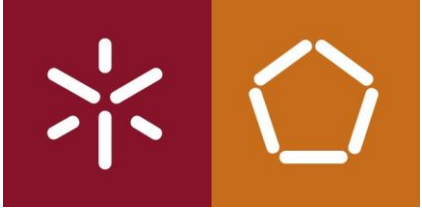


Relatório do Trabalho Prático

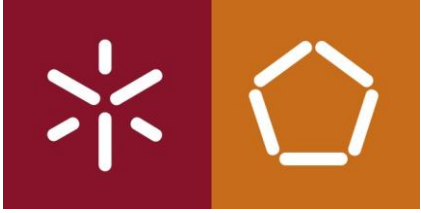
Computação Gráfica

Fase 4 – *Normals and Texture Coordinates*

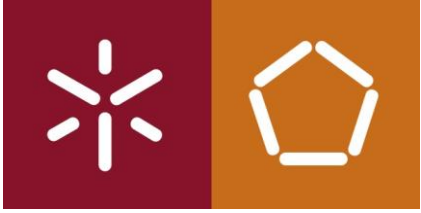
21 maio 2017

Catarina Rocha Cardoso – a75037
Diogo Mendes Gomes – a73825
Francisco Roriz Ferreira Mendes – a75097
Luís Manuel Leite Costa – a74819

Mestrado Integrado em Engenharia Informática | Universidade do Minho



Índice



Introdução

Na última fase, com o fim de aprimorar a representação do sistema solar foi solicitado que se incluísse texturas nas figuras e que se simulasse a emissão de luz do sol. Para tal, foi necessário modificar-se o programa para gerar figuras afim de calcular adicionalmente coordenadas normais e coordenadas de texturas. Foram ainda feitas alterações ao programa que lê os ficheiros XML para dar suporte a estas novas coordenadas.

Decisões

- Câmara

Como para a etapa segunda etapa do projeto já tinha sido implementada e devidamente explicada uma câmara FPS, decidiu-se não efetuar alterações.

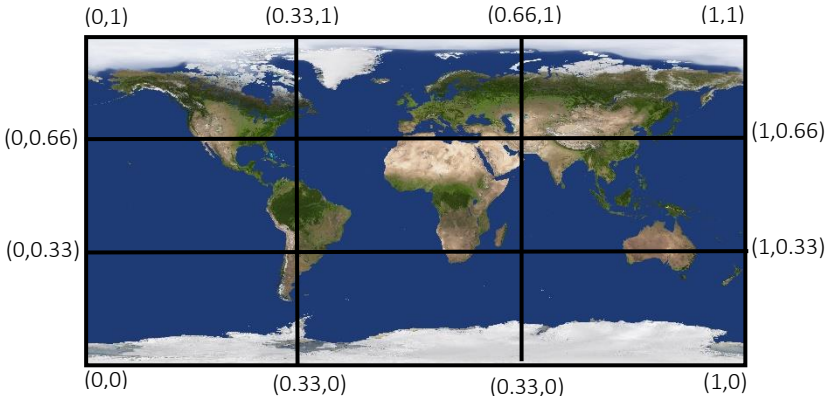
Abordagem

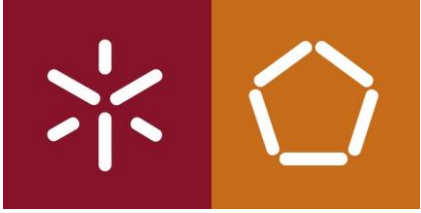
Generator

Nesta última fase foi necessário realizar algumas alterações no *Generator* previamente desenvolvido de modo a que fossem incluídas, para além das coordenadas de posição, as coordenadas dos vetores normais e as coordenadas de textura referentes a cada um dos vértices.

Para realizar as alterações realizadas afim de suportar a implementação de luzes, foi necessário escrever-se no ficheiro gerado novas coordenadas que representam o vetor normal à superfície no vértice em questão. Estas coordenadas tomam valores ente 0 e 1 devido a serem apenas vetores direcionais. O cálculo de vetores normais de figuras "básicas" pode diferenciar-se entre as superfícies planas e curvas. Para as superfícies planas, como numa *box* ou para a base de um *cone* basta escrever um vetor que indique a direção da normal, por exemplo **(0,-1,0)** caso a face esteja orientada para baixo. Em superfícies curvas como uma *sphere* ou um *cylinder*, o vetor normal é determinado pela normalização da coordenada do vértice que se está a desenhar.

Com o intuito de preencher as figuras com texturas, foi necessário completar-se os ficheiros .3d com as coordenadas das texturas, sendo que estas variam entre 0 e 1. Usando novamente o caso da esfera para servir de exemplo de explicação, as coordenadas são determinadas pela divisão entre a *stack* ou *slice* que se está a desenhar e o número total de *stacks* e *slices*. Segue-se uma ilustração das coordenadas da textura a aplicar-se a uma esfera com 4 *stacks* e *slices*.





Bezier Patches

Para calcular os vetores normais aos vértices de uma superfície de *Bezier* foi adicionado o método `bezier_normal` em que são feitas as seguintes operações:

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

$$M = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

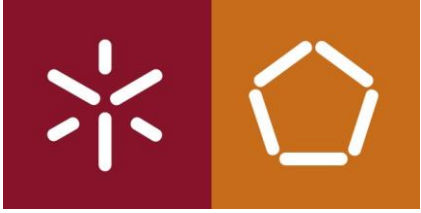
$$\frac{\partial B(u,v)}{\partial v} = UM \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} M^T \begin{bmatrix} 3v^2 \\ 2v \\ 1 \\ 0 \end{bmatrix}$$

1. Após estes cálculos, os vetores obtidos através destas derivações são normalizados, calcula-se produto vetorial entre estes, sendo o resultado novamente normalizado. Obtém-se então o vetor normal de um dado vértice calculado pelo método `bezier_point`.

```
    ...
    //calculo da normal
    normalize(pU);
    normalize(pV);

    float *normal = new float[3];
    cross(pV, pU, normal);
    normalize(normal);

    return normal;
}
```

Ao código previamente implementado, foram apenas adicionadas as linhas indicadas.

```
for (i = 0; i < nr_patches; i++) {
   int patch_size = patches[i].size();

   for (j = 0; j < patch_size; j++) {
      k = patches[i][j];
      control_points[j][0] = cpoints[k][0];
      control_points[j][1] = cpoints[k][1];
      control_points[j][2] = cpoints[k][2];
   }

   for (j = 0; j <= tess; j++) {
      for (k = 0; k <= tess; k++) {
          float* p = bezier_point(control_points, patch_size, k / tess, j / tess);
          f[j][k][0] = p[0]; f[j][k][1] = p[1]; f[j][k][2] = p[2];

          float* t = bezier_normal(control_points, patch_size, k / tess, j / tess);
          r[j][k][0] = t[0]; r[j][k][1] = t[1]; r[j][k][2] = t[2];
      }
   }
   ...}
```



Processo de Leitura

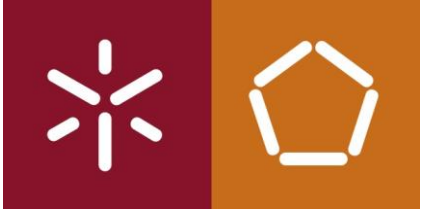
Os diferentes tipos de coordenadas (posição, normal, textura) são imprimidos nos ficheiros .3d sem qualquer separação, isto é, três linhas consecutivas representam, para um dado vértice, as coordenadas de posição, coordenadas do vetor normal e coordenadas de textura, respetivamente. Devido a esta alteração, a leitura dos ficheiros .3d teve de ser modificada.

São criados três vetores (`position`, `normal` e `texCoord`), onde são introduzidas todas as coordenadas lidas do ficheiro pela ordem que são apresentadas. Como foi mantida a implementação com VBOs (Vertex Buffer Object), foi adicionado à classe `File` um *array* de *buffers*, que substituí o único *buffer* previamente utilizado. São, mais uma vez, invocados os métodos que vinculam os dados dos vetores `position`, `normal` e `texCoord` à memória da placa gráfica.

```
glGenBuffers(3, fich.buffers);

glBindBuffer(GL_ARRAY_BUFFER, fich.buffers[0]);
glBufferData(GL_ARRAY_BUFFER,position.size()*sizeof(float),&(position[0]),GL_STATIC_DRAW);

glBindBuffer(GL_ARRAY_BUFFER, fich.buffers[1]);
glBufferData(GL_ARRAY_BUFFER,normal.size()*sizeof(float),&(normal[0]),GL_STATIC_DRAW);

glBindBuffer(GL_ARRAY_BUFFER, fich.buffers[2]);
glBufferData(GL_ARRAY_BUFFER,texCoord.size()*sizeof(float),&(texCoord[0]),GL_STATIC_DRAW);
```

Para que pudesse aplicar uma textura a um modelo, para além das coordenadas de textura do modelo foi necessário carregar a imagem e guardar o seu identificador na classe `File` (`unsigned int texID`). Quando, numa etiqueta do tipo `model` é encontrado o atributo `texture`, é carregado o ficheiro indicado.

As novas etiquetas do tipo `lights` e `light` também foram incluídas para que se pudesse definir no ficheiro XML as luzes a introduzir. Para isso, poderá ser recebido como atributo `type` que deverá ser “point”, “directional” ou “spotlight”, a posição da luz (`posX`, `posY`, `posZ`), a direção da luz (`dirX`, `dirY`, `dirZ`) e o *spread angle* (`angle`), se aplicável.



Estruturas de dados

Para se abranger os dados recolhidos na leitura do ficheiro XML sobre as luzes a incluir, foi adicionada uma nova classe à coleção já existente. Para se conservar todos os tipos de dados, manteve-se o vetor global que contém objetos da classe `Tag*`: `std::vector<Tag*> v;`

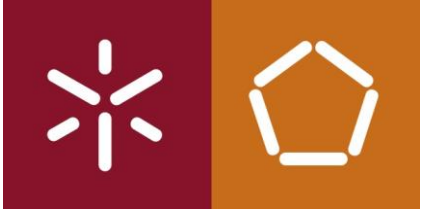
- **Classes**

1. **Tag:** Classe abstrata que possui o método virtual `draw` que será utilizado por cada uma das suas subclasses (`File`, `Translation`, `Rotation`, `Scale`, `Matrix`, `CatmullRom`, `Spin`, `Light`).
2. **File:** Classe representativa de um ficheiro. Contém um `int` com o número de coordenadas dos vértices lidos do ficheiro .3d, um `int` com a identificação da textura a aplicar e um *array* de *buffers* necessário aquando a invocação do método `glDrawArrays`. Os valores dos *arrays* `diffuse`, `ambient`, `specular`, `emission` e `shininess` são os *standard* e podem ser alterados no ficheiro XML.

```
class File : public Tag {
	public:
		int size;
		GLuint buffers[3];
		unsigned int texID = 0;

		float diffuse[4] = { 0.8f, 0.8f, 0.8f, 1.0f },
			ambient[4] = { 0.2f, 0.2f, 0.2f, 1.0f },
			specular[4] = { 0.0f, 0.0f, 0.0f, 1.0f },
			emission[4] = { 0.0f, 0.0f, 0.0f, 1.0f },
			shininess[1] = { 128 };

		void draw(Tag* file);
};
```

3. **Translation**
4. **Rotation**
5. **Scale**
6. **Matrix**
7. **CatmullRom**
8. **Spin**
9. **Light:** Classe representativa de uma luz. Armazena os dados necessários à representação sua representação: número da luz, posição, direção e posição (se aplicável). Também incluí uma *flag* que indica se a luz é, ou não, do tipo "spotlight".

```
class Light : public Tag {
	public:
		GLenum number;
		float position[4];
		float direction[3];
		int angle;
		bool flag = false;

		void draw(Tag* light);
};
```



Ciclo de Rendering e Recolha de Dados

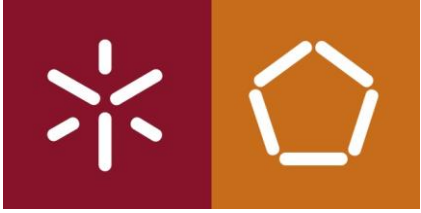
Neste ciclo é percorrido o vetor previamente carregado aquando o processo de leitura de dados do ficheiro XML. Cada iteração do ciclo que se segue irá invocar o método `draw` e, dependendo da classe na posição `i`, são executadas instruções diferentes.

```
std::vector<Tag*>::iterator it;
for (it = v.begin(), i = 0; it < v.end(); it++, i++)
     v[i]->draw(v[i]);
```

- Modelos

1. File: Os ficheiros previamente criados no *generator* irão ser usados através da sua invocação no ficheiro XML. Para os ficheiros presentes nas tags `<model file="..." />`, são guardados os seus dados (tamanho e buffer) numa classe do tipo `File`. Aquando a iteração do vetor `v`, caso seja encontrada uma classe deste tipo, irá se representada a figura através das funções `glVertexPointer` e `glDrawArrays`. As coordenadas dos vetores normais e de textura são aplicadas através dos métodos `glNormalPointer` e `glTexCoordPointer`, respetivamente. Os materiais são ajustados, consoante os valores lidos anteriormente através de invocação da função `glMaterialfv` e as texturas são aplicadas (e repostas no final) através do método `glBindTexture`.

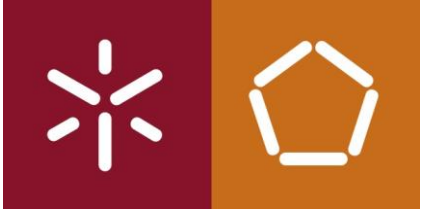
```
     void draw(Tag* file) {
          File& f = dynamic_cast<File&>(*file);

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

          glBindTexture(GL_TEXTURE_2D, f.texID);

          glBindBuffer(GL_ARRAY_BUFFER, f.buffers[0]);
          glVertexPointer(3, GL_FLOAT, 0, 0);

          glBindBuffer(GL_ARRAY_BUFFER, f.buffers[1]);
          glNormalPointer(GL_FLOAT, 0, 0);

          glBindBuffer(GL_ARRAY_BUFFER, f.buffers[2]);
          glTexCoordPointer(2, GL_FLOAT, 0, 0);

          glDrawArrays(GL_TRIANGLES, 0, f.size - 1);

          glBindTexture(GL_TEXTURE_2D, 0);
     }
```



- Luzes

2. **Light**: Quando se pretende seja produzido uma luz cria-se uma classe do tipo `Light` que irá conter nas suas variáveis os atributos presentes na tag `<light type=.../>`. Quando é encontrado um elemento da classe `Light` no vetor é executado o método `glLightfv(l.number, GL_POSITION, ...)`, caso o atributo que especifica o tipo de luz seja "point" ou "directional". Caso seja "spotlight", para além do método anterior, são invocados os seguintes: `glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, ...)` e `glLightf(l.number, GL_SPOT_CUTOFF, ...)`, que especificam a direção e o *spread angle* da luz a representar.

```
void draw(Tag* light) {
	Light& l = dynamic_cast<Light&>(*light);
	glLightfv(l.number, GL_POSITION, l.position);

	if (flag) {
		glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, l.direction);
		glLightf(l.number, GL_SPOT_CUTOFF, l.angle);
	}
}
```



Conclusão

Esta última etapa de desenvolvimento foi abordada primeiramente pela inclusão das texturas, o que implicou que fossem feitas alterações no G*enerator*, a nível do ficheiro XML e da sua leitura, de forma a carregar as texturas nas respetivas figuras. Seguiu-se, então, a implementação das luzes que começou novamente pela modificação no *Generator* para gerar as coordenadas dos vetores normais.

Quanto ao *Engine,* as alterações de maior relevo foram as modificações a nível dos *buffers* (para que comportasse as coordenadas normais e de textura) e a introdução dos elementos de iluminação.