

Computação Gráfica

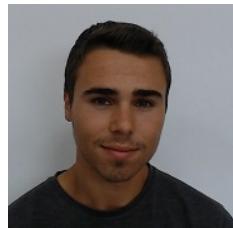
Trabalho Prático (Parte 4)

Luís Miguel Ramos (A83930)
Válter Carvalho (A84464)

27 de Maio de 2020



Escola de Engenharia



Grupo nº 26
Mestrado Integrado em Engenharia Informática
Universidade do Minho

Conteúdo

1	Introdução	3
2	Gerador	4
2.1	Plane	4
2.2	Box	5
2.3	Sphere	6
2.4	Cylinder	7
2.5	Ring	8
2.6	Cone	9
2.7	Vase	9
2.8	Teapot	10
3	Motor	11
3.1	Luzes	11
3.2	Texturas, Componente Material e Iluminação nos Objetos . .	12
3.3	Ficheiro de XML e Interpretador	16
3.3.1	Luzes	16
3.3.2	Câmera	17
3.3.3	Componente Material e Texturas	18
3.4	Alterações Adicionais	18
4	Resultado Final	20
4.1	Plane	20
4.2	Box	20
4.3	Sphere	21
4.4	Cylinder	21
4.5	Ring	22
4.6	Cone	22
4.7	Vase	23
4.8	Teapot	24
4.9	Sistema Solar	25
5	Extras	26
5.1	Câmera Parametrizada	26
5.2	Homem de Neve	26
5.3	Árvore de Natal	27

1 Introdução

Concluída a terceira fase deste projeto, demos inicio à quarta e última fase deste projeto. Esta incide mais sobre a iluminação e sobre as texturas de figuras.

Os modelos agora têm associados toda a componente material, assim como texturas, em vez de cor RGB fixa, pelo que foi necessário um tratamento e uma reformulação de uma grande parte da lógica de implementação.

Sendo esta a última fase, tivemos oportunidade de melhorar alguns dos aspectos do projeto, assim como adicionar mais alguns extras para o tornar apelativo.

2 Gerador

Para a introdução da iluminação no projeto, foi preciso calcular as normais de cada ponto, sendo assim necessário a adição de três novas coordenadas ao ponto. Para introduzir também as texturas, foi necessário adicionar mais duas novas coordenadas ao ponto, sendo estas as coordenadas na textura 2D.

2.1 Plane

Para esta figura geométrica, as normais variam consoante o plano. Caso seja um plano como o da imagem abaixo (xOz), as normais correspondem a $(0,1,0)$.

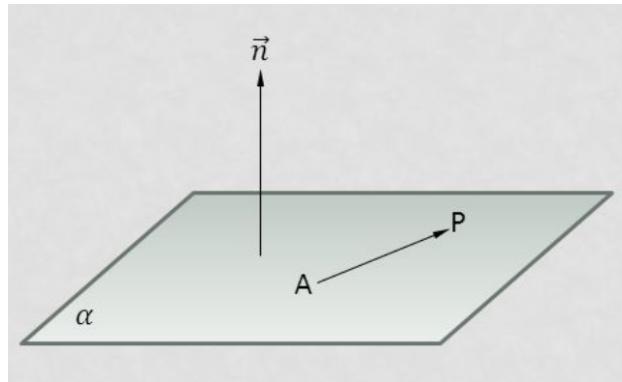


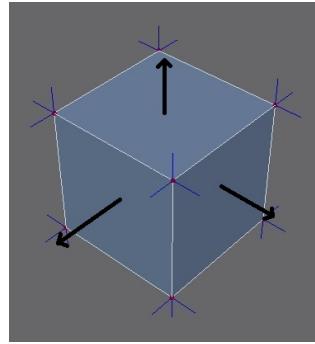
Figura 1: Plano xOz

Se o plano for do tipo xOy , as normais são $(0,0,1)$. Se o plano for do tipo yOz , as normais são $(1,0,0)$.

Quanto às texturas do plano, estas também são diretas, sendo que $(0,0)$ representa o canto esquerdo inferior da textura, $(1,0)$ o canto inferior direito, $(0,1)$ o canto superior esquerdo e $(1,1)$ o canto superior direito.

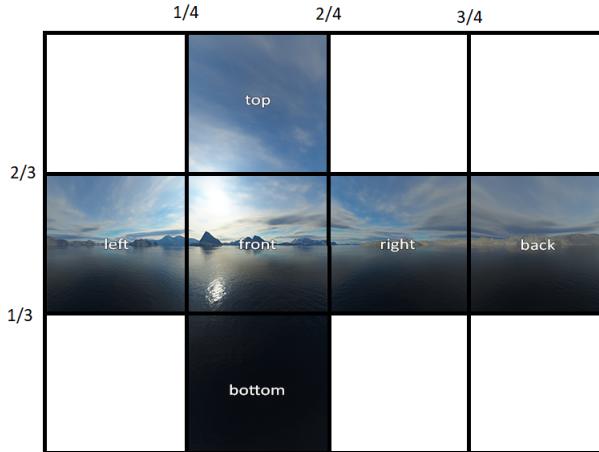
2.2 Box

Para a caixa, as normais são também bastante diretas. Estas variam de face para face como é visível na imagem abaixo.



Assumindo a face do eixo yOx como a face frontal da caixa, as normais dessa face correspondem a $(0,0,1)$, enquanto que a face contrária é $(0,0,-1)$. A face lateral direita é $(1,0,0)$ e a esquerda é $(-1,0,0)$. Por fim, temos a face superior, sendo esta $(0,1,0)$ e a face inferior que corresponde a $(0,-1,0)$.

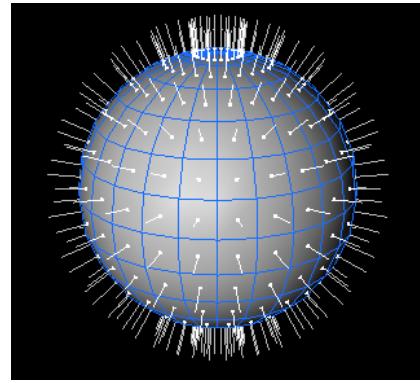
As texturas da caixa são um pouco diferente das texturas do plano, como é visível na imagem seguinte.



Para calcular a posição, sabemos que a textura de cada face corresponde a um quadrado com $1/4$ da largura da imagem da textura e $1/3$ da altura. Por exemplo, a posição da face do topo será de $1/4$ a $2/4$ de largura por $2/3$ a 1 de altura, como é visível na imagem.

2.3 Sphere

Na esfera, o calculo das normais já não é tão direto como na caixa, sendo necessário de calcular as normais em cada ponto, seguindo a seguinte formula $(x/\text{raio}, y/\text{raio}, z/\text{raio})$.



Quanto às texturas, foi necessária a criação de duas novas variáveis, uma que iterava consoante as stacks, denominada de cima, sendo o incremento igual a $1.0/\text{stacks}$ e outra variável que iterava consoante os slices, denominada de lado, sendo o incremento desta igual a $1.0/\text{slices}$. Estas coordenadas representam a posição do ponto atual na imagem 2D da textura, como é possível ver na imagem abaixo.

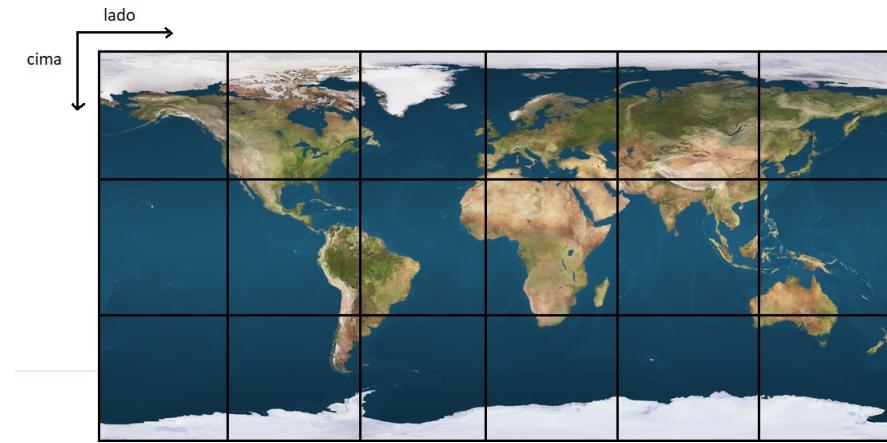
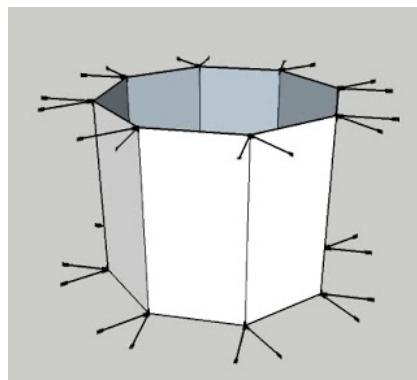


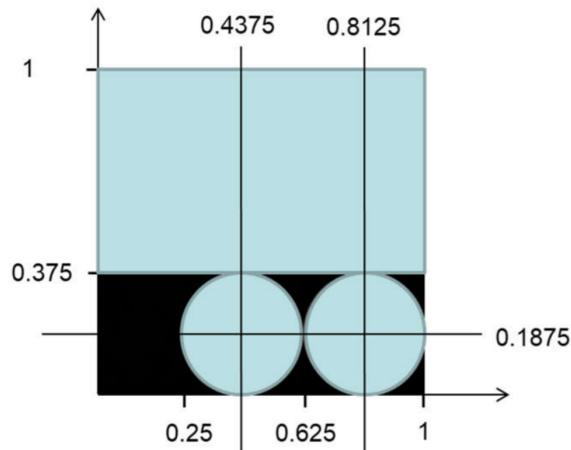
Figura 2: Stacks=3 e Slices=6

2.4 Cylinder

Para a determinação das normais do cilindro nas bases, o cálculo é direto sendo as normais da base superior $(0,1,0)$ e da base inferior $(0,-1,0)$. Já para as laterais as normais calculam-se de ponto para ponto com a seguinte fórmula $(\sin(\text{ang}), 0, \cos(\text{ang}))$.



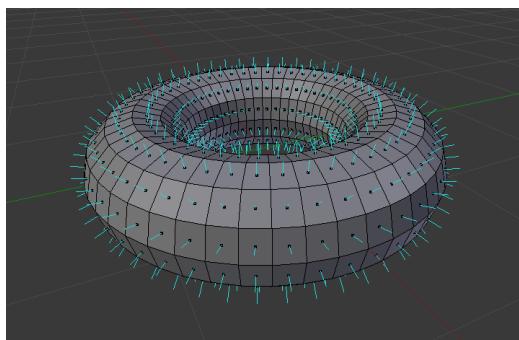
Quanto a texturas, o cilindro é bastante específico, sendo necessário saber os valores de onde se encontram as bases e as laterais. Neste caso os valores são dados pela seguinte figura.



Assim sendo, para as bases, usamos os pontos do centro e o valor do raio. Para as laterais, o valor do x vai de 0 a 1 e o y vai de 0.375 a 1.

2.5 Ring

Para o anel, as normais tem de ser calculadas a cada iteração. Para tal, achamos dois vetores (com os 3 pontos do triângulo) sendo que estes dois vetores contém o mesmo ponto na origem. Depois calcula-se o produto vetorial entre os dois, e normaliza-se. De notar, que a ordem desse produto vetorial é obtida através da regra da mão direita. Assim sendo obtemos os valores das normais referentes a esse triângulo.

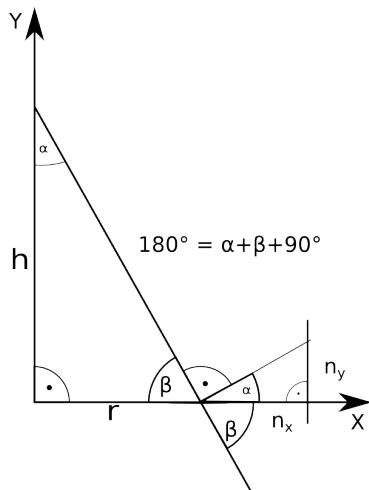


Quanto às texturas desta figura geométrica, foi difícil de arranjar uma que funcionasse perfeitamente. A nossa abordagem passou por calcular os pontos da textura em cada par de triângulos, correspondendo a textura a cada retângulo do anel. De notar que esta resolução apresenta algumas falhas pois a textura em cada retângulo fica ligeiramente diferente. Era necessário ter uma textura como, por exemplo, a do cilindro que já viesse previamente especificada o formato.

2.6 Cone

Para esta figura geométrica as normais da base são diretas, sendo estas $(0, -1, 0)$. Para as laterais, as normais vão ser diferentes de ponto para ponto, mas a inclinação será a mesma em todos. A fórmula é $(\sin(\text{ang}), \text{inclinacao}, \cos(\text{ang}))$, sendo que inclinação corresponde a:

$$\text{inclinacao} = \text{PI} - \text{PI}/2 - \text{atan2}(h, r)$$



2.7 Vase

Para o vaso, as normais das bases são diretas, tais como no cilindro. Assim sendo a base superior é $(0, 1, 0)$ e a inferior é $(0, -1, 0)$. Para as laterais, a fórmula é a mesma que a das laterais do cone, mas com uma leve diferença, o raio utilizado para calcular o ângulo é a diferença entre o raio da base superior, com o raio da base inferior.

2.8 Teapot

Para o calculo das normais do teapot, recorremos às seguintes fórmulas:

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

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

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

Assim sendo, ao calcular as coordenadas de cada ponto aproveitamos para calcular também as normais, pois as fórmulas são relativamente parecidas.

Para o calculo das texturas do teapot, dividimos a textura pelo numero de patches, ou seja, o valor da largura e altura a cada patch será:

```
tex = 1.0 / numPatchs * patch;
largura = tex + v;
altura = tex + u;
```

3 Motor

3.1 Luzes

Foi criada, em primeiro lugar, a classe **Light** definida hierarquicamente, isto é, funciona como classe abstrata para as subclasses, que são a **SpotLight**, **PointLight** e **DirectionalLight**. A classe **Light** implementa um método *virtual* que define que este é implementado nas sub-classes, ou seja, todos os parâmetros adicionais são colocados nas subclasses.

```
1 class Light {
2 public:
3     GLenum id; // id unico
4     float* pos; // posicao caso seja do tipo Spot ou do tipo
5         Point ou direcao caso seja Directional
6     float* diff; // componente difusa
7     float* amb; // componente ambiente
8     float* spec; // componente especular
9     (...)

10    void create() {
11        glLightfv(GL_LIGHT0 + this->id, GL_POSITION, this->pos);
12        glLightfv(GL_LIGHT0 + this->id, GL_AMBIENT, this->amb);
13        glLightfv(GL_LIGHT0 + this->id, GL_DIFFUSE, this->diff);
14        glLightfv(GL_LIGHT0 + this->id, GL_SPECULAR, this->spec);
15    }
16
17    virtual void start(){} // a que é implementada nas
18        subclasses
19
20    class DirectionalLight: public Light{
21 public:
22     (...)

23     void start() {
24         this->create(); // é muito semelhante a classe principal,
25             so muda o pos[3] que fica 0 em vez de 1
26     }
27 };
28
29 class PointLight: public Light {
30 public:
31     float quad_att; // atenuacao quadratica
32     (...)
```

```

33     void start() {
34         this->create();
35         glLightfv(GL_LIGHT0 + this->id, GL_QUADRATIC_ATTENUATION,
36             &(this->quad_att));
37     }
38 }
39 class SpotLight: public Light {
40 public:
41     float* dir; // direcao dos feixes
42     float cutoff, exponent; // angulo de cutoff [0,90] ou 180,
43         valor do expoente [0,128]
44     (...)

45     void start() {
46         this->create();
47         glLightfv(GL_LIGHT0 + this->id, GL_SPOT_DIRECTION, this->
48             dir);
49         glLightfv(GL_LIGHT0 + this->id, GL_SPOT_EXPONENT, &(this
50             ->exponent));
51         glLightfv(GL_LIGHT0 + this->id, GL_SPOT_CUTOFF, &(this->
52             cutoff));
53     }
54 };

```

Agora, no **Motor**, é necessário guardar quantas luzes existiam no ficheiro *input* XML, para serem aplicadas em todas as iterações do programa, **vector<Light*>* lights**. Tiramos partido do polimorfismo permitido pelo C++ para implementar desta forma.

3.2 Texturas, Componente Material e Iluminação nos Objetos

Para as texturas, na classe **Model**, foi necessário reformular para admitir tanto texturas como a componente material.

```

1 class Model {
2     vector<Point> points; // nao foi mudado
3     float *diff, *amb, *spec, *emiss; // componente material,
4         diff e a componente difusa, amb e a ambiente, spec e a
5         especular e emiss e a emissiva
6     float shininess; // "shininess" do objeto
7     string file; // ficheiro da textura (sera string vazia se
8         nao tiver textura)

```

```

6   GLuint vboID; // nao foi mudado
7   (...)
```

Isto implica ter que tratar um ficheiro de textura para podermos usar dentro do programa, para tal usamos a API **DevIL**, que se verifica dentro do método *getTexture*, parte da classe **Model**.

```

1 int getTexture() {
2     unsigned int t, tw, th;
3     unsigned char* texData;
4     unsigned int texID;
5
6     ilInit();
7     ilEnable(IL_ORIGIN_SET);
8     ilOriginFunc(IL_ORIGIN_LOWER_LEFT);
9     ilGenImages(1, &t);
10    ilBindImage(t);
11    ilLoadImage((ILstring) this->file.c_str());
12    tw = ilGetInteger(IL_IMAGE_WIDTH);
13    th = ilGetInteger(IL_IMAGE_HEIGHT);
14    ilConvertImage(IL_RGBA, IL_UNSIGNED_BYTE);
15    texData = ilGetData();
16
17    glGenTextures(1, &texID);
18
19    glBindTexture(GL_TEXTURE_2D, texID);
20    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT
21        );
21    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT
22        );
23
23    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
24        GL_LINEAR);
24    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
25        GL_LINEAR_MIPMAP_LINEAR);
26
26    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, tw, th, 0, GL_RGBA,
27        GL_UNSIGNED_BYTE, texData);
27    glGenerateMipmap(GL_TEXTURE_2D);
28
28    glBindTexture(GL_TEXTURE_2D, 0);
29
31    return texID;
32 }
```

As duas outras funções que receberam mudanças foram a *prepareModel*

e a *drawModel* tanto para incluirem a textura como a componente material, assim como as normais para as luzes funcionarem corretamente.

```

1 void prepareModel(GLuint* buffer, GLuint* normal, GLuint*
2 textures, int* ids) { // // agora recebe 3 novos
3   argumentos, que sao o array de indices dos VBOs para as
4   normais, texturas e um array que indica o ID associado a
5   cada textura
6   int size = this->points.size();
7   float* p = new float[size * 3]; // nao mudou
8   float* n = new float[size * 3]; // array par as normais
9   float* t = new float[size * 2]; // array para as texturas
10  int i = 0, tx = 0;
11  for (int j = 0; j < size; j++) {
12    p[i] = this->points.at(j).x;
13    p[i + 1] = this->points.at(j).y;
14    p[i + 2] = this->points.at(j).z;
15
16    n[i] = this->points.at(j).nx;
17    n[i + 1] = this->points.at(j).ny; // desdobra os pontos
18      para retirar as normais
19    n[i + 2] = this->points.at(j).nz;
20
21
22    glBindBuffer(GL_ARRAY_BUFFER, buffer[vboID]); // nao mudou
23    glBufferData(GL_ARRAY_BUFFER, sizeof(float) * size * 3, p,
24                  GL_STATIC_DRAW);
25
26    glBindBuffer(GL_ARRAY_BUFFER, normal[vboID]); // criar o
27      buffer para as normais e enviar para a placa grafica
28    glBufferData(GL_ARRAY_BUFFER, sizeof(float) * size * 3, n,
29                  GL_STATIC_DRAW);
30
31    glBindBuffer(GL_ARRAY_BUFFER, textures[vboID]); // // criar
32      o buffer para as texturas e enviar para a placa grafica
33    glBufferData(GL_ARRAY_BUFFER, sizeof(float) * size * 2, t,
34                  GL_STATIC_DRAW);
35
36    if (this->file.compare("") != 0) { // se tiver textura
37      int txid = getTexture(); // vai busca-la ao ficheiro

```

```

            respetivo
33     ids[vboID] = txid; // atualiza o vetor de ID s para
            conter esse ID gerado pela funcao anterior
34 }
35 else {
36     ids[vboID] = -1; // diz que o objeto nao tem textura
37 }
38 }
39
40 void drawModel(GLuint* buffer, GLuint* normal, GLuint*
    textures, int* ids) { // agora recebe 3 novos argumentos,
    que sao o array de indices dos VBOs para as normais,
    texturas e um array que indica o ID associado a cada
    textura
41
42 float* amb2 = new float[4];
43 float* diff2 = new float[4];
44 float* spec2 = new float[4];
45 float* emiss2 = new float[4];
46 amb2[0] = amb2[1] = amb2[2] = 0.2f; amb2[3] = 1.0f;
47 diff2[0] = diff2[1] = diff2[2] = 0.8f; diff2[3] = 1.0f; //
            valores por defeito para a componente material
48 spec2[0] = spec2[1] = spec2[2] = 0.0f; spec2[3] = 1.0f;
49 emiss2[0] = emiss2[1] = emiss2[2] = 0.0f; emiss2[3] = 1.0f;
50
51 glBindBuffer(GL_ARRAY_BUFFER, buffer[vboID]); // nao mudou
52 glVertexPointer(3, GL_FLOAT, 0, 0);
53
54 glBindBuffer(GL_ARRAY_BUFFER, normal[vboID]); // buscar o
            buffer das normais
55 glNormalPointer(GL_FLOAT, 0, 0);
56
57 glBindBuffer(GL_ARRAY_BUFFER, textures[vboID]); // buscar o
            buffer da textura
58 glTexCoordPointer(2, GL_FLOAT, 0, 0);
59
60 glMaterialfv(GL_FRONT, GL_AMBIENT, this->amb);
61 glMaterialfv(GL_FRONT, GL_DIFFUSE, this->diff); // colocar
            os valores do material acedendo aos valores
62 glMaterialfv(GL_FRONT, GL_SPECULAR, this->spec);
63 glMaterialfv(GL_FRONT, GL_EMISSION, this->emiss);
64
65 if (ids[vboID] == -1) { // se nao tiver textura
66     glDrawArrays(GL_TRIANGLES, 0, points.size());
67 }

```

```

68     else { // se tiver textura
69         glBindTexture(GL_TEXTURE_2D, ids[vboID]); // vai buscar a
70             textura com dado ID do VBO
71         glDrawArrays(GL_TRIANGLES, 0, points.size()); // desenha
72             a figura
73         glBindTexture(GL_TEXTURE_2D, 0); // limpar o apontador
74     }
75
76     glMaterialfv(GL_FRONT, GL_AMBIENT, amb2);
77     glMaterialfv(GL_FRONT, GL_DIFFUSE, diff2); // valores por
78         defeito para recomendar o processo
79     glMaterialfv(GL_FRONT, GL_SPECULAR, spec2);
80     glMaterialfv(GL_FRONT, GL_EMISSION, emiss2);
81 }
```

3.3 Ficheiro de XML e Interpretador

Para garantir que possuímos texturas, cores, luzes e câmara dadas pelo utilizador, o leitor de XML tem de suportar novos campos nos ficheiros, a ser explorados de seguida.

3.3.1 Luzes

As luzes têm várias componentes para a cor, como já visto em cima, que são a **especular**, **ambiente** e **difusa**. Para além destas, temos os vários tipos de luz (ver secção 3.1) que têm os seus campos específicos. Decidimos, então, o seguinte novo elemento principal para o ficheiro XML:

```
<lights>
    <light type="DIRECTIONAL" posX="..." posY="..." posZ="..."
        diffR="..." diffG="..." diffB="..." ambR="..." ambG="..."
        ambB="..." specR="..." specG="..." specB="..."/>
    <light type="POINT" posX="..." posY="..." posZ="..." diffR=
        "..." diffG="..." diffB="..." ambR="..." ambG="..." ambB=".-.
        ..." specR="..." specG="..." specB="..." attenuation="..."/>
    <light type="SPOT" posX="..." posY="..." posZ="..."
        diffR="..." diffG="..." diffB="..." ambR="..." ambG="..."
        ambB="..." specR="..." specG="..." specB="..." dirX="...
        dirY="..." dirZ="..." cutoff="..." exponent="..."/>
    (...)
```

```
</lights>
```

O programa atualmente aceita no máximo 8 luzes (limitação do próprio OpenGL), em que todos os atributos acima representados por reticências são *floats* e totalmente opcionais (evidente que serão colocados valores por defeito caso não sejam definidos), à exceção do type que tem de estar definido obrigatoriamente.

Para evitar a verbosidade introduzida pelo *tinyXML2*, essencialmente o processo de leitura do ficheiro de XML quando encontra a tag **lights** para um novo elemento, começa o processamento das luzes. Caso não seja encontrado, não existem luzes no modelo.

Percorrendo todos os sub-elementos do elemento **lights**, o algoritmo procura saber primeiro o *type* da luz. Sabendo o tipo, gera a classe respetiva (secção 3.1) e faz o tratamento dos valores possíveis para esse tipo de luz. Se o *type* for escrito incorretamente / não existir será ignorado o elemento que o contém e passará para o processamento do próximo.

3.3.2 Câmara

A ideia foi criar uma forma mais fácil do utilizador alterar as configurações da câmara exploradora através do ficheiro de XML, que será visto com mais detalhe em 5.1.

Novamente, para evitar a verbosidade do *tinyXML2*, o que o algoritmo está a fazer essencialmente é a colocar, por defeito, valores para todos os campos da câmara. Caso consiga, efetivamente, procurar esses valores na definição da câmara, são substituídos e no final é retornada a nova **Camera**, que é uma classe nova criada para este propósito.

```
1 class Camera {
2 public:
3     float rad; // raio da esfera da camara exploradora
4     float sensitivity; // sensibilidade para o movimento
5     float* pos; // posicao na esfera
6     float* la; // ponto para o qual a camara esta a apontar
7     float* up; // vetor que apresenta a direcao "cima" da
8         camara
9     (...)
```

3.3.3 Componente Material e Texturas

Para implementarmos as funcionalidades descritas no capítulo anterior, foi necessário os **models** também terem os atributos necessários, pelo que já não basta ter o **file** é preciso ter em conta a textura e a componente material, que são ambos opcionais.

A componente material é dividida nas seguintes componentes: **especular**, **ambiente**, **difusa** e **emissiva** e tem uma propriedade que é a **shininess**. É esta, então, a estrutura do **model**:

```
<models>
    <model file=<file>.3d texture=<image> diffR="..." diffG-
        ="..." diffB="..." ambR="..." ambG="..." ambB="..." specR=
        "..." specG="..." specB="..." emissR="..." emissG="..." emissB="..." shininess="..."/>
    (...)
```

</models>

Todos os atributos representados por reticências são *floats* e são opcionais, caso não esteja presente um deles, será assumido um valor por defeito para esse atributo.

Novamente, para simplificar o processo do *tinyXML2*, o algoritmo essencialmente o que faz é para além de procurar o atributo obrigatório **file**, procura os opcionais indicados acima. Se não encontrar textura, é criado um modelo simples, caso contrário é utilizado o modelo "complexo".

3.4 Alterações Adicionais

Na fase anterior tínhamos funções que permitiam a construção de eixos e da construção visual da rota dada pelas curvas Catmull-Rom utilizando para isso a primitiva *glColor3f*, que com a luz e componentes materiais não funciona como é esperado.

Para isso foi necessário fazer as seguintes alterações sempre que precisávamos de a utilizar:

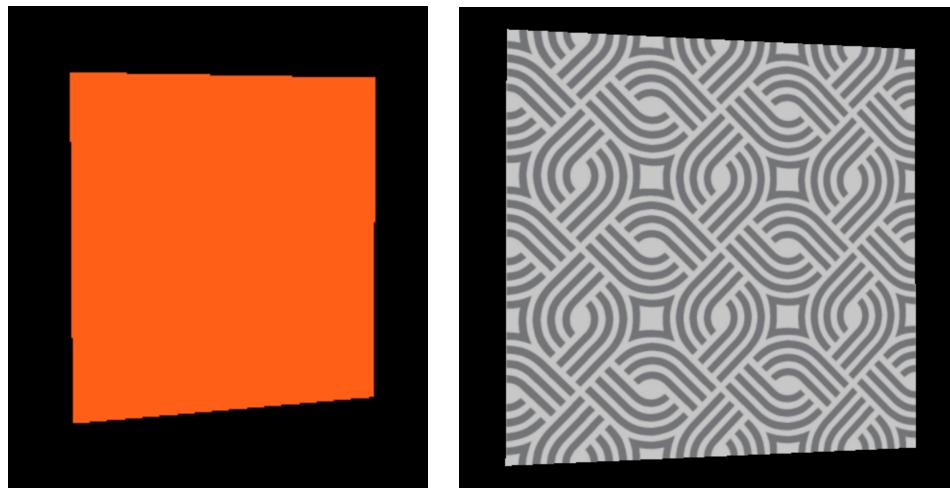
1. Antes de chamar *glColor3f*, ativa-se o **GL_COLOR_MATERIAL** com o comando *glEnable(GL_COLOR_MATERIAL)*;
2. Faz-se o(s) *glColor3f* para a cor pretendida;

3. Faz-se o $glColor3f(1.0f, 1.0f, 1.0f)$ para voltar para a cor original.
4. Desativa-se o que foi ativado em 1) com o comando $glDisable(GL_COLOR_MATERIAL)$.

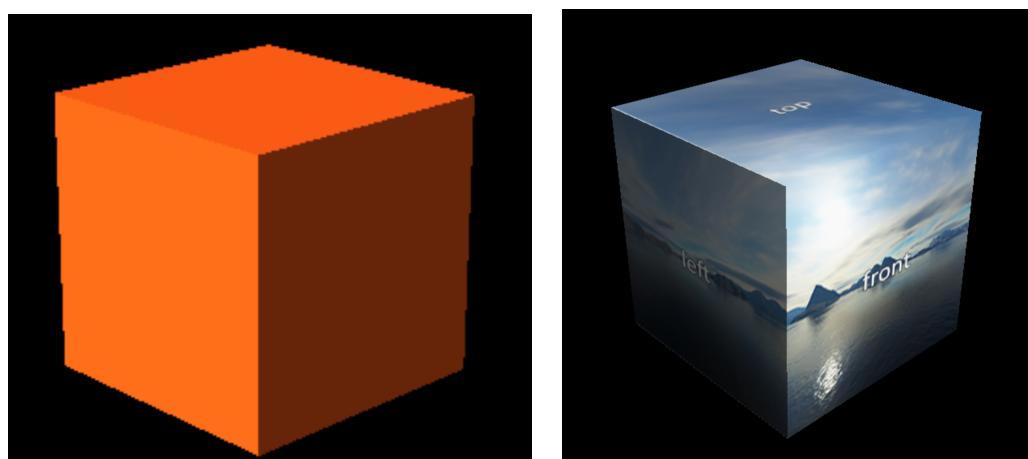
Para além disso, agora é dada a possibilidade ao utilizador de escolher se o objeto sofre uma rotação consoante a sua trajetória na curva de Catmull-Rom ou se a ignora. É dado através do atributo **follows** no elemento `<translate>` dinâmico e terá valor *yes* ou *no*, se for o primeiro fará a rotação segundo a curva, se for o segundo não o fará. O valor por defeito é tomado como se fosse *no*, pois o campo é opcional.

4 Resultado Final

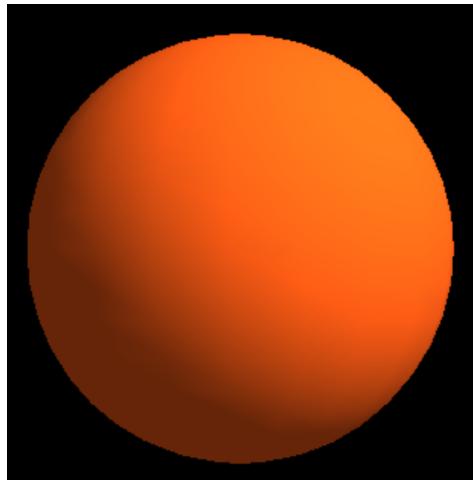
4.1 Plane



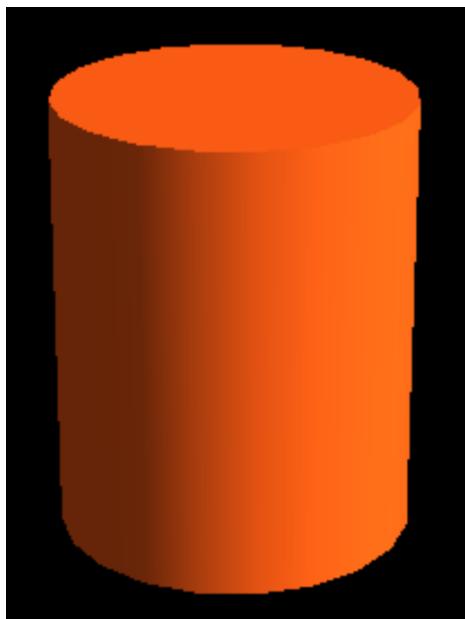
4.2 Box



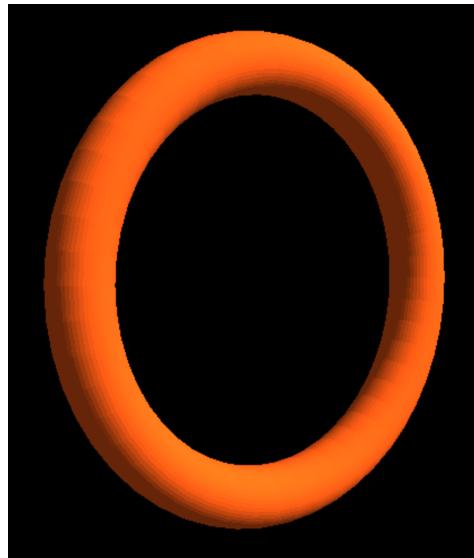
4.3 Sphere



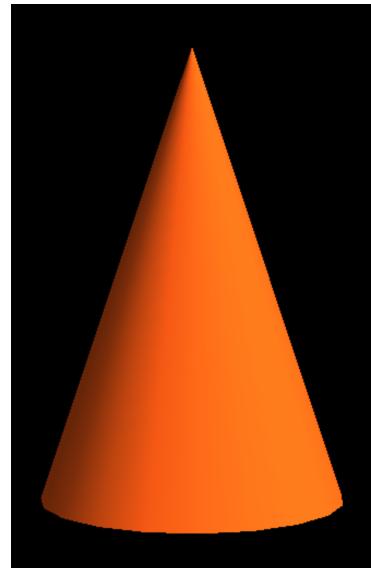
4.4 Cylinder



4.5 Ring



4.6 Cone



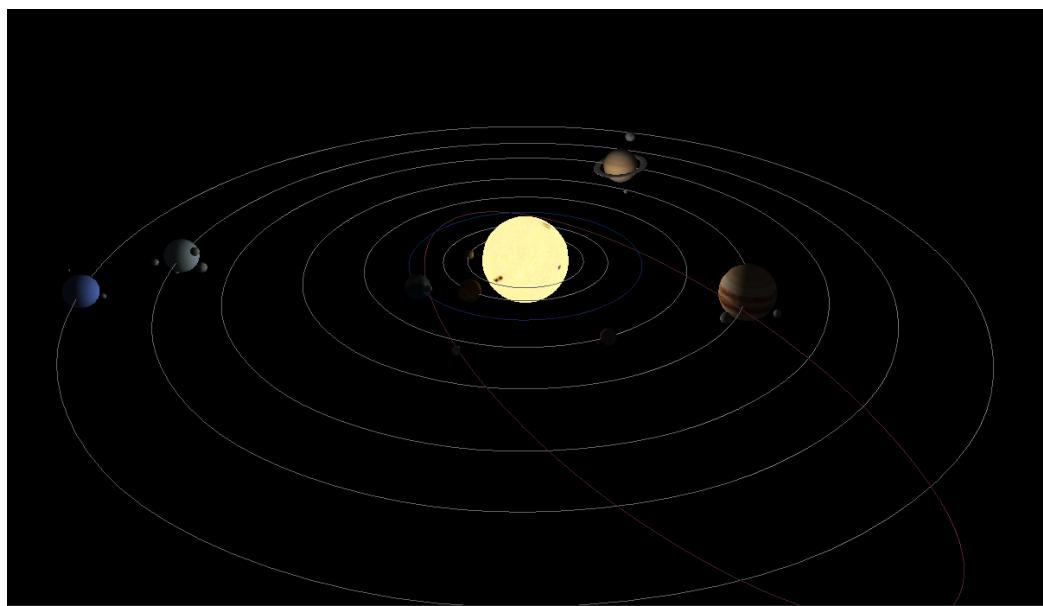
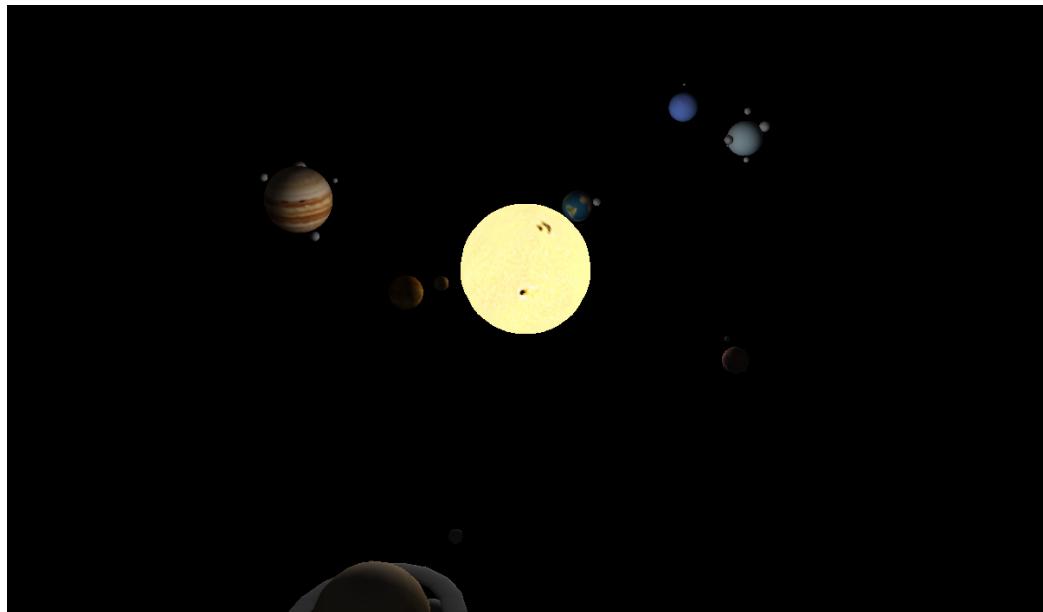
4.7 Vase



4.8 Teapot



4.9 Sistema Solar



5 Extras

5.1 Câmara Parametrizada

Para esta fase decidimos que era uma boa altura para incluir algo que desse mais liberdade ao utilizador. O modo por defeito deste projeto, como visto nas fases anteriores, é o modo de explorador, porém, não era dado ao utilizador a liberdade de mudar os campos, já vinham *hard-coded* no nosso projeto.

Agora, com uma nova classe **Camera**, é possível ler do ficheiro XML uma câmara definida pelo utilizador (limitada ao modo explorador). Caso não exista essa linha de configuração da câmara, é assumida uma por defeito.

O novo elemento aceitado pelo XML é o seguinte (todos os campos são opcionais):

```
<camera>
    <position x="..." y="..." z="..."/>
    <lookat x="..." y="..." z="..."/>
    <up x="..." y="..." z="..."/>
    <radius value="..."/>
    <sensitivity value="..."/>
</camera>
```

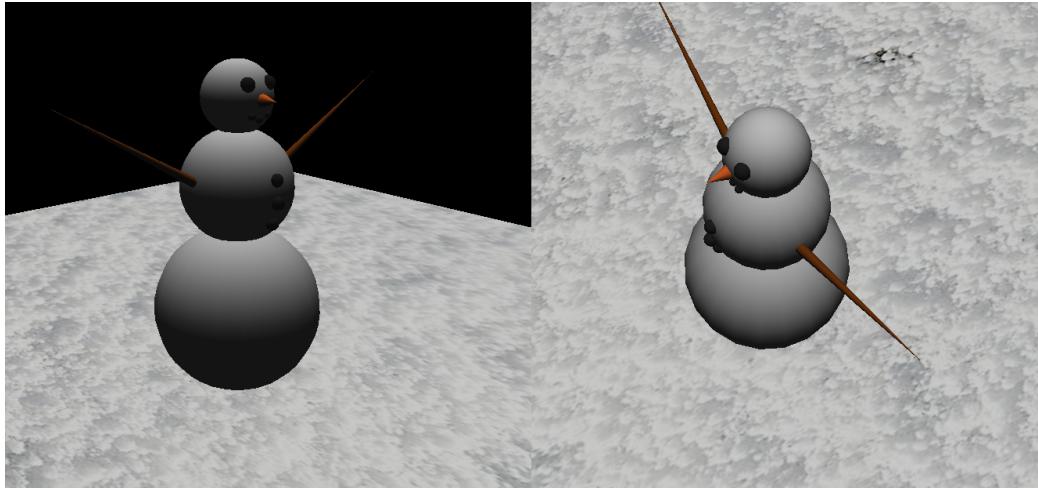
O elemento **position** diz respeito à posição da câmara, o **lookat** indica o ponto para o qual a câmara está a apontar, o **up** indica o vetor *up* para determinar o eixo dos yy da câmara. Estes valores são utilizados no *gluLookAt*.

Por outro lado, o **radius** é específico de se tratar duma câmara modo explorador e indica o raio da esfera cuja câmara está localizada. Consoante se pretenda que a câmara reaja mais rapidamente ou mais lentamente, alterase o valor de **sensitivity**. Um maior valor significa que a câmara, por cada movimento do rato, se desloque uma maior distância do que uma câmara cujo valor de **sensitivity** seja mais baixo.

5.2 Homem de Neve

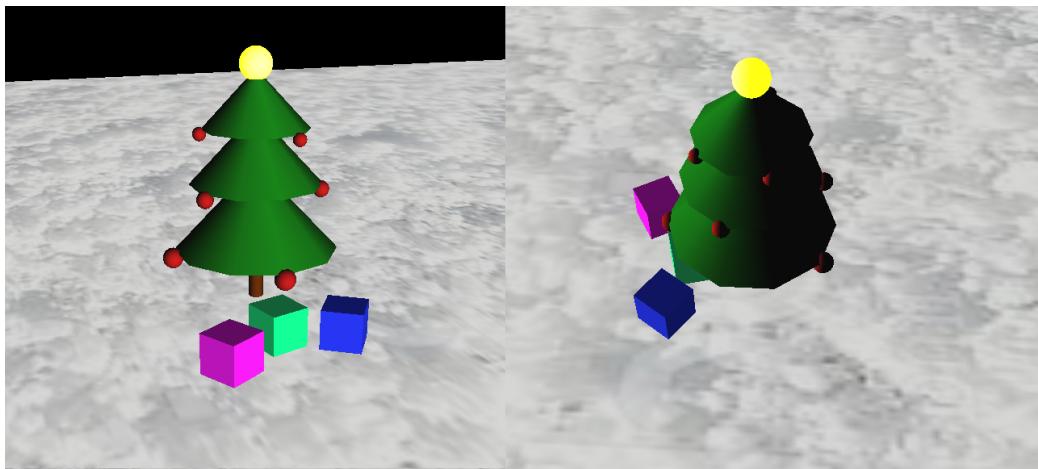
Para testar a capacidade e o funcionamento das luzes, recriamos as seguintes cenas da **Fase 2** para suportarem luzes/texturas. No caso desta em

específico, utiliza uma luz do tipo *spotlight* diretamente em cima do homem de neve.



5.3 Árvore de Natal

Para esta cena, também recriada da **Fase 2**, decidimos testar com outro tipos de luzes, a *direcional* (a direção é no sentido da frente da árvore). Testar a cor dos objetos utilizando a componente material foi o principal foco desta demonstração.



6 Conclusão

Em suma, nesta fase final do trabalho, o grupo considera que já aparenta ter um conhecimento da matéria mais aprofundado sobre texturas e iluminação em OpenGL.

Conseguimos implementar tudo o que foi pedido e de forma eficiente, pelo que o programa funciona com iluminação e texturas sem saber qual a forma da figura para o qual o está a fazer, que era o objetivo inicial de todo o projeto.

Por fim, terminada esta quarta fase, o objetivo será começar aprofundar ainda mais o conhecimento em OpenGL porque sentimos que apenas "tocamos na superfície" de tudo o que é possível fazer com esta ferramenta.