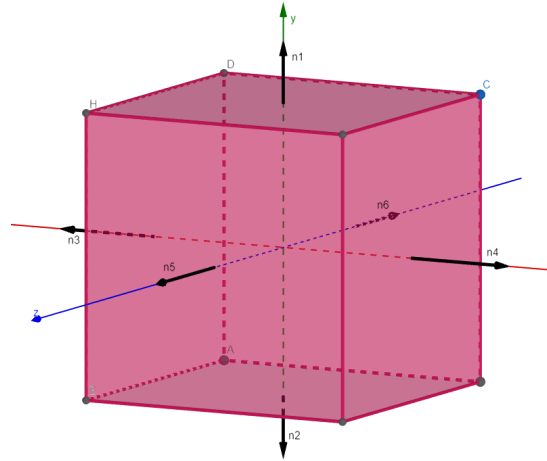


Figura 2: Vetores normais da caixa

As coordenadas das normais dos pontos das faces possuem então um dos valores seguintes:

- $\vec{n}_1 = (0, 1, 0)$  -> face de cima
- $\vec{n}_2 = (0, -1, 0)$  -> face de baixo
- $\vec{n}_3 = (-1, 0, 0)$  -> face da esquerda
- $\vec{n}_4 = (1, 0, 0)$  -> face da direita
- $\vec{n}_5 = (0, 0, 1)$  -> face da frente
- $\vec{n}_6 = (0, 0, -1)$  -> face de trás

### 2.1.3. Sphere

A normal de uma esfera num dado ponto da superfície é igual ao vetor normalizado que vai do centro da esfera até a esse ponto. A normalização é feita dividindo cada componente das coordenadas (x, y, z) pelo raio da esfera. Esta estratégia gera vetores como demonstra o exemplo a seguir.

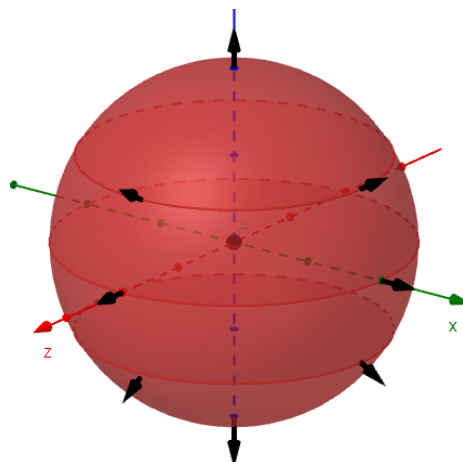


Figura 3: Vetores normais da esfera

### 2.1.4. Cone

O cálculo das normais para os pontos do cone está dividido em duas etapas: o cálculo das normais da base e o cálculo das normais das faces laterais.

Para todos os pontos da base, a normal terá coordenadas  $(0, -1, 0)$ , visto que a base do cone deve estar no plano XZ e, no total do cone, a base está virada para baixo.

Na obtenção dos vetores normal das faces laterais do cone, as componentes x e z do vetor normal são iguais ao valores de x e z dos pontos onde são calculadas, sendo possível obtê-las através das seguintes equações:

- $x = \cos(\text{slice} \times 2\pi / \text{slices})$
- $z = \sin(\text{slice} \times 2\pi / \text{slices})$

Já a componente y do vetor é possível calcula-la através das seguintes equações:

- $\frac{h}{r} = \tan(\alpha)$
- $\alpha = \text{atan}(\alpha)$
- $\beta = \pi - \alpha$
- $y = \sin(\beta)$

A seguinte figura demonstra como estas equações são aplicadas na prática:

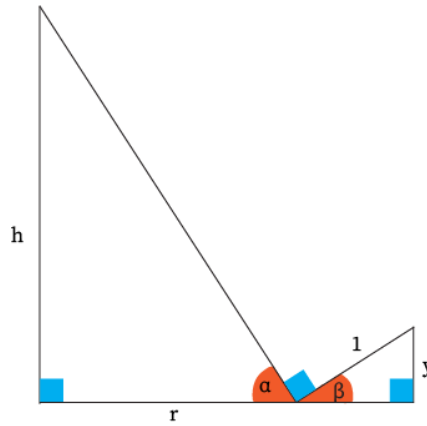


Figura 4: Cálculo da componente y da normal da superfície lateral do cone

### 2.1.5. Superfícies de Bezier

Para calcular as normais das superfícies de Bezier, recorreremos às seguintes equações:

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

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

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

Dados os valores de  $u$  e  $v$ , a normal de qualquer ponto é o resultado do produto interno dos vetores tangentes ao ponto, cujo cálculo se encontra a seguir:

$$\vec{u} = \frac{\partial p(u, v)}{\partial u}$$

$$\vec{v} = \frac{\partial p(u, v)}{\partial v}$$

$$\vec{n} = \vec{u} \times \vec{v}$$

O vetor  $\vec{n}$  representa, assim, o vetor normal para um dado ponto  $(u, v)$ .

### 2.1.6. Torus

Os vetores normal para pontos num torus são calculados assumindo uma lógica semelhante à usada para esferas, ou seja, dado os ângulos  $\alpha$  e  $\beta$  de um ponto calculamos a sua coordenada esférica. De seguida, esses pontos são normalizados para obter o vetor normal de cada um.

## 2.2. Cálculo das Coordenadas de Textura

### 2.2.1. Plano

Um plano é visto como uma matriz de dimensão *divisões*  $\times$  *divisões* em que cada elemento dessa matriz é um quadrado de lado *comprimento*/*divisões*. Essa matriz vai ser construída linha a linha, seguindo a ordem que refere a Figura 3.

O mapeamento das coordenadas de textura é feito diretamente, começando no canto superior esquerdo (0,1) e seguindo a mesma ordem de “construção” do plano, deslocando-se para os lados, quando avançar de linha ou coluna, uma distância de  $1/\text{divisões}$ . Deste modo, aplicamos um modelo *stretched* onde cobrimos todo o plano com uma textura, e os cantos da textura correspondem aos cantos do plano.

As figuras a seguir demonstram com mais clareza a correspondência entre as coordenadas de textura e as coordenadas dos pontos do plano.

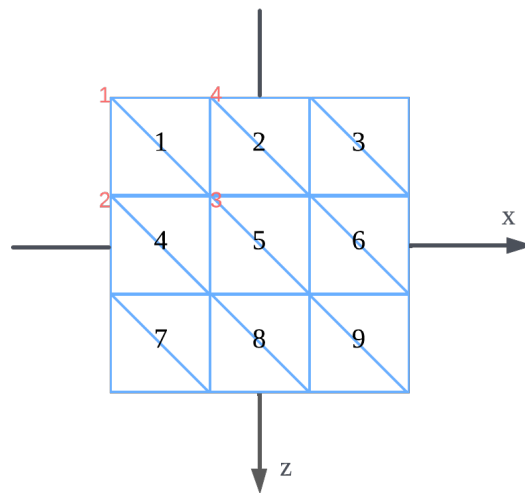


Figura 5: Desenho do plano

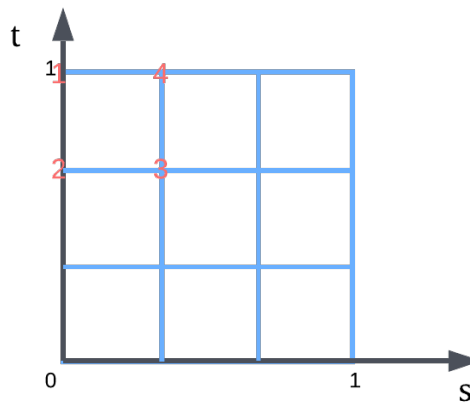


Figura 6: Coordenadas de textura

### 2.2.2. Box

O cálculo das coordenadas de textura da caixa segue um processo muito semelhante ao do plano, visto que a caixa é um conjunto de planos. Para cada face da caixa corresponde uma textura, ou seja a textura é aplicada 6 vezes, uma para cada face da caixa.

Contudo, para acomodar ao nosso método de construção da caixa e respeitar a orientação da textura e da face, foi preciso ter em atenção as coordenadas de textura que queríamos fazer corresponder inicialmente.

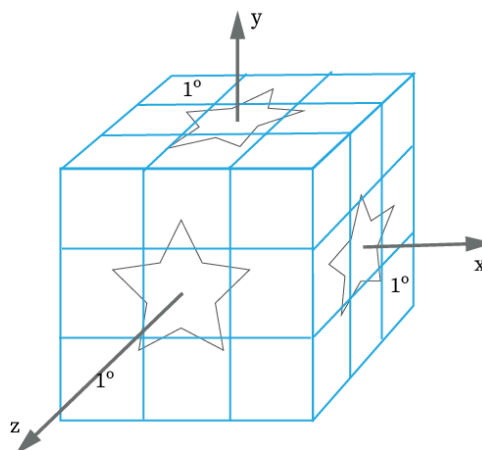


Figura 7: Desenho de uma caixa

A figura a cima indica qual seria o primeiro quadrado a ser construido para algumas faces da caixa. Como podemos observar, as primeiras coordenadas de textura raramente serão as mesmas, por isso tivemos que ter em consideração esse fator no mapeamento das coordenadas, alterando os valores iniciais quando necessário.

### 2.2.3. Sphere

Para calcular as coordenadas de textura para uma esfera, dividimos a imagem de textura pelo número de *slices* e *stacks* totais da esfera, cujas divisões na textura correspondem a  $1 / \text{slices}$  e  $1 / \text{stacks}$ , respetivamente, como demonstra a figura seguinte.

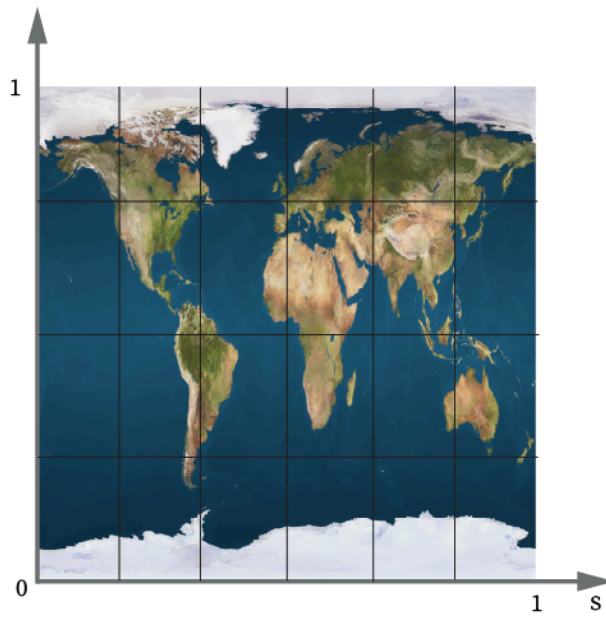


Figura 8: Divisão da textura para esfera de 6 slices e 4 stacks

Cada quadrado na textura será mapeado para uma divisão da esfera. Assim para cada divisão, cujo  $i$  é a *slice* e  $j$  é a *stack* onde nos encontramos, temos 4 coordenadas de textura e usá-mo-las :

- $s0 = i / slices$
- $s1 = (i + 1) / slices$
- $t0 = j / stacks$
- $t1 = (j + 1) / stacks$

#### 2.2.4. Cone

Para mapear as coordenadas de textura para os pontos do cone, o processo é dividido, novamente, em duas etapas: o cálculo para a base e o cálculo para as laterais.

Para as faces laterais do cone utilizamos um método muito semelhante ao da esfera, onde para cada divisão do cone corresponde uma divisão da textura, isto é, encontrando-nos numa *slice*  $i$  e na *stack*  $j$ , fazemos os seguintes cálculos para obter as coordenadas de textura necessárias para o quadrado que estamos a trabalhar.

- $s0 = i / slices$
- $s1 = (i + 1) / slices$
- $t0 = j / stacks$
- $t1 = (j + 1) / stacks$

A base segue um processo parecido, contudo, em vez de termos um número de divisões  $slices \times stacks$ , o mapeamento vai passar a ser feito por *slice* do cone, no qual cada divisão da imagem de textura corresponde a um triângulo de largura  $1 / slices$  e de altura 1. A figura seguinte demonstra como seria mapeado a textura para a base de um cone com 4 *slices*, onde cada triângulo apresentado corresponde a uma *slice* do cone.



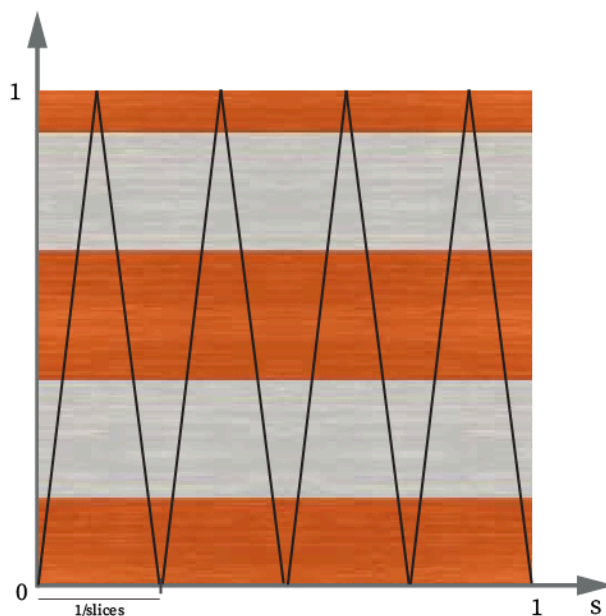


Figura 9: Divisão da textura para a base de um cone de 4 slices

### 2.2.5. Superfícies de Bezier

No mapeamento da textura para superfícies cúbicas de Bezier aplicamos a textura para cada *patch* existente na primitiva.

Assim, as coordenadas de textura são mapeadas diretamente com base nas coordenadas do *patch* que estamos a calcular. Portanto para um dado ponto  $(u, v)$ , corresponde a coordenada de textura  $(u, v)$ , seguindo o mesmo modelo usado para o cálculo das texturas na esfera.

### 2.2.6. Torus

O cálculo das coordenadas de textura para o torus é igual ao cálculo usado para a textura da esfera, ou seja, para cada quadrado do torus corresponde uma divisão imagem de textura fornecida.

## 3. Engine

### 3.1. Parser XML

Nesta nova fase, foram introduzidas novas informações aos cenários, desde luzes (Point, Directional e Spotlight) bem como a introdução de cores e de texturas aos modelos.

Assim, foram criadas novas estruturas e alteradas estruturas já existentes, de maneira a poder modelar toda a nova informação existente.

#### 3.1.1. Estrutura Light

Contém a informação do tipo de luz (através do enum `LightType`) bem como os valores das coordenadas de posição para luzes Point e Spotlight, coordenadas de direção para luzes Directional e Spotlight, e um cutoff apenas para as luzes Spotlight).

```
//Iluminação
enum LightType{
    POINT,
    DIRECTIONAL,
    SPOTLIGHT
};
struct Light {
    LightType type;
    float posX, posY, posZ; //usando em luzes point e spotlight
    float dirX, dirY, dirZ; //usando em luzes directional e spotlight
    float cutoff; //usando em luzes spotlight
};
std::list<Light> lights; //lista com as luzes presentes no cenário
```

Figura 10: Estruturas de Iluminação e Lista com as diversas Luzes

### 3.1.2. Estrutura Color e Model

Possui informação sobre a cor de uma determinada figura, onde um grupo tem então uma lista das figuras (Model) que lhe pertencem, tendo cada Model a informação do seu ficheiro .3d, da sua imagem de Textura (caso exista) e da sua estrutura Color.

A estrutura Color é inicializada por defeito com valores 200 para difusão, 50 para ambiente, e 0 para os restantes campos, sendo substituídos durante o parse do xml caso exista informação sobre as suas cores.

```
struct Color { //cada campo tem um valor default caso o modelo não tenha cor no xml
    float diffuseR = 200, diffuseG = 200, diffuseB = 200;
    float ambientR = 50, ambientG = 50, ambientB = 50;
    float specularR = 0, specularG = 0, specularB = 0;
    float emissiveR = 0, emissiveG = 0, emissiveB = 0;
    float shininessValue = 0;
};
struct Model {
    std::string modelFile, textureFile;
    Color color;
};
struct Group {
    std::list<Transformation> transformations;
    std::list<Model> models;
    std::list<Group> children;
};
```

Figura 11: Estruturas de Cores e Modelos

## 3.2. Parser Ficheiros .3d

Nesta fase, devido à introdução das normais e texturas, o parse dos ficheiros .3d foi realizado através de duas fases, onde numa primeira cria-se tokens através do parse pelo ponto e vírgula, criando 3 tokens por linha, sendo eles os tokens de coordenadas, normais e texturas.

Numa segunda fase, faz-se o parse pela vírgula dos tokens previamente criados, tirando assim a informação de X, Y e Z para as coordenadas do ponto e do vetor normal e de s e t para as texturas.

```
// Leia o arquivo linha por linha
while (std::getline(file, linha)) {
    std::istringstream iss(linha);
    std::string tokenSemiColon;

    /* As linhas são do estilo: coordenadas;normais;texturas
    Cada um dos componentes tem 3 ou 2 valores, separados por vírgulas */
    int tokenCount = 0;
    while (std::getline(iss, tokenSemiColon, ';')) {
        std::istringstream issSemiColon(tokenSemiColon);
        std::string tokenComa;

        // Divide tokenSemiColon por vírgula
        while (std::getline(issSemiColon, tokenComa, ',')) {
            float value = std::stof(tokenComa);
```

Figura 12: Parse Ficheiros .3d

## 3.3. Extra (Câmera Terceira Pessoa)





Durante a segunda fase deste projeto passou a ser possível movimentar a câmara em modo exploração, isto é, com movimentos orbitais nas coordenadas esféricas. No entanto, a fim de dar ainda mais liberdade ao modelo de visualização, decidimos passar a ter dois modos: SPHERICAL e FIRSTPERSON.

Ao iniciar a visualização de qualquer modelo, a câmera encontra-se sempre no modo SPHERICAL, e é possível alterar de um modo para o outro com a tecla “C”. Quando se pretende alterar para o modo FIRSTPERSON é necessário calcular as coordenadas do ponto para onde a câmera está a olhar (LookAt) através das seguintes fórmulas:

- $\text{camLookAtX} = \text{camPosX} + \cos(\beta) * \sin(\alpha);$
- $\text{camLookAtY} = \text{camPosY} + \sin(\beta);$
- $\text{camLookAtZ} = \text{camPosZ} + \cos(\beta) * \cos(\alpha);$

Nestas equações, o “camPos...” representa a posição da câmera e os ângulos  $\alpha$  e  $\beta$  determinam a posição nas coordenadas esféricas.

Os comandos de manipulação permitidos no modo FIRSTPERSON são:

-  - Deslocar para a frente
-  - Deslocar para trás
-  - Deslocar para a direita
-  - Deslocar para a esquerda
- **F1** - Deslocar para cima (eixo Y)
- **F2** - Deslocar para baixo (eixo Y)
- **A** - Virar câmera para esquerda
- **D** - Virar câmera para direita
- **W** - Virar câmera para cima
- **S** - Virar câmera para baixo

Para alterar o modo de renderizar os polígonos:

- **F** - Muda o modo para GL\_FILL
- **L** - Muda o modo para GL\_LINE
- **P** - Muda o modo para GL\_POINT

Ao alterar o modo de FIRSTPERSON para SPHERICAL consideramos as coordenadas do *LookAt* como 0 por questões de simplificar a implementação e os cálculos.

### 3.4. Carregamento das diversas coordenadas

Nesta fase foram criados 3 buffers ao invés de apenas um. O primeiro buffer contém as coordenadas dos pontos do objeto a ser desenhado, o segundo contém as coordenadas das normais ao objeto e o último contém as coordenadas das texturas.

Com estas informações, podemos fazer bidding com os respectivos buffers e assim carregar as informações para o nosso programa. Relativamente às texturas, cada modelo foi associado à imagem de textura correspondente para podermos então fazer a associação entre as coordenadas do objeto e as de textura.

### 3.5. Setup das luzes

Por cada luz introduzida no XML, criou-se uma luz no programa com as informações contidas nos ficheiros XML. As luzes foram desenhadas antes da configuração da câmara. Há de ressaltar que mesmo que o ficheiro XML não contenha nenhuma luz, os objetos serão observáveis, uma vez que existe uma componente que luz ambiente que foi introduzida por default.

### 3.6. Setup das cores

Para os objetos que continham cores, tivemos de configurar várias propriedades de material e luz para posterior renderização gráfica. Esses parâmetros de setup foram retirados dos ficheiros XML e continham parâmetros como as componentes de luz difusa, especular e emissiva e o brilho.

### 3.7. Extra (Framerate)

Ao utilizar o *Engine*, o título da janela vai possuir informação extra relativamente ao *frames per second* (FPS - frames por segundo) para podermos analisar o desempenho dos modelos.

## 4. Conclusão

Nesta quarta e última fase deste trabalho, fomos capazes de consolidar os conceitos de textura e iluminação aprendidos durante as aulas teóricas e práticas, que era o objetivo deste quarto guião.

Durante o desenvolvimento de todo o projeto, o grupo deparou-se com inúmeros obstáculos e dificuldades, apesar disso sentimos que fomos capazes de superar a grande maioria desses, não sendo apenas possível aplicar várias texturas simultaneamente por falta de tempo, apesar de reconhecermos que esse mesmo erro facilmente se resolveria apenas com um bocado mais de tempo. Reconhecemos também que há bastante espaço para melhoria e otimização do trabalho, sendo esse o foco principal do trabalho futuro, assim como acrescentar mais componentes ao nosso Sistema e outras texturas, como por exemplo temáticas. O nosso trabalho foi entregue sem a possibilidade de renderizar objetos com diferentes texturas (aceita somente uma). Apesar de a determinado momento termos conseguido implementar a funcionalidade pedida, por vezes o programa fechava quando era corrido, pelo que decidimos entregar a versão anterior. Infelizmente não tivemos tempo de realizar a correção pelo que o nosso trabalho futuro, no que diz respeito a este trabalho, passaria por corrigir esse erro.

Quanto ao projeto final, estamos satisfeitos com o resultado obtido, implementando o que era proposto assim como alguns extras realizados ao longo das quatro fases, considerando assim que alcançamos um bom trabalho. Além disso, sentimos também que, graças ao mesmo, consolidamos melhor os conhecimentos de Computação Gráfica pretendidos pelos docentes, principalmente na vertente prática.