Universidade do Minho

Mestrado Integrado em Engenharia Informática

Computação Gráfica

2ª Fase - Transformações Geométricas

Grupo 36

Gonçalo Esteves
.(A85731).

João Araújo
.(A84306).

Mário Real
.(A72620).

Rui Oliveira
.(A83610).

31 de Maio de 2020

Conteúdo

Resumo

Quarta fase do trabalho prático realizado no âmbito da Unidade Curricular de Computação Gráfica da Universidade do Minho. Esta fase consiste na adição de novas funcionalidades e modificações ao trabalho já realizado, tais como o uso de texturas e a aplicação de iluminação, por forma a obter representações mais realistas dos sistemas a representar.

1 Introdução

Neste documento vamos esclarecer os aspetos mais importantes relativamente à quarta fase do nosso trabalho, onde faremos uma síntese e explicação do código elaborado e resultados obtidos.

Iremos também descrever e explicar as abordagens que tomamos quanto às modificações do gerador e do *engine*, bem como o seu funcionamento e os diversos raciocínios utilizados para ultrapassar as barreiras encontradas, apresentando, também, algumas imagens de exemplo para facilitar a compreensão.

2 Gerador

Nesta secção iremos explicar as alterações implementadas no nosso gerador, bem como as novas funcionalidades implementadas.

Novamente, este componente é responsável por gerar os pontos dos triângulos que permitem a construção das diversas figuras geométricas. No entanto, mais uma vez, sofreu alterações com a implementação desta fase, principalmente devido à inclusão das texturas.

2.1 *Figures*

Tendo em vista o desenvolvimento desta fase, tornou-se necessário determinar as normais e as coordenadas das texturas das figuras elaboradas nas fases anteriores.

- **Plano**
  - **Normais**
    Uma vez que o plano se encontra definido no plano xOz, é fácil determinar que o vetor normal em cada vértice do plano e igual a (0, -1, 0).

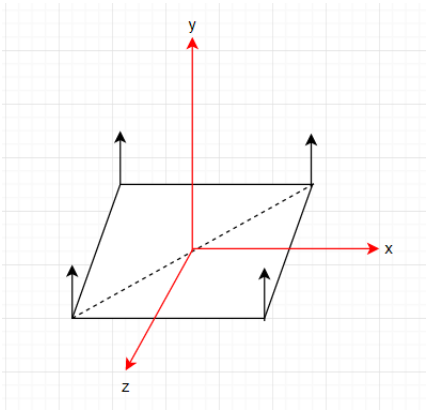

Figura 1: Normais aos vértices do plano

  - **Texturas**
    Relativamente às coordenadas das texturas, estas deverão estar incluídas na àrea ocupada pelos triângulos que formam o plano. Estas são representadas em seguida, sendo fácil observar quais as coordendas da textura que irão estar associadas a cada um dos vértices do plano.

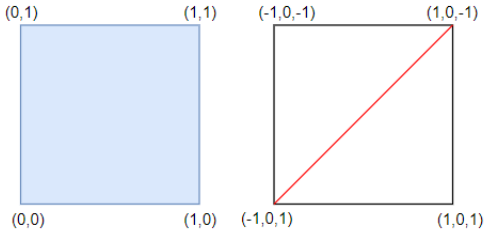

Figura 2: Coordenadas das texturas (à esquerda) e coordenadas dos vértices do plano (à direita)

- **Esfera**
  - **Normais**
    O cálculo das normais da esfera é realizado tendo em conta a origem da mesma e um qualquer ponto que esteja presente nela. Uma vez que a própria representação da esfera já se baseava neste pressuposto, podemos concluir que o valor das normais da figura poderá corresponder às coordenadas destes pontos. Ou seja, o vetor das normais poderá ser representado como $\overrightarrow{N} = P - O = P$

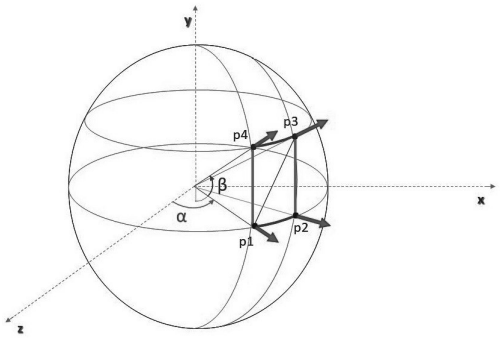

Figura 3: Normais aos vértices da esfera

Por fim, é necessário normalizar o vetor, recorrendo, para isso, à fórmula $\overrightarrow{N} = \frac{P}{||P||}$.

  - **Texturas**
    Criadas as normais, torna-se necessário definir as coordendas das texturas. Uma vez que os vértices da esfera são gerados por camadas, iremos aplicar o mesmo procedimento à geração das coordenadas das texturas, considerando que cada camada de esfera corresponde a uma camada de textura. Tal será gerado a partir do seguinte excerto de código:

```
for (int i = 0; i < stacks; i++){

    beta1 = (float)i*(float)(M_PI/stacks)-(float)M_PI_2;
    beta2 = (float)(i+1)*(float)(M_PI/stacks)-(float)M_PI_2;

    for (int j = 0; j < slices; j++) {

        alpha1 = (float)j*2*(float)M_PI/(float)slices;
        alpha2 = (float)(j+1)*2*(float)M_PI/(float)slices;

        /*geracao dos pontos dos triangulos atraves das
    coordenadas esfericas alpha1/2 e beta1/2 e das respetivas
     normais*/
```

```
13          (*texture).push_back((float)j/(float)slices);
            (*texture).push_back((float)(i+1)/(float)stacks);
15          (*texture).push_back((float)j/(float)slices);
            (*texture).push_back((float)i/(float)stacks);
17          (*texture).push_back((float)(j+1)/(float)slices);
            (*texture).push_back((float)(i+1)/(float)stacks);
19          (*texture).push_back((float)(j+1)/(float)slices);
            (*texture).push_back((float)(i+1)/(float)stacks);
21          (*texture).push_back((float)j/(float)slices);
            (*texture).push_back((float)i/(float)stacks);
23          (*texture).push_back((float)(j+1)/(float)slices);
            (*texture).push_back((float)i/(float)stacks);
25      }
}
```

- **Cone**
  - **Normais**
    De modo a calcular os vetores normais ao cone, deveremos distinguir os vetores dos pontos da base, dos vetores da superfíce lateral. Os primeiros, devido ao facto de que a base se encontra contida em xOz, serão simplesmente (0, -1, 0). Quanto aos outros, a sua determinação é realizada em dois passos. Primeiro, determinamos os vetores normais para uma circunferência paralela à base, como a representada em seguida:

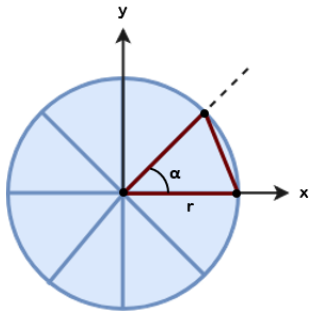

Figura 4: Normais aos pontos de um círculo

Por fim, necessitamos de ter em conta a inclinação do cone, por nós representada como θ. Para obter o seu valor, necessitamos de considerar o raio r da circunferência bem como a altura h a que está da base. Posto isto, podemos chegar à seguinte expressão:

$$tan(\theta) = \frac{h}{r} \Rightarrow \theta = arctan(\frac{r}{h})$$

Posto tudo isto, podemos concluir que um ponto da lateral do cone terá como normal $\overrightarrow{N} = (cos(\alpha) \times cos(\theta), sin(\theta), sin(\alpha) \times cos(\theta))$.

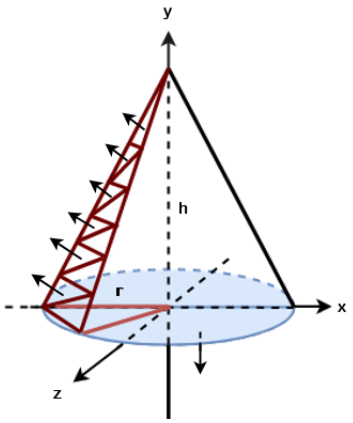

Figura 5: Normais aos pontos de um cone

– **Texturas**
Por forma a aplicar as texturas, começamos por definir o formato que estas terão, representado em seguida.

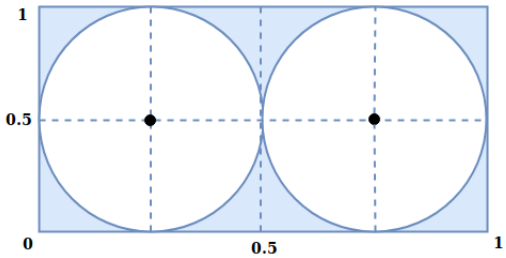

Figura 6: Formato da textura

Deste modo, optamos por definir a circunferência da esquerda como sendo a representação da base do cone, enquanto que a da direita representará a sua lateral. Posto isto, construímos o seguinte código, de forma a criar as texturas de um cone:

```
for (int i = 0; i < slices; i++){
    teta = (float)i*alpha;
    tetaNext = (float)(i+1)*alpha;

    //geracao dos pontos e das suas normais

    (*texture).push_back(0.25f);
    (*texture).push_back(0.5f);
```

```
    (*texture).push_back(0.25f+cos(tetaNext)/4.0f);
    (*texture).push_back(0.5f+sin(tetaNext)/2.0f);
    (*texture).push_back(0.25f+cos(teta)/4.0f);
    (*texture).push_back(0.5f+sin(teta)/2.0f);
}

angle = atan(radius/height);

for (int i = 0; i < stacks; i++){
    heightNow = (float)i*scaleH;
    heightNext = (float)(i+1)*scaleH;
    radiusNow = radius-(float)i*scaleR;
    radiusNext = radius-(float)(i+1)*scaleR;

    for (int j = 0; j < slices; j++) {
        teta = (float)j*alpha;
        tetaNext = (float)(j+1)*alpha;

        //geracao dos pontos e das suas normais

        res = (float)(stacks-i)/(float)stacks;
        resNext = (float)(stacks-(i+1))/(float)stacks;

        (*texture).push_back(0.75f+0.25f*cos(teta)*res);
        (*texture).push_back(0.5f+0.5f*sin(teta)*res);
        (*texture).push_back(0.75f+0.25f*cos(tetaNext)*res);
        (*texture).push_back(0.5f+0.5f*sin(tetaNext)*res);
        (*texture).push_back(0.75f+0.25f*cos(tetaNext)*
    resNext);
        (*texture).push_back(0.5f+0.5f*sin(tetaNext)*resNext
    );
        (*texture).push_back(0.75f+0.25f*cos(teta)*res);
        (*texture).push_back(0.5f+0.5f*sin(teta)*res);
        (*texture).push_back(0.75f+0.25f*cos(tetaNext)*
    resNext);
        (*texture).push_back(0.5f+0.5f*sin(tetaNext)*resNext
    );
        (*texture).push_back(0.75f+0.25f*cos(teta)*resNext);
        (*texture).push_back(0.5f+0.5f*sin(teta)*resNext);
    }
}
```

Neste caso, o primeiro ciclo representa a criação das texturas da base, enquanto que o segundo ciclo, que possui um ciclo aninhado, representa a criação das texturas da face lateral.

- **Cilindro**
  - **Normais**
    Por forma a calcular os vetores normais, consideremos o cilindro dividido em três partes: o topo, a base, e a face lateral.
    Deste modo, torna-se fácil definir a normal aos pontos do topo como sendo (0, 1, 0) e aos da base como sendo (0, -1, 0). Quanto à face lateral, a normal a um ponto desta será definida por $(sin(\theta), 0, cos(\theta))$, onde θ é a amplitude em que se encontra o vértice.

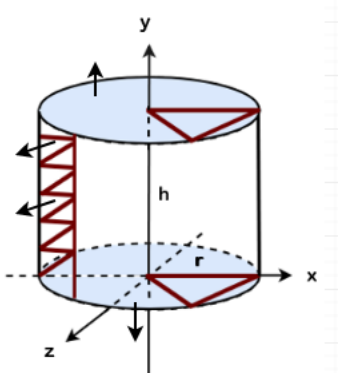

Figura 7: Normais aos pontos de um cilindro

  - **Texturas**
    Tal como acontece com o cone, por forma a obter as coordenadas das texturas do cilindro foi necessário definir a sua planificação, podendo esta ser observada em seguida:

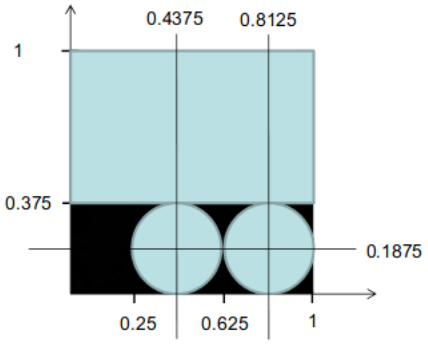

Figura 8: Formato da textura

Consideramos também o topo como sendo a circunferência à esquerda e a base a da direita. Tendo tudo isto em consideração, elaboramos o seguinte código que visa a obter os valores para as coordenadas das texturas.

```
for(int i = 0; i < slices; i++){

    teta = (float)i*alpha;
    tetaNext = (float)(i+1)*alpha;

    //geracao dos pontos da base e das suas normais

    (*texture).push_back(0.8125f);
    (*texture).push_back(0.1875f);
    (*texture).push_back(0.8125f+0.1875f*sin(teta+alpha));
    (*texture).push_back(0.1875f+0.1875f*cos(teta+alpha));
    (*texture).push_back(0.8125f+0.1875f*sin(teta));
    (*texture).push_back(0.1875f+0.1875f*cos(teta));

    //geracao dos pontos do topo e das suas normais

    (*texture).push_back(0.4375f);
    (*texture).push_back(0.1875f);
    (*texture).push_back(0.4375f+0.1875f*sin(teta));
    (*texture).push_back(0.1875f+0.1875f*cos(teta));
    (*texture).push_back(0.4375f+0.1875f*sin(tetaNext));
    (*texture).push_back(0.1875f+0.1875f*cos(tetaNext));
}

for (int i = 0; i < stacks; i++) {

    heigthNow = -(height/2)+((float)i*scaleHeigth);
    heigthNext = heigthNow+scaleHeigth;

    for (int j = 0; j < slices; j++) {

        teta = (float)j*alpha;
        tetaNext = (float)(j+1)*alpha;

        //geracao dos pontos do topo e das suas normais

        (*texture).push_back((1.0f/(float)slices)*((float)j)
    );
        (*texture).push_back((float)i*0.625f/(float)stacks
    +0.375f);
        (*texture).push_back((1.0f/(float)slices)*((float)j
    +1));
        (*texture).push_back((float)i*0.625f/(float)stacks
    +0.375f);
```

```
41          (*texture).push_back((1.0f/(float)slices)*(float)(j
    +1));
            (*texture).push_back((float)(i+1)*0.625f/(float)
    stacks+0.375f);
43          (*texture).push_back((1.0f/(float)slices)*((float)j)
    );
            (*texture).push_back((float)i*0.625f/(float)stacks
    +0.375f);
45          (*texture).push_back((1.0f/(float)slices)*(float)(j
    +1));
            (*texture).push_back((float)(i+1)*0.625f/(float)
    stacks+0.375f);
47          (*texture).push_back((1.0f/(float)slices)*(float)j);
            (*texture).push_back((float)(i+1)*0.625f/(float)
    stacks+0.375f);
49       }
}
```

Assim, o primeiro ciclo representa a criação das coordenadas das texturas da base e do topo do cilindro, enquanto que o segundo ciclo representa a geração das coordendas das texturas da superfície lateral.

- ***Torus***
  - **Normais**
    Por forma a obter as normais do *Torus*, aplicou-se o mesmo raciocínio que utilizado na esfera, em que o próprio ponto corresponde à sua normalização. Desta forma, as expressões ficam iguais às da esfera.

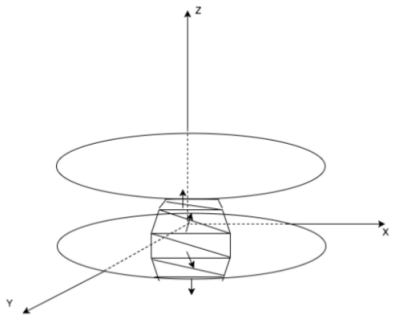

Figura 9: Normais aos pontos de um *torus*

– **Texturas**
Quanto às coordenadas das texturas, cada anel do *torus* corresponde a uma textura, fazendo com que chegando ao fim de todas as iterações tenha sido preenchido cada um destes aneis, estando o *torus* preenchido na totalidade. Deste feita, desenvolvemos o seguinte excerto de código:

```
for (int i = 0; i < layers; i++) {
    beta = (float)i*(float)(2*M_PI/layers);
    nextBeta = (float)(i+1)*(float)(2*M_PI/layers);

    for (int j = 0; j < slices; j++) {
        alpha = (float)j*(float)(2*M_PI/slices);
        nextAlpha = (float)(j+1)*(float)(2*M_PI/slices);

        //geracao dos pontos do topo e das suas normais
        (*texture).push_back((float)j/(float)slices);
        (*texture).push_back((float)i/(float)layers);
        (*texture).push_back((float)(j+1)/(float)slices);
        (*texture).push_back((float)i/(float)layers);
        (*texture).push_back((float)j/(float)slices);
        (*texture).push_back((float)(i+1)/(float)layers);
        (*texture).push_back((float)j/(float)slices);
        (*texture).push_back((float)(i+1)/(float)layers);
        (*texture).push_back((float)(j+1)/(float)slices);
        (*texture).push_back((float)i/(float)layers);
        (*texture).push_back((float)(j+1)/(float)slices);
        (*texture).push_back((float)(i+1)/(float)layers);
    }
}
```

2.2 *Patch*

Por forma a determinar as normais nos ficheiros *patch*, recorremos novamente às matrizes apresentadas em seguida:

- Matrizes a e b

$$a = \begin{bmatrix} a^3 & a^2 & a & 1 \end{bmatrix}$$

$$b = \begin{bmatrix} b^3 & b^2 & b & 1 \end{bmatrix}$$

- Matrizes com Pontos de Controlo

$$P = \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_30 & P_{31} & P_{32} & P_{33} \end{bmatrix}$$

- Matrizes de Bézier

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

Tendo por base os valores de a e b, que se encontram entre 0 e 1, torna-se possível obter um ponto da superfície de *Bézier*, através da seguinte fórmula:

$$P(a, b) = a \times M \times P \times M \times b$$

Por forma a calcular a normal a qualquer ponto P(a, b), tivemos de determinar o produto cruzado das tangentes ao ponto normalizadas que, neste caso, correspondem aos vetores a e b.

Definindo uma função capaz de obter as tangentes (designada de *getTangent*, e aplicando em seguida a função *cross*, que determina o produto cruzado das mesmas, obtemos a normal ao ponto após a normalização do resultado obtido. Isto pode ser observado no excerto de código seguinte:

```
for(int i = 0; i < tesselation; i++){
    for (int j = 0; j < tesselation; j++){
        u = (float)i*t;
        v = (float)j*t;
        uu = (float)(i+1)*t;
        vv = (float)(j+1)*t;

        p0 = getPoint(u,v,coordenadasX,coordenadasY,coordenadasZ);
        tangenteU = getTangent(u, v, coordenadasX, coordenadasY,
    coordenadasZ, 0);
        tangenteV = getTangent(u, v, coordenadasX, coordenadasY,
    coordenadasZ, 1);
        cross(tangenteU, tangenteV, res);
        normalize(res);
        n0 = new Point(res[0],res[1],res[2]);

        /*Calculo dos restantes pontos e normais e das texturas*/
    }
}
```

Definiram-se, também, as texturas, seguindo a mesma ordem que os pontos que foram gerados. Este processo é apresentado em seguida, por meio do código desenvolvido:

```
for(int i = 0; i < tesselation; i++){
    for (int j = 0; j < tesselation; j++){
        u = (float)i*t;
        v = (float)j*t;
        uu = (float)(i+1)*t;
        vv = (float)(j+1)*t;

        /*Calculo dos pontos e das normais*/

        texture->push_back(1-u);  texture->push_back(1-v);
        texture->push_back(1-uu); texture->push_back(1-v);
        texture->push_back(1-u);  texture->push_back(1-vv);
        texture->push_back(1-u);  texture->push_back(1-vv);
        texture->push_back(1-uu); texture->push_back(1-v);
        texture->push_back(1-uu); texture->push_back(1-vv);
    }
}
```

3 Engine

Esta parte do projeto também sofreu alterações, tais como a disponibilização das funcionalidades de iluminação, bem como a aplicação de texturas.

3.1 *Parser*

Devido as alterações efetuadas aos ficheiros XML utilizados, foi necessário alterar também a forma como é efetuado o *parsing* dos mesmos, por forma a representar os modelos de forma correta. De seguida são exemplificados os formatos de escrita das novas funcionalidades no ficheiro XML:

- **Iluminação:** Indicação de todas as luzes a ser usadas.

```
<lights>
    <light type="POINT" posX="0" posY="0" posZ="0" diffR="1"
    diffG="1" diffB="1" />
</lights>
```

- **Texturas:** Indicação da localização do ficheiro com a imagem a utilizar (a textura vem incluída no ficheiro .3d).

```
<models>
    <model file="sphere.3d" texture="../../Textures/moon.jpg" />
</models>
```

- **Materiais:** Materiais associados às primitivas.

```
<models>
    <model file="sphere.3d" diffR="1" diffG="0.9" diffB="0.8" />
</models>
```

Tendo isto em vista, definimos duas funções, *parseLights* e *parseMaterials*, que serão responsáveis por efetuar o *parse* da informação relativa à iluminação e aos materiais, respetivamente. Devido a isto, e também à implementação das texturas, foi necessário alterar a função *parseModel* e várias das suas auxiliares, por forma a realizar o pretendido.

3.2 *Light*

Esta classe foi criada com o intuito de facilitar a iluminação dos objetos, tendo em conta a incidência, total, parcial ou nula de uma luz, cujas informações serão guardadas nesta classe.

- **Cor -** devido à possibilidade de associar variados tipos de cores, tivemos em conta os seguintes casos:
  - *GL_AMBIENT:* intensidade da luminosidade permitida num dado cenário através de uma fonte de luz;
  - *GL_DIFFUSE:* luz direcional projetada por uma fonte que quando atinge outro objeto espalha-se de forma uniforme pela superfície;
  - *GL_SPECULAR:* influencia um dado objeto quanto à sua cor do destaque especular.
- **Posição -** possibilita-nos a distinção entre luzes pontuais e luzes direcionais:
  - *POINT:* as coordenadas retratam a posição da luz, a partir da qual ilumina todas as direções;
  - *DIRECTIONAL:* as coordenadas retratam a direção da luz.

Por forma a processar tudo isto, elaboramos a seguinte função:

```
void Light :: apply(GLenum number){

    // posicao da luz
    glLightfv(GL_LIGHT0 + number, GL_POSITION, info);

    // atributos da luz
    for (const int atr : attributes){
        switch(atr){
            case DIFFUSE:
                glLightfv(GL_LIGHT0+number, GL_DIFFUSE, info + 4);
                break;
            case AMBIENT:
                glLightfv(GL_LIGHT0+number, GL_AMBIENT, info + 8);
                break;
            case SPECULAR:
                glLightfv(GL_LIGHT0+number, GL_SPECULAR, info + 12);
                break;
            default:
                break;
        }
    }
}
```

3.3 *Material*

Nesta classe introduzimos os parâmetros necessários à representação de cores produzidas através de vetores. Deste modo, teremos os vetores *ambient*, *diffuse*, *emission* e *specular*, todos eles representados através da primitiva *Transformation*. O conteúdo de cada um é explicitado de seguida:

- ***ambient:*** após a aplicação do material, o modelo reflete a cor em questão, sendo a luz de incidência igual em todas as superfícies;
- ***diffuse:*** após a aplicação do material e da luz branca, é definida a cor primária do modelo;
- ***emission:*** após a aplicação do material, é emitida a própria luz deste;
- ***specular:*** após a aplicação do material, a superfície aparenta esta brilhante.

Tendo tudo isto em conta, definimos a função em seguida:

```
void Material :: draw() {

    if(diffuse[3] != -1)
        glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, diffuse);

    if(ambient[3] != -1)
        glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, ambient);

    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, specular);
    glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, emission);
}
```

3.4 *Camera*

Nesta fase, optamos por alterar a câmara previamente construída, capacitando-a agora de se mover livremente pelo cenário.

Tal como previamente, a direção da câmara é dada tendo em conta os ângulos *alpha* e *beta* que estão internamente definidos e associados à câmara. A principal diferença implementada nesta fase deve-se ao facto de que a câmara agora é passível de ser movida recorrendo às teclas *up*, *down*, *left* e *right*.

A tecla *up* move a câmera para a frente, somando o vetor direção ao vetor posição. A tecla *down* faz precisamente o contrário, recorrendo à subtração do vetor direção ao vetor posição.

Por forma a mover a câmera para o lado esquerdo, deve-se pressionar a tecla *left*, uma vez que, deste modo, irá ser calculado o vetor perpendicular aos vetores direção e orientação da câmera, por forma a que o produto dos dois seja também ele um vetor

perpendicular. A câmara irá movimentar-se ao longo deste, sendo necessário subtrair o valor deste vetor ao vetor posição, por forma a obter a deslocaç ão pretendida. O mesmo acontece com as movimentações à direita, causadas pelo pressionar da tecla *right*, recorrendo, no entanto, à soma do vetor e não à subtração.

```
case GLUT_KEY_UP: {
    posX += lookX * 1.7f;
    posY += lookY * 1.7f;
    posZ += lookZ * 1.7f;
    break;
}
case GLUT_KEY_DOWN: {
    posX -= lookX * 1.7f;
    posY -= lookY * 1.7f;
    posZ -= lookZ * 1.7f;
    break;
}
case GLUT_KEY_LEFT: {
    float up[3], dir[3], res[3];
    up[0] = up[2] = 0;
    up[1] = 1;
    dir[0] = lookX;
    dir[1] = lookY;
    dir[2] = lookZ;

    cross(dir, up, res);

    posX -= res[0] * 1.7f;
    posY -= res[1] * 1.7f;
    posZ -= res[2] * 1.7f;
    break;
}
case GLUT_KEY_RIGHT: {
    float up[3], dir[3], res[3];
    up[0] = up[2] = 0;
    up[1] = 1;
    dir[0] = lookX;
    dir[1] = lookY;
    dir[2] = lookZ;

    cross(dir,up,res);

    posX += res[0] * 1.7f;
    posY += res[1] * 1.7f;
    posZ += res[2] * 1.7f;
    break;
}
```

3.5 *Shape*

Por forma a otimizar o desempenho do programa, foram criados novos VBO's, capazes de suportar as coordenadas das normais e das texturas, agora representadas nos ficheiros .3d. Como tal, foi necessário alterar as funções *prepareBuffer* e *draw*:

```
void Shape :: prepareBuffer(vector<Point*> vert, vector<Point*>
    normal, vector<float> text){
    int i = 0;
    float* vertexs = new float[vert.size() * 3];

    for(vector<Point*>::const_iterator vertex_it = vert.begin();
    vertex_it != vert.end(); ++vertex_it){
        vertexs[i++] = (*vertex_it)->getX();
        vertexs[i++] = (*vertex_it)->getY();
        vertexs[i++] = (*vertex_it)->getZ();
    }

    i = 0;
    float* normals = new float[normal.size() * 3];

    for(vector<Point*>::const_iterator vertex_it = normal.begin();
    vertex_it != normal.end(); ++vertex_it){
        normals[i++] = (*vertex_it)->getX();
        normals[i++] = (*vertex_it)->getY();
        normals[i++] = (*vertex_it)->getZ();
    }

    glGenBuffers(3, buffer);
    glBindBuffer(GL_ARRAY_BUFFER, buffer[0]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(float) * numVert[0] * 3,
    vertexs, GL_STATIC_DRAW);

    glBindBuffer(GL_ARRAY_BUFFER, buffer[1]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(float) * numVert[1] * 3,
    normals, GL_STATIC_DRAW);

    glBindBuffer(GL_ARRAY_BUFFER, buffer[2]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(float) * numVert[2], &(text
    [0]), GL_STATIC_DRAW);

    free(vertexs);
    free(normals);
}
```

```
void Shape :: draw(){

    materials -> draw();

    glBindBuffer(GL_ARRAY_BUFFER, buffer[0]);
    glVertexPointer(3, GL_FLOAT, 0, 0);

    if(numVert[1] > 0) {
        glBindBuffer(GL_ARRAY_BUFFER, buffer[1]);
        glNormalPointer(GL_FLOAT, 0, 0);
    }

    if(numVert[2] > 0) {
        glBindBuffer(GL_ARRAY_BUFFER, buffer[2]);
        glTexCoordPointer(2, GL_FLOAT, 0, 0);
        glBindTexture(GL_TEXTURE_2D, texture);
    }

    glDrawArrays(GL_TRIANGLES, 0, (numVert[0]) * 3);
    glBindTexture(GL_TEXTURE_2D, 0);
}
```

Desta forma, torna-se possível carregar uma textura associada ao modelo, recorrendo para isso à função *loadTexture*, que irá guardar em memória os dados.

3.6 *Scene*

Esta classe foi criada com o intuíto de facilitar o desenho do cenário, contendo para isso todas as luzes existentes bem como o grupo principal do sistema.

Definimos aqui também uma função que irá aplicar as luzes, recorrendo à função *apply* definida na classe *Light*. Deste modo, no *engine* será apenas necessário invocar a função *lightApplication*, em seguida à aplicação das propriedades da câmara.

```
void Scene :: lightApplication(){
    GLenum number = 0;
    for(Light *l : lights)
        l -> apply(number++);
}
```

De realçar que, no entanto, o desenho do resto do cenário mantém-se igual.

4 Demonstração

Por forma a gerar os resultados pretendidos, torna-se necessário executar os seguintes comandos, a partir da pasta principal:

- **Gerador**

```
mkdir build && cd build
cmake ..
make
./generator torus 0.5 3 20 20 torus.3d
./generator sphere 3 20 20 sphere.3d
./generator -patch teapot.patch 10 teapot.3d
```

Figura 10: Comandos para gerar os ficheiros .3d necessários

O projeto inclui uma pasta designada por *Files3d* onde serão inseridos todos os ficheiros .3d criados, para posteriormente serem lidos pelo *engine*.

- ***Engine***

```
mkdir build && cd build
cmake ..
make
./engine solarsystem.xml
```

Figura 11: Comandos para gerar o Sistema Solar

Passando como parâmetro o nome do ficheiro XML que se pretende ler (este deverá estar guardado na pasta *XML's* incluída no projeto), o *engine* irá ler esse ficheiro, de modo a gerar as figuras pretendidas.

4.1 Menus

Mantivemos o menu criado nas fases anteriores, alterando tendo em conta as novas capacidades da câmara. Este menu aparece quando o utilizador, ao executar o *engine*, passa como argumento *-h* em vez do nome de um ficheiro XML.

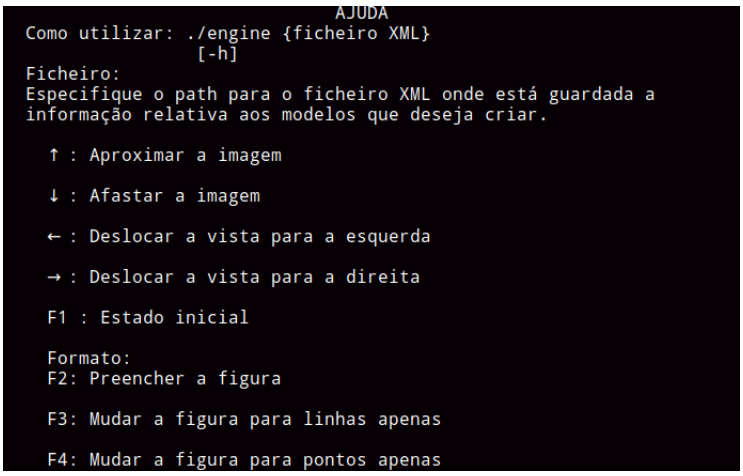

Figura 12: Menu Principal

Também se manteve o menu próprio do Sistema Solar, continuando este a aparecer quando pressionamos o botão direito do rato, durante a observação do sistema. Os parâmetros de "Movimento"foram criados de forma *responsive*.

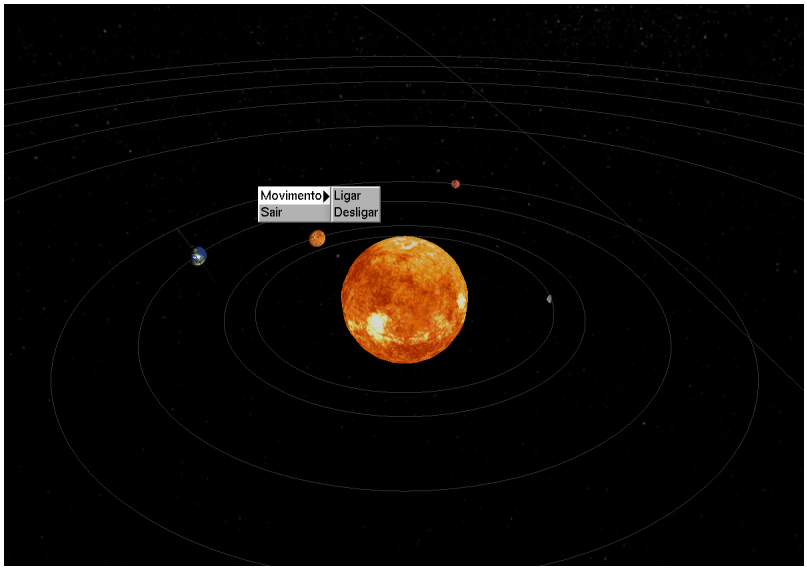

Figura 13: Menu dos Planetas

4.2 Sistema Solar

Tendo em vista a construção do Sistema Solar, tivemos em conta os planetas principais que o constituem (Mercúrio, Vénus, Terra, Marte, Júpiter, Saturno, Urano e Neptuno), a sua disposição, forma e cor. Optamos por não fazer o Sistema Solar à escala, isto é, não respeitamos totalmente as dimensões dos astros nem as distâncias que os separam, uma vez que, caso o fizessemos, ficaríamos com um cenário muito disperso e de difícil observação, acreditando nós não ser este o objetivo do projeto.

No entanto, tivemos em consideração a adição de alguns extras, como por exemplo alguns satélites naturais (a Lua e outros satélites de Júpiter e Saturno) bem como os anéis de Saturno e Urano (estes criados a partir do *torus* gerado). Acrescentamos também o planeta anão Plutão e um comenta, construído a partir do *teapot* processado do ficheiro *patch* fornecido.

Tendo tudo isto em vista, formulamos um ficheiro XML que permita a construção do Sistema Solar, tendo em atenção tudo o que fora referido previamente. Deste modo, podemos obter o resultado seguidamente apresentado.

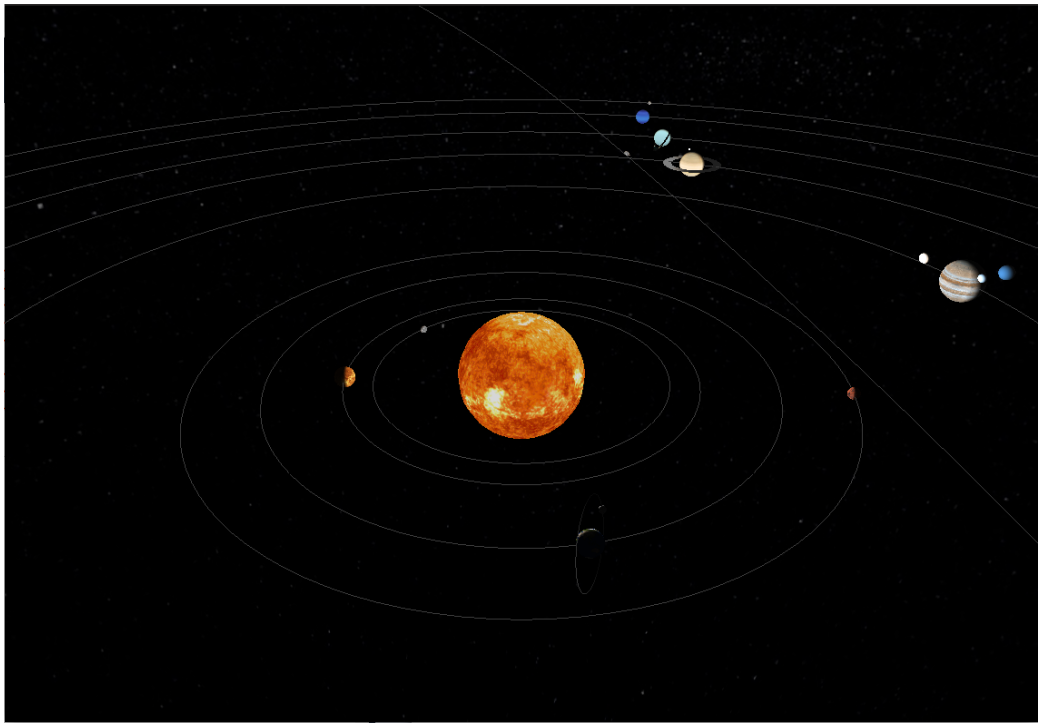

Figura 14: Sistema Solar

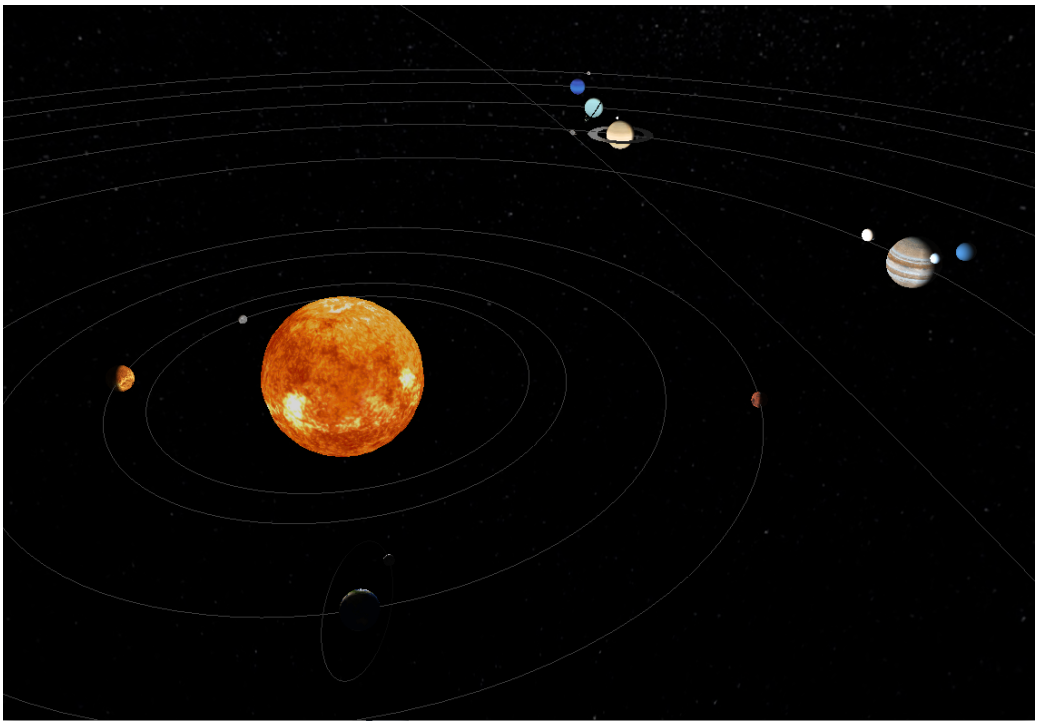

Figura 15: Sistema Solar ligeiramente descentralizado

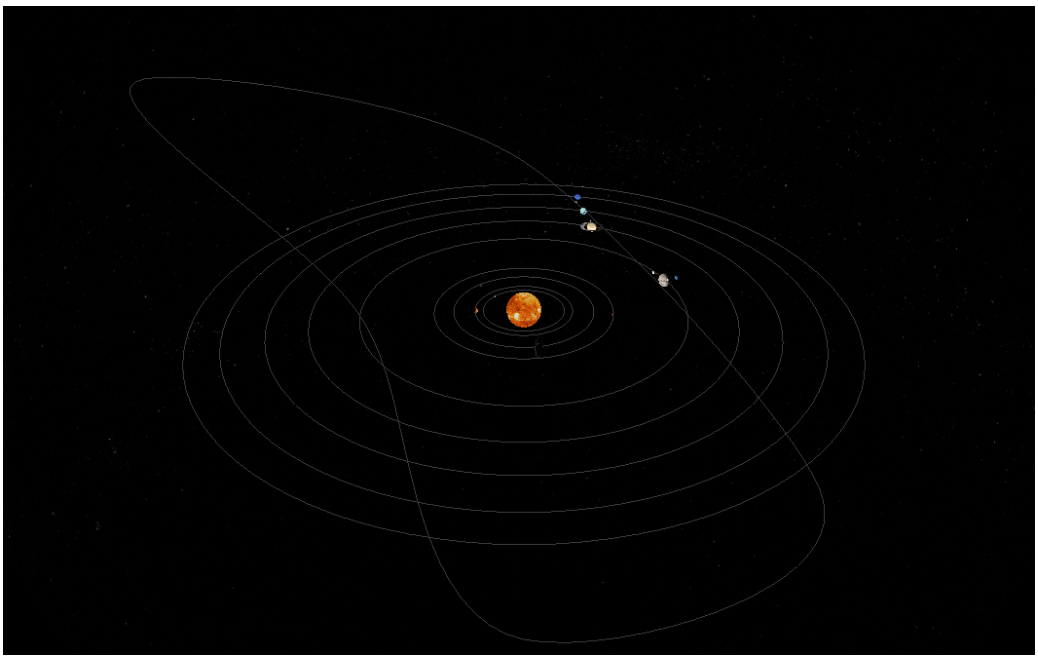

Figura 16: Sistema Solar visto à distância

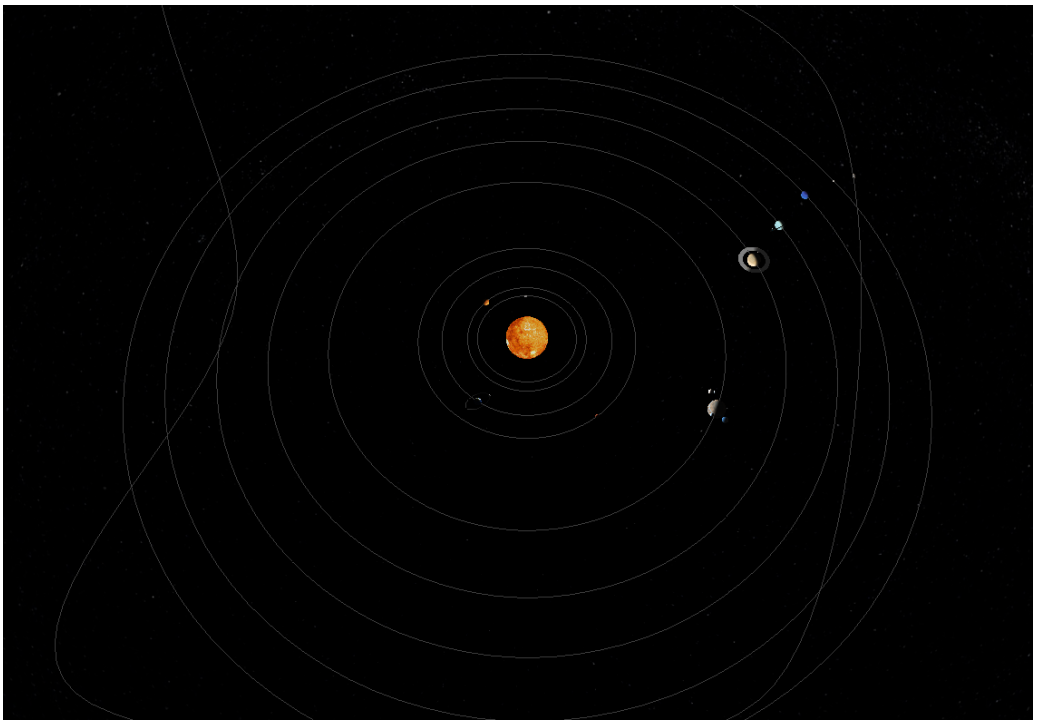

Figura 17: Sistema Solar visto de cima

De referir que, para além do apresentado nestas imagens, o nosso projeto é capaz de processar qualquer uma das primitivas geométricas definidas nas fases anteriores, aplicando-lhes texturas e iluminação, caso seja requisitado.

5 Conclusão

Para esta quarta e última fase podemos afirmar que cumprimos com os objetivos estabelecidos, uma vez que adicionamos as funcionalidades pretendidas, isto é, o uso de técnicas de iluminação e a aplicação de texturas aos diferentes objetos. Para além disto, também atualizamos o ficheiro XML responsável por suportar as informações do Sistema Solar, acrescentando as novas transformações agora processáveis, bem como o *parser* que torna capaz o processamento do mesmo.