

Graphical primitives - Phase 4

Miguel Oliveira, Nelson Faria, José Rodrigues, and Filipe Costa (Grupo 46)

Universidade do Minho, Departamento de Informática, 4710-057 Braga, Portugal
e-mail: {a83819,a84727,a85501,a85616}@alunos.uminho.pt

1 Introdução

Nesta quarta fase do projeto, os principais objetivos eram:

- aprimorar o **Generator** e fazer com que ele gere ficheiros.3d com os pontos dos vértices dos triângulos, as normais e as coordenadas de textura, para construir os pontos a serem interpretados pelo **Engine**;
- fazer uma extensão à aplicação engine, de modo a suportar texturas e iluminação. Por outras palavras, devemos ser capazes de adicionar aos modelos a serem desenhados a capacidade de apresentarem uma textura, através de uma imagem, bem como a iluminação dos objetos a serem desenhados, de forma a serem apresentadas as sombras que correspondem aos respetivos objetos.

2 Generator

O **Generator** foi uma peça necessária a ser evoluída nesta 4.^a fase do projeto de Computação Gráfica. Muito resumidamente, o objetivo era:

1. Acrescentar as normais e as coordenadas de textura em todos os objetos que são possíveis gerar, nomeadamente nos objetos que são possíveis gerar a partir de patches de Bezier

2.1 Adição das Normais

As normais acrescentadas aos diferentes objetos são importantes para depois no **Engine** poder-se acrescentar a iluminação. Deste modo, foram acrescentadas as normais para cada vértice no plano, na caixa, na esfera, no cone, na coroa circular (que nós desenvolvemos a mais) e no objeto que se pode desenvolver a partir de patches de Bezier, que no nosso caso é o teapot.

Por exemplo, para o caso do plano foi fácil o cálculo das normais, uma vez que se o plano está acento no plano xz do eixo de coordenadas, as normais a todos os vértices é o vetor $\vec{n} = (0, 1, 0)$. Já na caixa, por exemplo, o raciocínio é idêntico, com a nuance de agora os diferentes planos que constituem uma caixa terem normais diferentes, consoante a sua posição. Na caixa, existem sempre dois planos paralelos aos planos do eixo de coordenadas, sendo que a única diferença entre esses dois planos paralelos é o sentido da normal a esse plano, sendo simétricos.

No caso da esfera, que era dos casos mais importantes para o nosso exemplo, as normais foram calculadas para cada vértice tendo em conta que a esfera tem centro na origem. Deste modo, a normal a cada vértice da esfera é o vetor normalizado \vec{AB} , em que \vec{A} é a origem e \vec{B} o vértice que se encontra no corpo da esfera. Na figura seguinte tentou-se explicar o raciocínio por detrás disto:

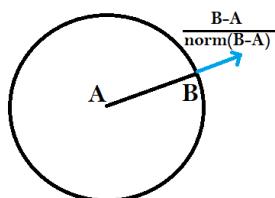


Figura 1. Esquema do cálculo da normal para a esfera

Quanto aos patches de Bezier, as coisas já tem de ser um pouco diferentes, visto que foi necessário usar-se as derivadas parciais para se poder obter a normal. Assim, visto que na outra fase já se mostrou os cálculos das posições num patch de Bezier, agora mostramos os cálculos das derivadas parciais.

Na figura seguinte mostra-se um esquema com as derivadas parciais e como é que é obtida a normal ao vértice em questão.

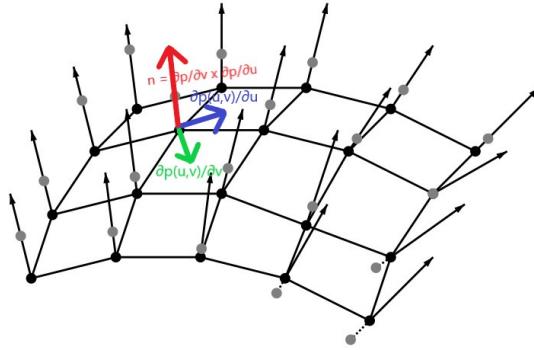


Figura 2. Esquema do cálculo da normal para uma superfície de Bezier

Na figura, pode-se visualizar as derivadas parciais em ordem a v e em ordem a u e a forma como a partir do produto vetorial entre os dois vetores das derivadas se pode obter a normal a um determinado ponto na superfície.

Assim, a partir das fórmulas das derivadas parciais em forma matricial, foi possível calcular as derivadas para cada ponto, não esquecendo que era preciso usar as fórmulas para calcular as componentes x, y e z das derivadas. As fórmulas usadas foram as seguintes:

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

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

Assim, depois de calculadas as derivadas, foi só necessário usar esses valores para calcular o produto externo entre o vetor \vec{v} (vetor da derivada parcial em ordem a v) e o \vec{u} (vetor da derivada parcial em ordem a u).

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

Deste modo, obteve-se os vetores normais para todos os vértices, sendo só necessários escrevê-los depois no ficheiro de output.

2.2 Adição das Coordenadas de Textura

Quanto às coordenadas de Textura, foi primeiro necessário perceber como é que se organizavam as coordenadas de Textura aquando da leitura posteriormente no Engine.

Para começar, as coordenadas de Textura são a 2D, sendo que o ponto (0,0) inicialmente encontra-se no canto superior esquerdo, mas depois no Engine, através de umas pequenas funções que são usadas na leitura, o ponto (0,0) passa a ficar no canto inferior esquerdo da imagem. Seguidamente é importante perceber que as coordenadas de textura variam entre [0,1] nas componentes x e y do seu referencial, pelo

que o cálculo das coordenadas de textura para cada objeto teve isso em conta, pelo que não foi muito difícil a introdução dessas coordenadas no ficheiro.

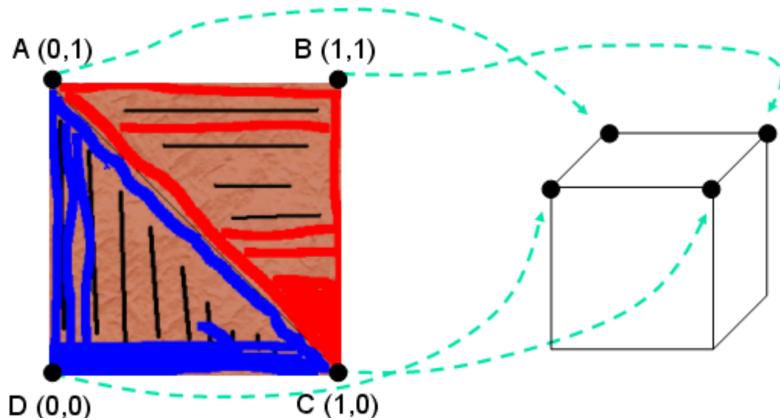


Figura 3. Esquema das coordenadas de Textura para o caso de uma caixa

Nesta última imagem, é possível observar o caso da caixa e como, para cada face da caixa vão ser mapeadas as coordenadas de Textura.

Como a esfera foi aquilo que nós realmente usamos para as coordenadas de Textura, vamos falar um pouco de como é que nós procedemos ao seu cálculo. Primeiramente é de referir que a imagem que temos para a Textura é a de um quadrado, que varia conforme o que foi dito anteriormente. Deste modo, a partir do ciclo que implementamos para o cálculo das posições e das normais, aproveitamos para calcular também as coordenadas de textura, sendo que para cada vértice da esfera, a sua coordenada de textura está no intervalo de [0,1].

Como uma esfera varia através de slices e stacks, a coordenada de Textura de um vértice pode ser atribuída começando em 0 e iterativamente indo até ao número de slices(no caso do x)/stacks(no caso do y). Deste modo, as coordenadas nunca ultrapassam 1. De seguida, pode ver-se um excerto de código com a ideia principal da atribuição de coordenadas numa esfera:

```
x = (float) j / slices;
y = (float) i / stacks;
```

Nesta imagem podemos ver como fica o resultado final após a aplicação das coordenadas de textura na esfera:

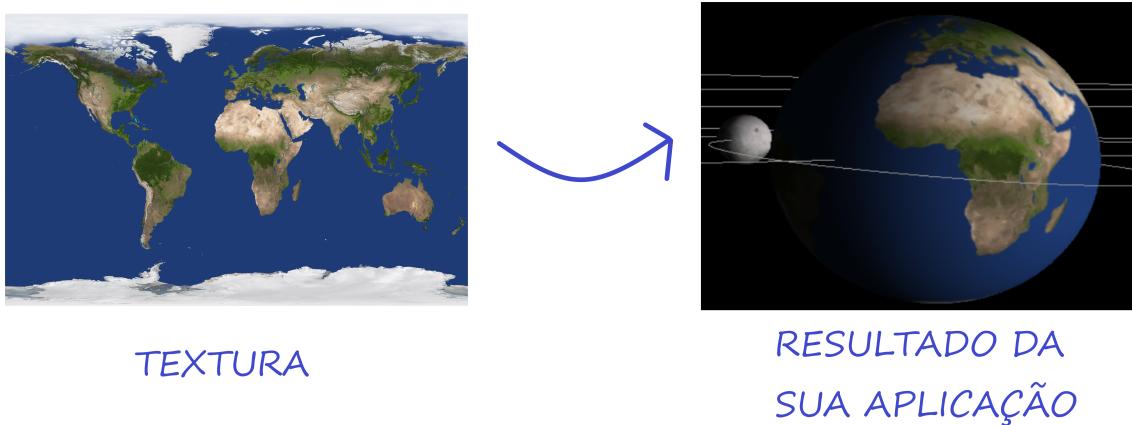


Figura 4. Resultado da aplicação da textura da Terra numa esfera

2.3 Estrutura do ficheiro Output

Quanto à estrutura do ficheiro de output, ele passa a estar organizado de maneira diferente daquilo que temos feito até agora. Deste modo, o que é necessário fazer é passar a guardar as coordenadas das normais e das texturas de cada vértice. Assim, na figura seguinte exemplifica-se a estrutura do ficheiro que será o resultado de uma geração de um certo objeto:

```

1 1800 — Número total de vértices
2 -0.000000 -5.000000 -0.000000 — posição do vértice
3 -0.000000 -1.000000 -0.000000 — normal do vértice
4 0.000000 0.000000 — coordenada de textura do vértice
5 0.908178 -4.755282 1.250000
6 0.181636 -0.951056 0.250000
7 0.100000 0.100000
8 0.000000 -4.755282 1.545085
9 0.000000 -0.951056 0.309017
10 0.000000 0.100000
11 0.908178 -4.755282 1.250000
12 0.181636 -0.951056 0.250000
13 0.100000 0.100000
14 -0.000000 -5.000000 -0.000000
15 -0.000000 -1.000000 -0.000000
16 0.000000 0.000000
17 -0.000000 -5.000000 -0.000000
18 -0.000000 -1.000000 -0.000000
19 0.100000 0.000000
20 -0.000000 -5.000000 -0.000000
21 -0.000000 -1.000000 -0.000000
22 0.100000 0.000000

```

Figura 5. Estrutura do ficheiro de output do Generator

3 Ficheiro XML

O ficheiro xml desta fase foi uma atualização do ficheiro **xml** da fase 3 do trabalho prático. Assim apenas foram adicionados novos *"comandos"* ao xml de modo a cumprir os objetivos desta fase, nomeadamente a noção de iluminação, onde podemos ter diferentes tipos de iluminação, como por exemplo a iluminação difusa e emissiva, e também a inserção de ficheiros de texturas para uma melhor aproximação à realidade, no contexto do Sistema Solar.

```
<group>
    <rotate angle="0.5" axisX="1" axisY="0" axisZ="0" />
    <translate time="48.7008" >
        <point X="-195.3" Y="0" Z="0" />
        <point X="-138.1" Y="0" Z="-138.1" />
        <point X="0" Y="0" Z="-195.3" />
        <point X="138.1" Y="0" Z="-138.1" />
        <point X="195.3" Y="0" Z="0" />
        <point X="138.1" Y="0" Z="138.1" />
        <point X="0" Y="0" Z="195.3" />
        <point X="-138.1" Y="0" Z="138.1" />
    </translate>
</group>
<group>
    <rotate time=10 axisX=0 axisY=1 axisZ=0 />
    <models>
        <model file='terra.3d' texture='textTerra.jpg' diffR='1.0' diffG='1.0' diffB='1.0' ambR='0.2' ambG='0.2' ambB='0.2' />
    </models>
</group>
<group>
    <rotate angle="5" axisX="1" axisY="0" axisZ="0" />
    <translate time="7.84" >
        <point X="-7" Y="0" Z="0" />
        <point X="-4.95" Y="0" Z="-4.95" />
        <point X="0" Y="0" Z="-7" />
        <point X="4.95" Y="0" Z="-4.95" />
        <point X="7" Y="0" Z="0" />
        <point X="4.95" Y="0" Z="4.95" />
        <point X="0" Y="0" Z="7" />
        <point X="-4.95" Y="0" Z="4.95" />
    </translate>
    <models>
        <model file='terraLua.3d' texture='textLuaTerra.jpg' diffR='1.0' diffG='1.0' diffB='1.0' ambR='0.2' ambG='0.2' ambB='0.2' />
    </models>
</group>
</group>
</group>
```

Figura 6. Excerto do XML para a representação do planeta Terra

Como se pode ver na figura acima, foram usadas tags de **texture**, bem como tags para a utilização de **iluminação**, de forma a utilizar várias formas de fazer. Assim, foram usadas as componentes **difusa**, **ambiente** e **emissiva**. Relativamente às texturas, foi usada a tag **textura**, onde é lido um ficheiro que contém as texturas do objeto associado a esta mesma tags, de forma a uma melhor aproximação à realidade. Em relação à iluminação, forma usadas várias componentes para que a iluminação do projeto têm uma perspetiva mais realista, de modo a possuir sombras e zonas de luz, o que traz uma melhor visão à simulação criada.

De seguida é possível ver o estado final do nosso Sistema Solar, com todos os corpos celestes que constituem o mesmo, como é o objetivo desta fase.

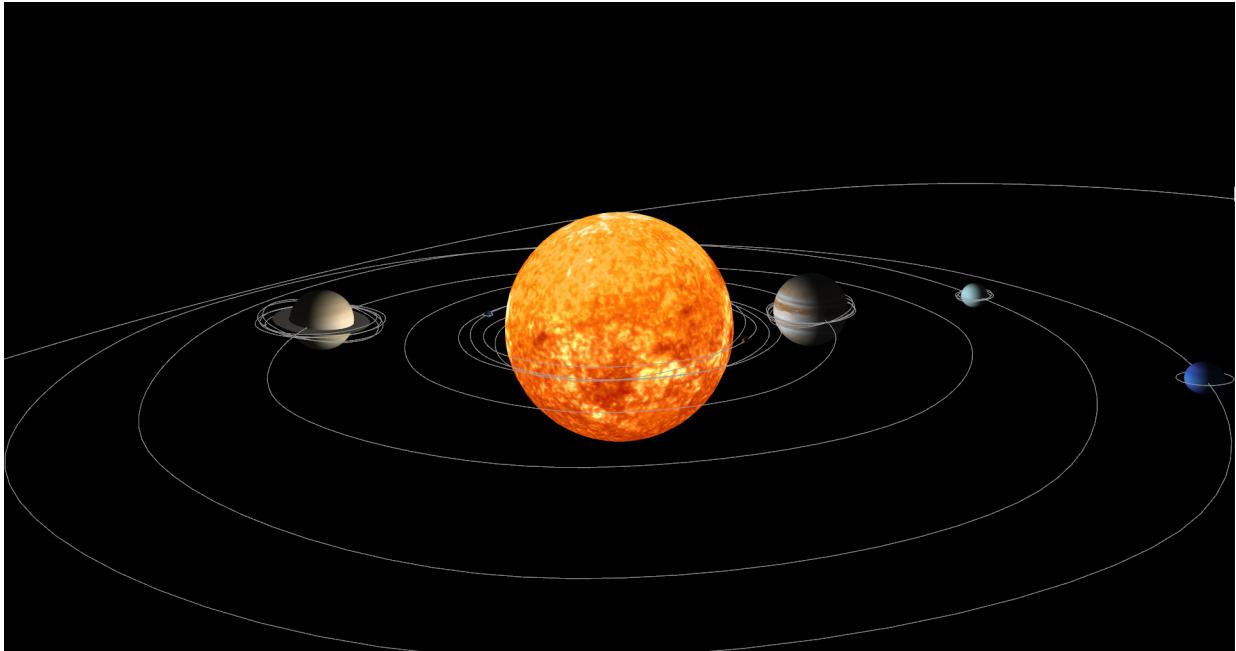


Figura 7. Representação do Sistema Solar

4 Engine

4.1 Breve introdução

Nesta fase de desenvolvimento da aplicação engine, era-nos pedida a capacidade de suportar a existência de luzes e, para além disto, a introdução da possibilidade de aplicar uma determinada textura para um modelo em específico, textura essa presente num ficheiro de extensão jpg (imagem).

Para ambos os *improvements* foi necessário ter em conta a alteração feita ao programa generator, uma vez que, a partir de agora, este, no ficheiro de output de um modelo, especifica, para além dos vértices que formam os triângulos para o desenho da figura, as suas normais (para efeitos de reflexão de luz) e ainda as suas coordenadas de textura.

Para tal foram necessárias algumas alterações à estrutura de dados que armazena a informação acerca dos groups lidos a partir do ficheiro xml, nomeadamente, na capacidade de especificação acerca de vetores normais a cada vértice, coordenadas de textura, especificação do material do próprio model, entre outros.

Mais uma vez, a alteração face à entrega anterior, começa no processamento do ficheiro xml de entrada. Agora somos capazes de reconhecer um elemento "lights" especificada antes de qualquer elemento do tipo "group". É neste campo que criamos as estruturas necessárias para ser possível o suporte de luz. Essa estrutura não é mais que um *vector<float>** que representa as posições de cada uma das luzes. De 4 em 4 é guardada informação acerca da posição de uma nova luz. Nota para o facto de que, a partir deste momento, para definir a textura de um determinado modelo podemos especificar o respetivo ficheiro que contém a textura e ainda definir cada uma das suas componentes de reflexão da luz (diffuse, specular, ambient, emission).

4.2 Alteração da estrutura de dados Group

No que toca à estrutura de dados Group, optamos por alterar um pouco o paradigma, face aos requisitos pedidos para esta fase. Anteriormente, colocavamos todos os vértices de um group a desenhar numa

mesma estrutura de dados comum, visto que não haveria distinção quanto às suas texturas, material e características relacionadas com a reflexão da luz.

Ora, de agora em diante, passamos a definir um group como sendo constituído por uma "lista" de operações (translate, rotate e scale) e ainda uma lista de Models. De uma forma muito simples, cada um dos models a desenhar dentro de cada group será afetado por cada uma das operações desse mesmo group.

Um Model é composto pelos vértices que formam os triângulos que darão forma ao model, as normais do objeto em cada um desses vértices e ainda informações acerca da sua composição, nomeadamente a sua textura (se aplicável) e as suas características face à existência de fontes luminosas.

Apresentamos de seguida a nova definição da estrutura de dados Group e ainda a definição de um Model:

```
struct group {
    Operacao* op;
    int sizeOp;
    int numOps;
    vector<Model>* models;
};

struct model {
    composicao c;
    vector<float>* vertices;
    vector<float>* normals;
};
```

Por sua vez, uma composição é uma estrutura composta por uma variável *Material* e um outra designada de *Texture*, também apresentadas no excerto de código seguinte.

```
typedef struct texture {
    int id;
    int buffer_id;
    vector<float>* textureCoord;
} *Texture;

typedef struct material {
    float diffuse[4];
    float specular[4];
    float ambient[4];
    float emission[4];
    int shininess;
} *Material;
```

Basicamente, um Texture é constituída por um valor inteiro designado por id que representa o id atribuído no momento em que ocorreu a leitura do ficheiro de textura do Model respetivo, aquando do processamento do ficheiro xml de entrada, um outro id que corresponde ao índice do id das coordenadas de textura aquando da inicialização do vbo na estrutura de dados usada para guardar todos esses id's para cada uma das texturas usadas num determinado momento e ainda as próprias coordenadas de textura para o Model em si.

Já o Material, apenas armazena 4 vetores com 4 posições que correspondem aos valores de cada uma das componentes relacionadas com a reflexão da luz e ainda um inteiro shininess que representa a constante da componente especular. Deixamos nas figuras seguintes a representação de cada uma dessas estruturas.

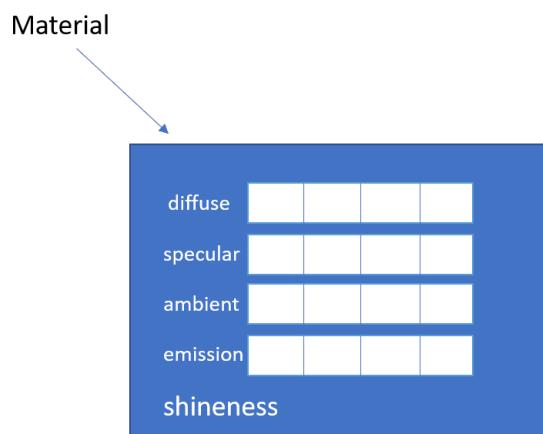


Figura 8. Representação da estrutura de dados Material

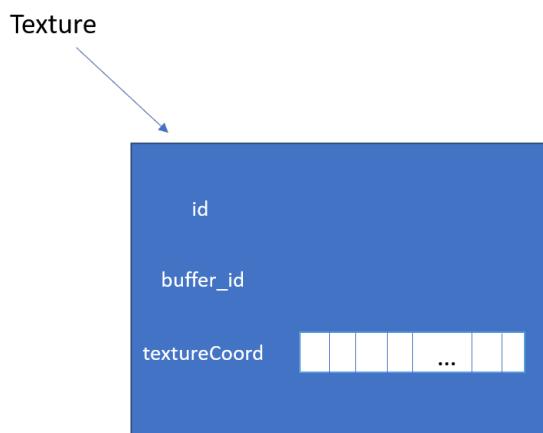


Figura 9. Representação da estrutura de dados Texture

Com estas alterações à estrutura de dados Group passamos a conseguir caracterizar cada um dos models dentro de cada group como sendo ”independente” de cada um dos outros, sendo capazes de atribuir a cada um deles, uma textura e características relacionadas com a iluminação diferentes.

Deixamos uma nota, tal como em cada um dos relatórios de entrega de cada uma das fases anteriores, para o facto de que um group é uma estrutura que guarda um conjunto de modelos descritos dentro de um elemento caracterizado pela tag ”models” dentro de um group e ainda todas as suas transformações, incluindo aquelas que herda pelo facto de estar incluído dentro de um ou mais *elements* com a mesma tag group.

4.3 Implementação da iluminação

Como já foi brevemente referido acima, neste momento, a nossa aplicação engine é capaz de suportar iluminação, caso especifiquemos no ficheiro xml de input a posição de cada uma dessas luzes. Para cada uma delas, as suas coordenadas serão guardadas no vector *lights*.

Para colocar luzes no cenário é necessário inicializá-las primeiro, atribuindo-lhe uma cor. Neste momento o nosso engine coloca a cor das luzes adicionadas como sendo branca.

No que toca ao seu ”desenho”, é necessário a cada iteração do openGL colocar a luz na respetiva posição (caso se trate de um ponto de luz), posição essa que está registada no vector criado para o efeito (*lights*).

De modo a que os objetos desenhados interajam com a luz emitida, é necessário conhecer para cada um deles as normais de cada um dos vértices utilizados que compõem os triângulos que por sua vez compõem o próprio modelo em si. Mais uma vez, esses dados são-nos fornecidos aquando da geração dos ficheiros que contêm informação para a representação dos models. Para além disto os objetos necessitam de ter um material associado, de modo a poder refletir a luz. Esses parâmetros são especificados no ficheiro de xml de entrada no element ”model”. Segue-se um exemplo:

```
<models>
    <model file='urano.3d' texture='textUrano.jpg' diffR='1.0' diffG='1.0'
           diffB='1.0' ambR='0.2' ambG='0.2' ambB='0.2' />
</models>
```

Para implementar o material de um modelo é necessário, a cada iteração do motor do openGL, caracterizar o objeto em cada uma destas componentes: **diffuse**, **specular**, **emission** e **ambient**. Para tal, na altura de desenhar os models de um determinado group, accedemos a cada um dos models contidos nesse group, assim como à sua especificação relacionada com a interação com a luz e aplicámos-lhe essas mesmas características.

Este elemento representa um objeto cuja componente difusa é (1.0, 1.0, 1.0, 1.0), ao passo que a sua componente ambiente é (0.2, 0.2, 0.2, 1.0), sendo todas as outras (emission e specular) atribuído o valor (0.0, 0.0, 0.0, 1.0).

4.4 Implementação das texturas

No que toca a implementação das texturas foi necessário adicionar alguma funcionalidade um pouco diferente, nomeadamente no carregamento das imagens que serão associadas à textura de um determinado objeto.

Assim sendo, a primeira tarefa a realizar no sentido de garantir a funcionalidade de aplicação de texturas a models passa pelo carregamento dessas mesmas texturas representadas em imagem para memória gráfica, sendo a cada uma delas atribuído um id com o qual a respetiva imagem ficou registada em memória gráfica. Esse id será guardado na respetiva estrutura que representa o model para mais tarde lhe podermos associar a textura correta.

Aquando da inicialização é necessário fazer a inicialização de um vbo para armazenar as coordenadas de textura em memória gráfica. Essas coordenadas de textura associadas a um objeto em específico encontram-se especificadas na estrutura model respetiva dentro do group no qual este está inserido, mais especificamente na variável composicao.

Para no desenho se fazer ver essa textura é necessário associar a imagem que corresponde ao respetivo model, o que não é difícil, tendo em conta que aquando do carregamento dessa imagem para memória

gráfica lhe foi atribuído um id único que foi armazenado na própria estrutura de dados. De seguida é desenhado o modelo com base nos vértices dos triângulos que dão forma ao próprio model, nas normais em cada um desses vértices e ainda nas coordenadas de textura, estando claro associada, a respetiva imagem representativa da textura do objeto.

É de salientar o facto de que não existe qualquer obrigatoriedade em atribuir uma textura a um objeto. É possível simplesmente atribuir-lhe uma cor aquando da especificação das suas características face à existência de uma fonte luminosa (através de `glMaterialfv()`).

4.5 Desenho dos modelos

Apresentamos de seguida o algorítmo base para o desenho dos modelos, tendo já sido previamente feita a inicialização das estruturas de dados para o efeito, etc. Este algorítmo é o mesmo apresentado no relatório da fase anterior, sendo que agora, para esta fase, a única alteração vem no algorítmo de desenho dos models, que será apresentado também de seguida.

Algorítmo base:

PARAMETRO FORNECIDO: time

```
- Percorrer cada uma das estruturas group
{
    glPushMatrix();
    - Percorrer cada uma das operações do group
    {
        - SE translação dinâmica
        - getGlobalCatmullRomPoint(group, operation, time, position, deriv);
        - glTranslatef(position[0], position[1], position[2]);
        - atualizarEixos(group);
        - SE translação ñ dinâmica
        - glTranslatef(*paramsDaTranslacao*);
        - SE rotação dinâmica
        - glRotatef(getCurrentAngle(time, group, operation), *paramsDaRotacao*);
        - SE rotação ñ dinâmica
        - glRotatef(*paramsDaRotacao*);
        - SE scale
        - glScalef(*paramsDeScale*);
        - SE color
        - glColor3f(*paramsDeColor*);
    }
    desenharModelos();
    glPopMatrix();
}
```

A função no nosso código que implementa o algorítmo especificado no pseudo-código apresentado acima é designada de `repositionModels(float gt)`.

De seguida mostramos o excerto de código que, de forma algo simplicista, ilustra o nosso algorítmo de desenho de um model:

-> Seja `_model` o model a desenhar

```
getMaterialInfo(_model, diffuse, specular, ambient, emission, &shineness);
```

```
/* Definimos o material para o modelo */
```

```

glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, specular);
glMaterialfv(GL_FRONT, GL_AMBIENT, ambient);
glMaterialfv(GL_FRONT, GL_EMISSION, emission);
glMaterialf(GL_FRONT, GL_SHININESS, shininess);

/* Estamos perante um model que possui
um textura associada */
if (hasTexture(_model) {

    /* Associamos a respetiva imagem ao model */
    glBindTexture(GL_TEXTURE_2D, getTextureId(_model));

    glBindBuffer(GL_ARRAY_BUFFER, texCoord[getBufferId(_model)]);
    glTexCoordPointer(2, GL_FLOAT, 0, 0);
}

/* Desenhamos os respetivos vértices */
glBindBuffer(GL_ARRAY_BUFFER, ...);
glVertexPointer(3, GL_FLOAT, 0, 0);

glBindBuffer(GL_ARRAY_BUFFER, ...);
glNormalPointer(GL_FLOAT, 0, 0);

glDrawArrays(GL_TRIANGLES, 0, ...);

glBindTexture(GL_TEXTURE_2D, 0);

```

4.6 Validação do ficheiro XML de input

Uma das tarefas que também levamos a cabo nesta fase foi a validação do ficheiro xml que server como input para o nosso engine. Não faria sentido não fazer nenhum tipo de validação, pois poderíamos estar a tentar processar um ficheiro xml que não fosse minimamente ao encontro daquele que especificasse uma cena válida para desenhar.

O processo usado para validar o xml foi o seguinte:

- Em primeiro lugar verificar se existe o elemento principal do ficheiro que englobará todos os outros: **scene**.
- Para cada filho deste elemento principal apenas poderemos encontrar um element ”lights” e poderemos encontrar um ou mais elements ”group”.
- Dentro do lights apenas podemos encontrar subelements do tipo ”light” caso contrário o formato do ficheiro é inválido.
- Dentro de um group podemos encontrar no máximo 1 element de operação de cada tipo (translate, rotate, scale), uma única tag ”models” e um número variável de outros elements ”group” para os quais aplicamos o mesmo algorítmo de validação escrito neste tópico.
- Dentro de um element ”models” apenas poderemos encontrar subelements do tipo ”model” caso contrário temos um ficheiro errado.

5 Conclusão

Tendo em conta os objetivos principais desta quarta fase do trabalho prático, consideramos ter conseguido um bom desempenho na elaboração de uma proposta de solução para o problema proposto na mesma, nomeadamente com todas as questões relacionadas com as texturas e a iluminação dos objetos, funcionalidade extra implementada no engine, de modo a representar a texturas dos planetas, bem como a iluminação dos mesmos, possuindo assim pontos de luz e de sombra, para uma melhor representação do Sistema Solar.

Assim, conseguimos desenvolver todos os objetivos desta fase, criando os respetivos mecanismos para a elaboração da mesma, de modo a obter um resultado que esteja dentro dos objetivos desta quarta fase e deste trabalho.