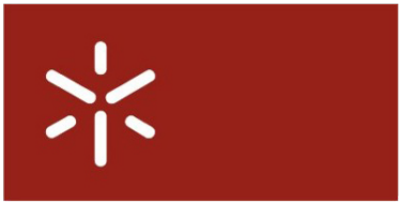

Computação Gráfica

Mestrado Integrado em Engenharia Informática

Fase 4

Normais e Coordenadas de textura

GRUPO 19 - PL2

A85227 João Pedro Rodrigues Azevedo

A85729 Paulo Jorge da Silva Araújo

A83719 Pedro Filipe Costa Machado

A89983 Paulo Jorge Moreira Lima

Braga
Maio 2020

Conteúdo

Capítulo 1

Introdução

1.1 Contextualização

Este relatório é relativo ao trabalho prático da UC de **Computação Gráfica** e tem como objetivo final desenvolver um *engine* para a resolução de modelos **3D**, mostrando o seu potencial através da apresentação de um ficheiro de **configuração xml** capaz de reproduzir um Sistema Solar[1], **nesta fase**, com texturas, luzes e o movimento dos planetas pelas suas órbitas e em torno do seu próprio eixo.

1.2 Objetivos para a fase 4

Esta última fase trouxe, no caso do gerador, as coordenadas de textura e as respetivas normais para cada vértice das primitivas introduzidas na 1ª fase, como pedido no enunciado.

Assim, para tirar partido dessa informação nova incluída nos modelos 3D, tivemos de ativar as luzes no OpenGL e definir as propriedades de cada modelo no que toca aos seus materiais, passando a usar este novo tipo de coloração.

Por outro lado, numa ótima de otimização, procedemos também à reformulação da criação dos modelos, no gerador, acrescentando os índices dos respetivos vértices (não repetidos) e posteriormente, no *engine*, os modelos passariam a ser desenhados utilizando com o *buffering* dos índices.

Posto isto, estabelecemos um conjunto (simplificado abaixo) de tarefas que incidiram tanto na atualização do **generator** como do **engine** para suportar os requisitos indicados no enunciado e outros definidos por nós:

- Atualizações à aplicação ***generator***:
    1. Alterar a geração de **todos** os modelos introduzindo os índices no ficheiro gerado[2];
    2. Aplicar algoritmos para gerar as coordenadas de textura e as normais dos modelos atualizados;
- Atualizações à aplicação ***engine***:
    1. Alterar o processamento dos vértices até agora utilizado, passando de VBOs (sem índices) para VBOs (com índices);
    2. Atualizar o *parser* de modo a dar suporte aos novos tipos de configuração *xml* e criar estruturas em memória para guardar as propriedades das luzes e materiais,...;
    3. Introduzir uma câmara *free*, assim como outras *features* para facilitar o *debugging*/visualização das cenas;

[1] O Sistema Solar trata-se apenas de uma configuração possível, isto é, a *engine* tem o potencial de reproduzir muitas outras *scenes* a gosto.

[2] Esta atualização não só melhorará a performance da aplicação como também simplificará a geração dos próprios modelos como iremos ver de seguida.

Capítulo 2

Atualizações ao gerador

2.1 Algoritmos de geração dos modelos e introdução de índices

Com a introdução desta fase, percebemos que com a criação das coordenadas de textura e as normais para cada vértice, a memória e o posterior carregamento dos dados no *engine* utilizando a estratégia até agora adotada, ou seja, gerar apenas os vértices para os modelos, não seria uma ideia muito boa em termos de custo computacional para a nossa aplicação, pelo que introduzimos uma otimização na criação dos modelos sem repetição de vértices.

Percebemos também que esta otimização poderia ter sido feita na fase anterior, o que nos permitiria gerir melhor o tempo para este projeto.

2.2 Atualização dos modelos

Tendo isto posto, segue-se uma breve descrição, ao longo das próximas secções, do que foi alterado tanto a nível dos algoritmos dos modelos como a geração das normais, coordenadas de textura, entre outros.

2.2.1 Plano

O plano foi a primitiva mais fácil de alterar, passamos a gerar os 4 vértices que o compõem e indicamos, nos índices a ordem pela qual esses vértices devem ser utilizados, no OpenGL:

```
algoritmo "gerar_plano_indexado"
Inicio

    escrever VERTICES, INDICES, NORMAIS, COORD_TEXTURA \n

    //escrever os 4 vertices (da mesma forma que na fase 1)
    (...)
    //A B C         //B A D
    escrever 0 1 2; escrever 1 0 3

    //escrever coordenadas de textura
    escrever 0 0 //ponto A
    escrever 1 1 //ponto B
    escrever 1 0 //ponto C
    escrever 0 1 //ponto D

    //escrever as 4 normais (iguais)
    para <i = 0> até 4 { escrever 0 1 0 } fim para

Fim
```

Aproveitando esta secção é importante dizer que, ao contrário das outras fases, no *header* do ficheiro gerado temos agora 4 campos separados por vírgulas, que indicam ao *parser* do *engine* o número de linhas/elementos associados:

```
Ficheiro: <nome-modelo>.3d.indexed
Linha 0: NR_VERTICES, NR_INDICES, NR_NORMAIS, NR_COORDTEXTURA
```

No plano, a escrita de vértices segue a mesma forma que a explicada na fase 1, no entanto, agora geramos os vértices A B C e D por esta ordem, ou seja, o índice do A será 0, do B será 1, etc..., daí justifica-se a instrução, por exemplo, **escrever 0 1 2**.

De seguida aplicámos as coordenadas de textura que são muito simples para este caso, mapeando cada um dos vértices para um dos extremos da textura, sempre entre 0 e 1[1], o que irá utilizar a textura toda.

2.2.2 Caixa

O algoritmo de geração da caixa foi em parte alterado, mantendo ciente a lógica adotada até ao momento:

```
algoritmo "gerar_caixa_indexada"
Inicio

    //criar vetores com os vértices de cada plano
    PlanoX0 = [], PlanoXmax = [], ..., Plano Zmax = []

    //vetor para as texturas (igual para todas as faces)
    texturas = []

    para <go_up = 0> até (divisoes+1)
        para <go_right = 0> até (divisoes+1)

            //Criar os vértices (mesmo que a primeira fase), sem repetições
            PlanoX0   -> inserir(novo Ponto(0, (y/(divisoes+1))*go_up), ...)
            PlanoXmax -> inserir(...)
            (...)

            //dividir cada face da caixa mapeando para uma textura entre 0 e 1
            texturas -> inserir(
                go_up / (divisions + 1),
                go_right / (divisions + 1)
            )
        fim para
    fim para
    (...) //explicado a seguir
Fim
```

Desta vez, o algoritmo de geração da caixa começa por criar os vértices para cada face e armazena-os em 6 vetores separados, para que depois, no ficheiro final, se possa escrever os vértices por ordem de cada face, de modo a que seja possível facilitar o algoritmo de criação dos indíces.

As coordenadas de textura também serão iguais para todas as faces, o que nos permitirá mapear uma textura, repetindo-a pelas 6 faces da caixa.

[1]Num próximo capítulo, falaremos sobre esta limitação entre 0 e 1.

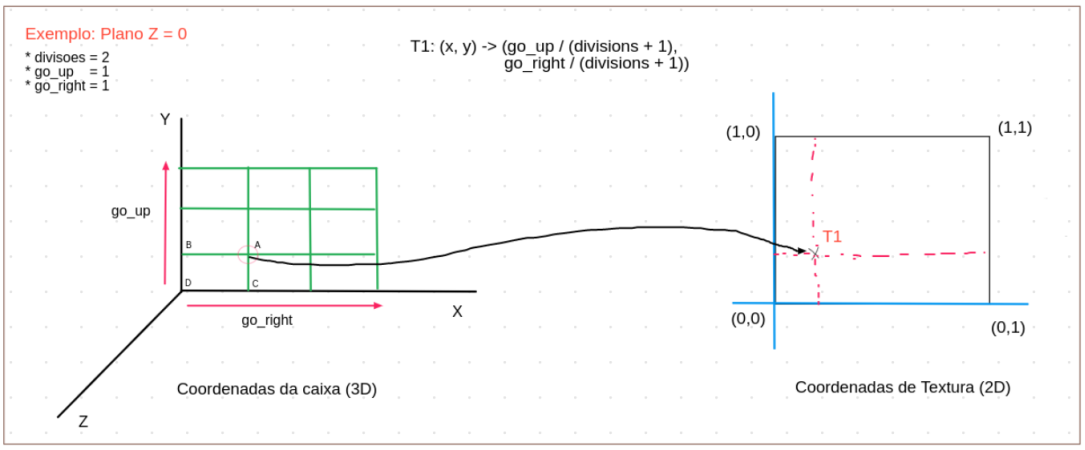

Figura 2.1: Mapeamento de coordenadas para a caixa.

A geração propriamente dita é também simples, ou seja, temos de converter uma posição dada pelos iteradores **go_up** e **go_right** numa entre 0 e 1 que utilize a textura toda, como se pode ver na imagem anterior.

Relativamente às normais da caixa temos que, para cada face, a normal será diferente e sempre perpendicular à própria face, pelo que é relativamente simples estabelecer as normais para os vértices da caixa:

```
algoritmo "gerar_caixa_indexada"
Inicio

    //explicado anteriormente
    (...)

    //normais para face onde X = 0
    para <i=0> até (TOTAL_VERTICES_CAIXA/6) { escrever -1 0 0 } fim para
    //normais para face onde X = Máximo
    para <i=0> até (TOTAL_VERTICES_CAIXA/6) { escrever +1 0 0 } fim para
    (...)

Fim
```

Por fim, falta-nos descrever, de forma breve, a geração dos índicies para este modelo, que tira partido da forma como os vértices são escritos para o ficheiro final. Ora como os vértices são apresentados por face, ou seja, todos da face X = 0, depois todos da face X = máximo e por aí adiante, podemos estabelecer um *offset* que nos ajudará a localizar o índice de um qualquer vértice, sabemos então que:

```
offset = (divisoes + 2) ^2  //número de vértices por cada face

para <go_up = 0> até (divisions + 1)
    para <go_up = 0> até (divisions + 1)
            //Indices: ABCD corresponde ao quadrado da figura 2.1
            iA = (go_up + 1) * (divisions+2) + (go_right + 1)
            iB = iA - 1; iC = ...; iD = ...;
            //Plano X = 0
            escrever iA iB iD; escrever iC iA iD;
            //Plano X = max
            escrever iD+offset iB+offset iA+offset;
            escrever iA+offset iC+offset iD+offset;
            (para os outros planos ...+offset * (2, 3, ...)
```

2.2.3 Esfera

O algoritmo da esfera também foi alterado e, na verdade, muito mais simplificado[2]. Ao invés de utilizar duas variáveis para controlar a criação de 2 semi-esferas aumentando um certo ângulo *beta_top* para a semi-esfera de cima e outro *beta_bottom* (para a inferior), utilizámos na verdade, apenas um ângulo e a geração é contínua desde $-\pi/2$ até $\pi/2$, sendo o deslocamento do único ângulo β entre *stacks* de $\pi/stacks$ incrementado a cada iteração.

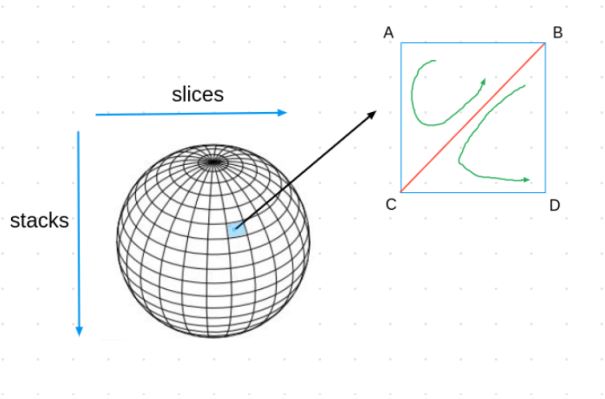

Figura 2.2: Geração de vértices da esfera e correspondência para o rectângulo ABCD adotada.

Os vértices são assim gerados através de coordenadas esféricas convertidas para cartesianas com a conversão habitual já implementada várias vezes em outros contextos.

O algoritmo para a geração dos índices passa a ser tão simples como:

```
para <st = 0> até stakcs
    para <sl = 0> até slices {
            //Considerando o rectângulo da figura 2.2
            A = st * (slices + 1) + sl;       B = A + 1;
            C = (st + 1) * (slices + 1) + sl; D = C + 1;

            escrever A C B; escrever B C D;
    }
```

No caso das coordenadas de textura e normais a estória já é um pouco diferente. Será mais fácil imaginar a textura em si se mapearmos uma esfera para um rectângulo como se pode ver na figura seguinte[3] (com as divisões em *stacks, slices*):

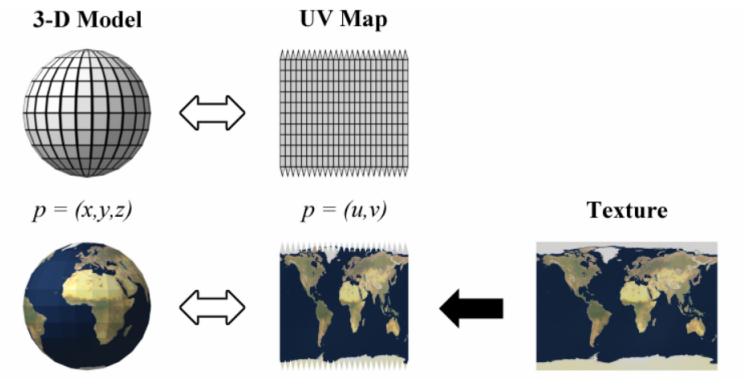

Figura 2.3: Mapemanto UV para a textura de uma esfera.

[2]Uma das grandes alterações incidiu na remoção do desenho das stacks como círculos, ficando apenas os vértices da superfície.

[3]Retirado de `https://en.wikipedia.org/wiki/UV_mapping`

Sendo assim, partimos do pressuposto que uma qualquer posição no iterador das *stacks* e *slices*, na formação da esfera, nos dará a seguinte posição na textura (entre 0 e 1):

```
texturas = []
(...)
para <st = 0> até stacks
    beta = M_PI / 2 - st * beta_offset;
    para <sl = 0> até slices {
        alpha = sl * alpha_offset;
        vertice = (..., ..., ...) //já explicado
        escrever vertice

        texturas -> inserir(
            (sl / slices)
            (1.0f - st / stacks) //inverter a ordem do Y da textura
        )
    }
```

Conseguimos perceber que o raciocínio é parecido com, por exemplo, a geração da caixa, no entanto, em comentário foi referido que se teve que inverter a ordem da ordenada da textura, pois como começamos com $\beta = -\pi/2$ a textura ficaria invertida e resolve-se esse problema subtraindo 1 com a posição do y.

Por fim, as normais são obtidas através da criação de um vetor entre a posição (x, y, z) já calculada e a posição de origem da esfera, neste caso (0,0,0) e, posteriormente, proceder à normalização desse vetor, isto é:

```
normal = Normalizar(Posição_atual - Posição_Origem)
       = Normalizar(Posição_Atual)
       = (x/raio, y/raio, z/raio)
```

Então, podemos inserir esta fórmula no algoritmo até agora desenvolvido:

```
normais = []
(...)
para <st = 0> até stacks
    beta = ...;
    para <sl = 0> até slices {
        alpha = ...;
        vertice = (x , y, z) //já explicado

        normais -> inserir(x/raio, y/raio, z/raio)
        (...)
    }
```

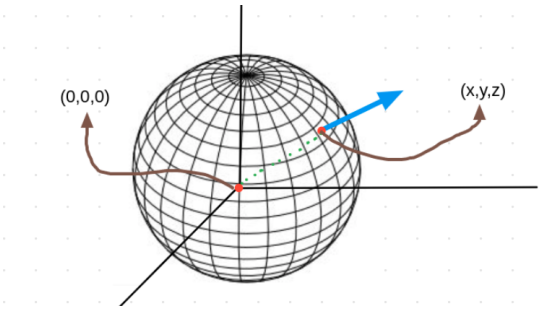

Figura 2.4: Normal da esfera.

Resta-nos apenas inserir, como de costume, os vértices, índices, normais e coordenadas de textura, por esta ordem, no ficheiro final.

2.2.4 Cone

O cone, da mesma forma que os outros modelos, foi gerado desta vez com os índices e apenas será descrito, de forma breve, a geração do mesmo, visto que acaba por ser mais ou menos um processo parecido com os anteriores.

Assim, o cone é gerado em 2 fases, uma para gerar a base do mesmo, que envolve criar um círculo com o raio da base do cone para a *stack = 0* e outra fase para gerar os vértices das *stacks* superiores até atingir a altura desejada.

Para obter os índices para a primeira fase, ou seja, para a base, é mais simples porque são os iniciais do ficheiro e representam **2 * (*slices*+1)** vértices, i. e., para cada *slice*, desenhamos o vértice (0,0,0) e o vértice da circunferência[4].

Na segunda fase, os índices são obtidos da mesma forma que a esfera, somando-lhe um deslocamento que corresponde ao número de vértices referidos anteriormente.

Para as coordenadas de textura também temos um processo de 2 fases, na primeira, ou seja, para a base, temos de extrair da textura final rectangular, um círculo para a *stack 0* e no caso da fase 2 temos um mapeamento parecido com o que tem vindo a ser feito até agora com a caixa e a esfera. A imagem seguinte ajudará na compreensão desta ideia:

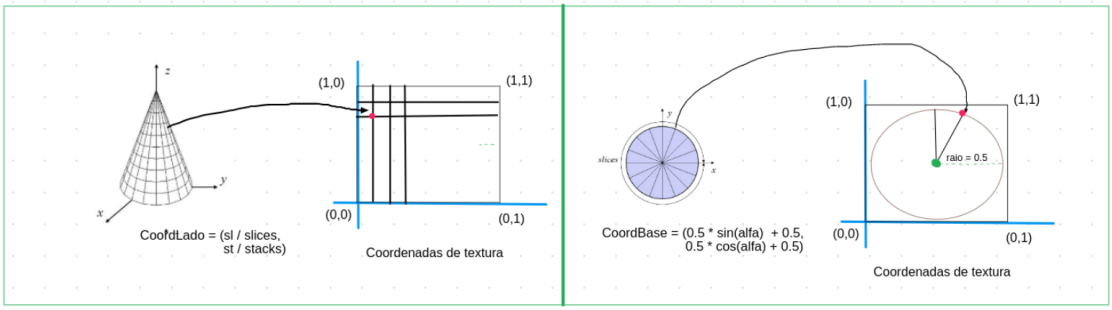

Figura 2.5: Mapeamento de texturas do cone.

Dar-se-á destaque agora ao processo mais complicado de criar as normais, que revelou ser para esta primitiva do cone. O processo poderia ser feito de várias formas, mas a adotada foi a seguinte:

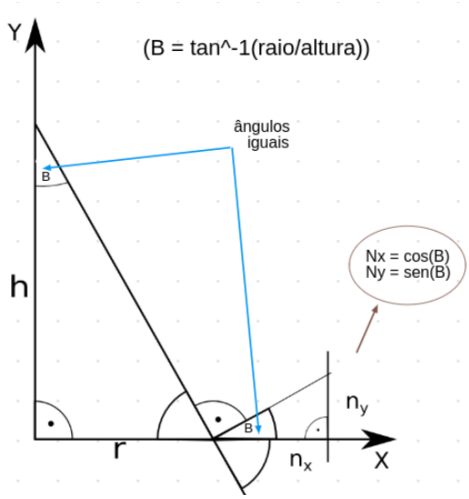

Figura 2.6: Mapeamento de texturas do cone.

Com a imagem anterior podemos perceber que o ângulo **B**, pelas regras de igualdade de ângulos, corresponde igualmente nos dois casos indicados pelas setas azuis e portanto conseguimos definir as coordenadas **x** e **y** da normal (sem considerar a coordenada **z**). Ora passando

[4]De notar que a repetição do vértice (0,0,0) será importante na criação das texturas.

isto para coordenadas a 3 dimensões, e ao longo dos *slices* do cone temos um outro ângulo que influencia o cálculo do vetor normal, o α que está presente no nosso algoritmo.

Segue-se então um excerto do cálculo das normais, considerando esse ângulo:

```
normais = []
(...)
para <st = 0> até stacks
    (...)
    para <sl = 0> até slices
    {
        (...)
        normais -> inserir( (cos(beta)) * sin(alpha),
                            (sin(beta)),
                            (cos(beta)) * cos(alpha))
    }
```

Onde o cálculo do **dx** é o mesmo para o **dy**, sendo apenas necessário multiplicar esses valores por **sin/cos** do ângulo que define os *slices*, o α.

2.2.5 Bezier patch

Os *patchs* de Bezier já tinham sido gerados com índices na fase anterior pelo que foi necessário apenas gerar as normais e as coordenadas de textura.

No caso das texturas, utilizou-se a seguinte estratégia:

```
texturas = []
(...)
para <indice = 0> até nr_total_indices
    para <v = 0> até nivel_tesselacao
        para <u = 0> até nivel_tesselacao
            (...)
            t_u = u / nivel_tesselacao
            t_v = v / nivel_tesselacao
            texturas -> inserir(t_v, t_u)
            (...)
```

Assim, a textura irá preencher um *patch* completo o que fará que a mesma se repetirá ao longo de todos os *patchs* do modelo, que geralmente resultam bem para texturas com padrões. Este método divide a textura em nivel_tesselacao vezes na horizontal e na vertical, mapeando o **u** e **v** numa coordenada de textura entre 0 e 1.

As normais são fáceis de calcular pois temos apenas que seguir a fórmula do cálculo das tangentes num ponto de um *patch* de bezier e proceder ao produto entre as tangentes para obter a normal. Pretende-se então reproduzir este processo:

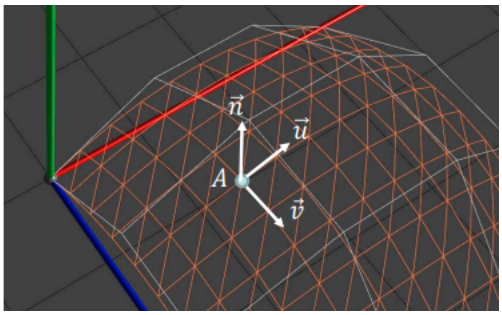

Figura 2.7: Normal de uma superfície de Bezier.

Assim, utilizando o formulário fornecido na secção das aulas teóricas, temos que, os vetores **u** e **v** que queremos calcular podem ser obtidos a partir das fórmulas seguintes:

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

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

Para isso, desenvolvemos uma nova função **getBezierNormal** que procede a esses cálculos, no qual iremos mostrar os mesmos para o vetor **v**, por exemplo:

```
algoritmo "calcular_normal_bezier"
Inicio
    (...)
    v_vector[4] = { (v * v * v), (v * v), v, 1 };
    v_vector_deriv[4] = { 3 * (v * v), 2 * (v), 1, 0 };

    // M_v = M * v
    M_v_deriv = multMatrixVector(*M_BEZIER, v_vector_deriv);

    // Pxyz_M_v_deriv = P * M_v_deriv
    Pxyz_M_v_deriv[0] = multMatrixVector(PijX, M_v_deriv);
    Pxyz_M_v_deriv[1] = multMatrixVector(PijY, M_v_deriv);
    Pxyz_M_v_deriv[2] = multMatrixVector(PijZ, M_v_deriv);

    // M_Pxyz_M_v_deriv = M * Pxyz_M_v_deriv
    (...)

    dV = [0, 0, 0];
    para <i=0> até 4 {
        (...)
        dV[0] += u_vector[i] * M_Pxyz_M_v_deriv[0][i];
        dV[1] += u_vector[i] * M_Pxyz_M_v_deriv[1][i];
        dV[2] += u_vector[i] * M_Pxyz_M_v_deriv[2][i];
    }

    normalizar(dU) //omitidos os calculos
    normalizar(dV)

    normal = produto_vetores(dV, dU)
Fim
```

Tão importante como seguir a fórmula à risca foi fazer também a normalização final dos vetores para que se possa aplicar o *cross product* entre eles, sendo depois retornada a normal que será normalizada novamente na função principal. Outro caso curioso que nos aconteceu foi que reparamos na criação de alguns **nan** no ficheiro gerado, visto que os números provavelmente seriam muito pequenos e impossíveis de representar em computador tendo "solucionado" esse problema substituindo a normal pelo vetor (0,0,0).

Capítulo 3

Atualizações ao Engine

3.1 Novas formas de configuração do xml

3.1.1 Luzes

Configuração das luzes

O ficheiro de configuração em **xml** que temos vindo a utilizar dará suporte, nesta fase, ao *setup* das luzes que definirão a nossa *scene* podendo ser definidas várias luzes de 3 diferentes tipos: **point**, **directional** e **spot**.

A definição de luzes deve aparecer, deste modo, entre as *tags*:

```
<lights>
    <light ... />
</lights>
```

Segue-se então a definição desejada para cada tipo de luz:

- Luzes definidas num ponto (**point**):

  ```
  <light type="POINT" posX="..." posY="..." posZ="..."  />
  ```

  Este tipo de luzes são definidas num ponto e emitem em todas as direções.

- Luzes com uma direção (**directional**):

  ```
  <light type="DIRECTIONAL" posX="..." posY="..." posZ="..."  />
  ```

  Este tipo de luzes não está definida num ponto mas segue uma dada direção. Neste caso, o posX/Y/Z especificado refere-se a um vetor, pelo que, o vetor fornecido ao OpenGL em a 4 coordenada **w** a 0.

- Luzes *spotlight* (**spot**):

  ```
  <light type="SPOT" posX="..." posY="..." posZ="..."
                     dirX="..." dirY="..." dirZ="..."
                     cutoff="4" exponent="100" />
  ```

  Estas luzes *spotlight* simulam um luz parecida a uma vinda de uma lanterna, onde especificámos um ponto, uma direção, a abertura da luz (*cutoff*), ou melhor, o seu *spread* e um expoente que define a intensidade de distribuição da luz.

Para todas as luzes podemos também especificar a intensidade RGBA das luzes no que toca à componente difusa, ambiente e especular, da seguinte forma:

```
<light ... ambiR="..." ambiG="..." ambiB="..."
           specR="..." specG="..." specB="..."
           diffR="..." diffG="..." diffB="..." />
```

Parser e estruturas de dados

O parser segue um algoritmo genério que temos vindo a adotar, fruto também da utilização da API do *tinyxml2*. A rotina que trata do *parsing* das luzes é a que contém o seguinte protótipo:

```
static vector<LightSource>* processLightsTag(XMLElement *lights_ptr)
```

A leitura de cada *tag* **light** em **lights** produz como resultado uma nova estrutura de dados que designamos por **LightSource** e encontra-se definida em *lights.h* nos *headers* do *engine*.

```
class LightSource {

    public:
        string lightType; //spot, dir, point
        int lightEnumNumber; //light offset from GL_LIGHT0

        float point[4];

        //light intensity RGBA colour
        float diffuseComponent[4];
        float specularComponent[4];
        float ambientComponent[4];

        float SpotDirection[3];
        float SpotExponent;
        float SpotCutoff;
}
```

Uma luz será apenas criada se, no mínimo, o seu *type* for especificado, visto que, inicialmente, valores *default* são atribuídos ao construtor da própria classe e esses valores podem ser consultados na documentação do próprio OpenGL.

Deste modo, a função com o protótipo especificada acima é chamada no *main.cpp* à parte da leitura dos grupos da nossa *scene*, visto que são *scopes* diferentes, e produz como resultado um vetor de luzes que será fornecido ao *engine*.

Inicialização e desenho das luzes

Assim que o parser inicialize as estruturas de dados principais, fornece-as ao *engine* que fará a inicialização e desenho das luzes no OpenGL. A primeira parte está em verificar se existem luzes ou não, e caso existam ativá-las:

```
  if (!scene.second->empty()) {
        glEnable(GL_LIGHTING); //turn on lighting
        for (auto it = scene.second->begin(); it < scene.second->end(); it++) {
            glEnable(GL_LIGHT0 + it->lightEnumNumber); //turn on the light source
        }
    }
```

De seguida, no **renderScene** procedemos à atribuição dos parâmetros às respetivas luzes à nossa *scene*:

```
//posicionar a luz
glLightfv(lightIndex, GL_POSITION, it -> point);

//aplicar intensidades
glLightfv(lightIndex, GL_SPECULAR, it -> specularComponent);
glLightfv(lightIndex, GL_AMBIENT, it -> ambientComponent);
glLightfv(lightIndex, GL_DIFFUSE, it -> diffuseComponent);

if (it -> lightType == "SPOT") {
   glLightfv(lightIndex, GL_SPOT_DIRECTION, it -> SpotDirection);
   glLightf(lightIndex, GL_SPOT_EXPONENT, it -> SpotExponent);
   glLightf(lightIndex, GL_SPOT_CUTOFF, it -> SpotCutoff);
}
```

3.1.2 Materiais e texturas dos modelos

Materiais

Os modelos agora possuem um tipo diferente de coloração, sendo-lhes atribuído materias que os caracterizam. Os materias em questão especificam os parâmetros de cada modelo (conjunto de vértices) para este sistema com luzes indicando a refletância das diferentes superfícies no que toca à cor RGBA reproduzida, e também a emissividade, que se tem por exemplo, no sol do Sistema Solar.

No que toca à configuração do *xml*, a configuração dos materiais pode ser feita acrescentando alguns atributos à *tag* do modelo, da seguinte forma:

```
<model file="..." ambiR="..." ambiG="..." ambiB="..."
                  specR="..." specG="..." specB="..."
                  diffR="..." diffG="..." diffB="..."
                  emisR="..." emisG="..." emisB="..."/>
```

Aplicação dos Materiais

Desta vez, antes de desenhar o modelo em questão, utilizamos uma função **setMaterials(model)** que faz a seguinte sequência de instruções:

```
glMaterialfv(GL_FRONT, GL_DIFFUSE, model -> diffuseComponent);
glMaterialfv(GL_FRONT, GL_SPECULAR, model -> specularComponent);
glMaterialfv(GL_FRONT, GL_EMISSION, model -> emissiveComponent);
glMaterialfv(GL_FRONT, GL_AMBIENT, model -> ambientComponent);
```

Especificação das Texturas

As texturas podem agora ser especificadas no próprio *xml* acrescentando um último atributo à *tag* do **model**:

```
<model file="..." texture="..." />
```

No momento de inicializar as texturas a partir do DevIL o *engine* assume que estas se encontram armazenadas em **examples/Textures** a partir da diretoria *root* do desta fase.

O ficheiro da textura passa a ser então um novo elemento da classe MODEL_INFO e, caso não seja possível carregar uma dada textura, o modelo fica inicializado com uma cor *default* branca (RGBA = 1).

3.2 Parsing do novo tipo de ficheiro e atualização do modelo

Como já foi dito no capitulo do *generator*, o ficheiro gerado para o modelo tem um novo formato:

```
0: NR_VERTICES, NR_INDICES, NR_NORMAIS, NR_COORDTEXT
```

A linha especificada, linha 0, corresponde ao *header* que nos permite estabelecer várias fases de *parsing*, lendo primeiro NR_VERTICES linhas, depois NR_INDICES linhas, etc..., principalmente para facilitar a leitura e inicialização dos modelos.

Estes últimos, nesta fase, tiveram uma série de variáveis importantes que foram adicionadas maioritariamente devido ao desenho e inicialização de VBOs:

```
class MODEL_INFO {

    vector<float>* vertices; //já tinha
    vector<GLuint>* indexes;
    vector<float>* texturesCoord;
    vector<float>* vertexNormals;

    GLuint verticesBuffer[1]; //já tinha
    GLuint indexesBuffer[1];
    GLuint textureBuffer[1];
    GLuint normalsBuffer[1];

    string textureFile;
    GLuint glutTextureID;

    float diffuseComponent[4];
    float specularComponent[4];
    float ambientComponent[4];
    float emissiveComponent[4];
}
```

3.3 Inicialização dos *buffers*

3.3.1 VBOs

Na fase anterior apenas foi necessário inicializar os VBOs para armazenamento dos vértices dos modelos, ora como neste fase introduzimos índices, texturas e normais, estes também terão de ser inicializados nesses *buffers*.

Assim, na função de inicialização introduzida na fase anterior, foram adicionadas as seguintes instruções:

```
  //Indexes
  glGenBuffers(1, model->indexesBuffer);
  glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, model -> indexesBuffer[0]);
  glBufferData(GL_ELEMENT_ARRAY_BUFFER, indexArraySize, indices, GL_STATIC_DRAW);

  //Normals
  glGenBuffers(1, model->normalsBuffer);
  glBindBuffer(GL_ARRAY_BUFFER, model -> normalsBuffer[0]);
  glBufferData(GL_ARRAY_BUFFER, vertexNormalArraySize, normais, GL_STATIC_DRAW);
```

```
if (model->settings[1]) { //has textures

  //Texture Coordinates
  glGenBuffers(1, model -> textureBuffer);
  glBindBuffer(GL_ARRAY_BUFFER, model -> textureBuffer[0]);
  glBufferData(GL_ARRAY_BUFFER, textureArraySize, texturaCoord, GL_STATIC_DRAW);
}
```

3.3.2 Carregamento de texturas

Visto que o processo de carregamento de texturas produz um código um pouco diferente do utilizado nos outros módulos da aplicação decidimos separar o carregamento das texturas num ficheiro à parte, sendo ele **textures.cpp/h**.

O carregamento propriamente dito utiliza a biblioteca de funções disponibilizada num dos guiões práticos chamada DevIL que nos fornece rotinas de leitura de *bitmaps*.

O processo completo é idêntico ao realizado nos guiões práticos e, de forma resumida, trata-se do seguinte:

```
//init DevIL
ilInit();

//Set texture origin in lower left side
ilEnable(IL_ORIGIN_SET);
ilOriginFunc(IL_ORIGIN_LOWER_LEFT);

//Generates 1 image name/id
ilGenImages(1, &IMG_ID);
ilBindImage(IMG_ID);

ilLoadImage((ILstring) textureFile);
ilGetInteger(IL_IMAGE_HEIGHT);
ilConvertImage(IL_RGBA, IL_UNSIGNED_BYTE);
DATA = ilGetData();

//Generates 1 texture for the gpu
glGenTextures(1, &model -> glutTextureID);

glBindTexture(GL_TEXTURE_2D, model -> glutTextureID);
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_MIPMAP_LINEAR);

//upload image data
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, IMG_WDT, IMG_HGT, 0,
             GL_RGBA, GL_UNSIGNED_BYTE, DATA);

glGenerateMipmap(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, 0);
```

De forma resumida, estabelece-se a origem da textura para o canto inferior esquerdo da mesma, o que nos permite criar coordenadas de textura entre 0 e 1 excluindo valores negativos,

depois pedimos ao DevIL para gerar um ID para a imagem e carregamos os dados da mesma, i.e., um array de pixéis.

Com isto, passamos ao OpenGL e geramos a textura para o GPU, estabelecendo um conjunto de propriedades para a mesma de modo a que o OpenGL interprete a textura duma determinada forma e até gere os *mipmaps*.

3.4 Desenho de VBOs

O próximo passo será desenhar os nossos modelos com este conjunto de *buffers* que foram criados e atribuídos aos modelos, tendo sido feito na fase anterior apenas o desenho dos vértices em si.

Assim, no módulo **draw-elements.cpp/h** atualizamos a função que desenha os modelos com VBOs, passando a fazer o seguinte:

```
//vertex buffer
glBindBuffer(GL_ARRAY_BUFFER, model.verticesBuffer[0]);
glVertexPointer(3, GL_FLOAT, 0, nullptr);

//normal buffer
glBindBuffer(GL_ARRAY_BUFFER, model.normalsBuffer[0]);
glNormalPointer(GL_FLOAT, 0, 0);

if (isTextured) {

   //texture buffer
   glBindBuffer(GL_ARRAY_BUFFER, model.textureBuffer[0]);
   glTexCoordPointer(2, GL_FLOAT, 0, 0);
   glBindTexture(GL_TEXTURE_2D, model.glutTextureID);
}

//draw buffer
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, model.indexesBuffer[0]);
glDrawElements(GL_TRIANGLES, indexArraySize, GL_UNSIGNED_INT, nullptr);
```

De notar que, na configuração dos nossos modelos, alguns sofreram transformações que se revelaram indesejadas na criação dos VBOs, porque as escalas dos eixos provocaram alterações não só nos vértices como também nas normais do modelo e os resultados obtidos, numa fase inicial, não se mostraram adequados no que toca à iluminação. Descobrimos então que era necessário pedir ao OpenGL que normalizasse esses vetores caso estes tenham sido alterados pelo que tivemos de ativar a seguinte opção:

```
glEnable(GL_RESCALE_NORMAL);
```

Por outro lado, para permitir a utilização de coordenadas de textura e normais com VBOs tivemos de ativar as seguintes opções:

```
glEnableClientState(GL_NORMAL_ARRAY);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
```

Tendo isto posto, ficam assim concluídos os objetivos para esta última fase, pelo que o próximo capítulo trata-se de algumas considerações sobre decisões que foram tomadas ao longo deste projeto.

3.5 Outras implementações

3.5.1 Câmara *free*

Tendo isto sido falado nas tarefas estabelecidas na introdução deste relatório, consideramos importante dedicar uns parágrafos à nossa implementação de uma câmara em modo *free*.

O raciocínio passa por considerar que a câmara possui uma posição P e um ponto para onde está a olhar e com isso, conseguimos criar o vetor da direção do olhar da mesma. Com esse vetor podemos reproduzir o movimento para atrás e para a frente (teclas 'w' e 's') sempre na direção do olhar.

Já o movimento lateral pode ser obtido calculando o vetor resultante do *up* com a direção do olhar multiplicando o resultado por -1 ou 1 caso seja para ir para a esquerda ou direita (teclas 'a' e 'd').

Por fim, a câmara também consegue olhar em volta, tanto para cima/baixo, como rodar para a esquerda/direita utilizando um ângulo alfa e beta para definir a posição do olhar com coordenadas esféricas (teclas 'arr_up', 'arr_down',...), conseguindo também consegue descer e subir de altitude (com o 'spacebar' e a tecla '2').

As variáveis de controlo da câmara bem como a aplicação dos vetores referidos acima encontra-se definida em *load-graphincs.cpp*.

3.5.2 Eixos dos modelos e curvas de *catmull*

Os modelos podem também ser visualizados com ou sem os eixos para facilitar o *debugging*, alternando a tecla 'q'. Por outro lado, achamos por bem fazer o *rendering* das curvas de *catmull* através de pontos para tornar a *engine* visualmente mais interessante.

Capítulo 4

Resultados obtidos

Nesta última fase conseguimos juntar tudo o que teria sido implementado até agora culminando na criação de uma série de demonstrações em *xml* do seu funcionamento, seguem-se algumas delas:

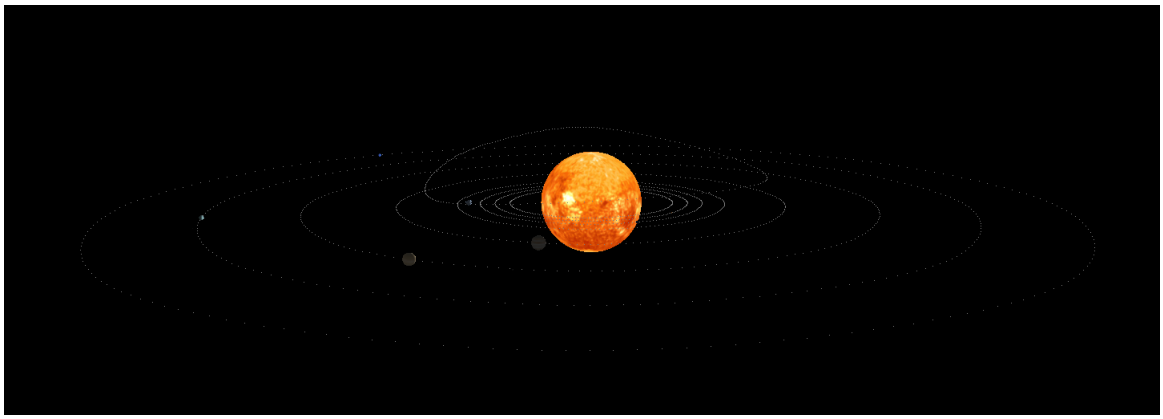

Figura 4.1: Sistema Solar - vista longe.

Figura 4.2: Sistema Solar - vista aproximada

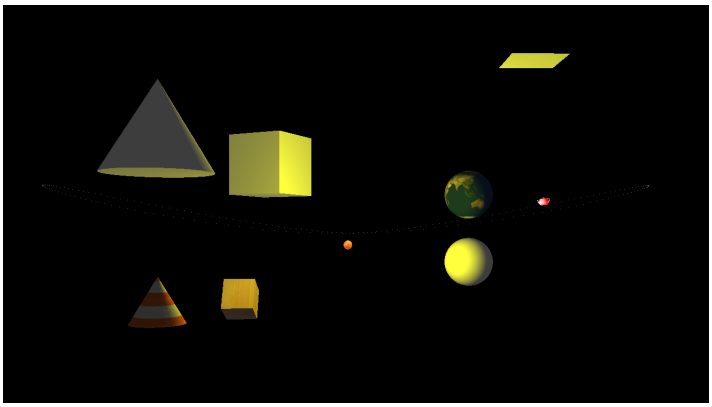

Figura 4.3: Primitivas da fase 1.

Figura 4.4: Exemplos de *spotlights* em esferas.

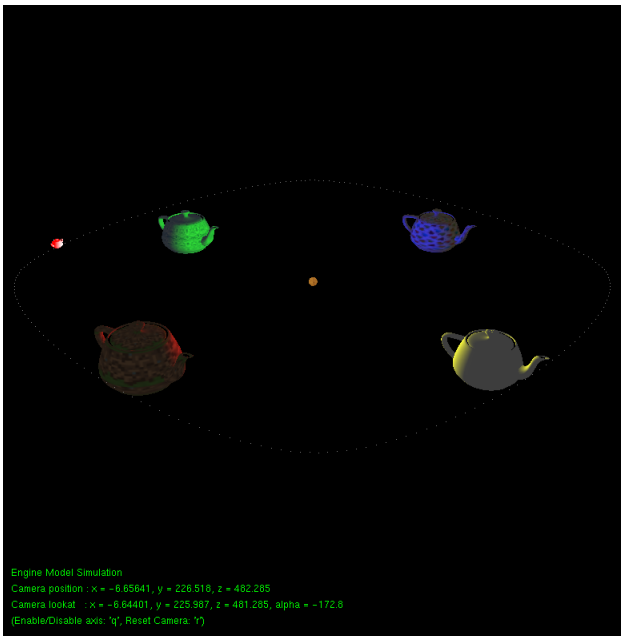

Figura 4.5: Exemplos de luzes com materiais diferentes em *teapots*.

Capítulo 5

Conclusões

A *engine* termina assim, nesta última fase, o desenvolvimento faseado que o nosso grupo tem vindo a estabelecer, resultando num *software* flexível de configuração de cenas em *xml* tirando partido do processamento que a API de Computação Gráfica OpenGL nos proporciona.

Quer isto dizer que, mais importante que obter o resultado final, é conseguir, em cada fase, ter a oportunidade de aplicar os conceitos teóricos e comprovar o seu funcionamento através de código e algoritmos de geração de modelos, movimento dos mesmos, luzes, texturas e utilização de VBOs.

Já nesta fase, como forma de resumo do trabalho desenvolvido, pudemos estabelecer a essência do sistema solar diferenciando os planetas através das suas texturas e incluindo o sol como a única fonte de luz emissiva de todo sistema.

A otimização foi também uma parte fundamental para perceber como os dados eram carregados em GPU e quais as melhores alternativas nas tomadas de decisão acerca deste assunto, avaliando o *trade-off* entre mais memória e/ou carregamentos mais rápidos. Neste tópico, aproveitamos para referir que testes de *profiling* poderiam ser uma boa forma de tirar conclusões acerca deste assunto, pelo que seria algo a implementar num futuro projeto semelhante.

Em suma, esta última fase revelou-se fulcral para o desenvolvimento do projeto no sentido visual do sistema e das próprias capacidades do *engine*, concluindo então que todos os objetivos para este projeto foram cumpridos na íntegra e possibilitando, num futuro próximo, a introdução de novos conceitos/funcionalidades.