Computação Gráfica

Fase 4 - Normais e Coordenadas de Texturas

Grupo 10

Carlos Pereira (A61887)
João Barreira (A73831)
Rafael Costa (A61799)

Maio 2017

Índice

1 Introdução

Nesta quarta e última fase, o *Generator* passou a gerar normais e coordenadas de texturas para cada um dos pontos gerados. Além disso, a *Engine* passou a suportar funcionalidades relativas à iluminação da cena (utilizando as normais geradas anteriormente) e à aplicação de texturas aos modelos.

Como complemento, foi também implementada uma câmara em terceira pessoa e aplicação do *Frustrum Culling* para a visualização da cena.

A câmara em terceira pessoa utiliza um modelo prédefinido de uma nave, sendo também possível utilizar um outro qualquer modelo cujo ficheiro *XML* terá de ser indicado à *Engine*.

Com a aplicação do *Frustrum Culling*, a visualização da cena e a execução da *Engine* tornou-se muito menos dispendiosa em termos gráficos e computacionais, visto que apenas são desenhados os modelos visíveis ao utilizador (que controla a câmara em terceira pessoa).

Neste relatório descrevem-se com detalhe cada uma das componentes desta fase. Além disso, é também feito um resumo das componentes do programa (*Generator* e *Engine*), sendo feita uma descrição pormenorizada de todas as funcionalidades finais do projeto.

2 *Generator*

Até à fase anterior, o programa *Generator* gerava as coordenadas dos pontos constituintes das primitivas, como também os índices para serem utilizados os *VBOs*. Nesta fase, o *Generator* foi alterado de maneira a que, para além de gerar os pontos e os índices, gerar também as coordenadas das normais e as coordenadas da textura de cada ponto. Ou seja, para cada primitiva (*plane*, *box*, *sphere*, *cylinder* e *teapot*) são geradas as coordenadas dos pontos que a constituem, como também as coordenadas da normal e textura de cada um desses pontos e, por fim, os índices a indicar a ordem pela qual os pontos deverão ser desenhados. Cada um destes conjuntos é escrito num ficheiro (tal como nas fases anteriores, o nome é introduzido pelo utilizador), onde são separados por uma linha em branco.

As normais dos pontos são geradas para que, ao introduzir-se uma fonte de luz, o *OpenGL* possa desenhar a sombra da primitiva. A normal de um ponto é calculada por face. Contudo, existem pontos que pertencem a mais que uma face ao mesmo tempo (por exemplo, os vértices de uma *box*, entre outros). Nesse caso, o *Generator* considera serem pontos que, apesar de terem as mesmas coordenadas, terão normais diferentes. Assim, as coordenadas destes pontos serão escritas mais que uma vez no ficheiro de *output*. Aliás, o número de vezes que as coordenadas de um ponto aparece no ficheiro resultante é igual ao número de faces que esse ponto pertence ao mesmo tempo.

As coordenadas de textura servem para que se possa atribuir uma textura a uma primitiva. Cada ponto terá um par de coordenadas x e y, que variam entre 0 e 1. Estas coordenadas são utilizadas pelo *OpenGL* para saber que ponto na imagem a ser utilizada como textura corresponde ao ponto da primitiva em questão. Na imagem, a origem das coordenadas x e y encontra-se no canto inferior esquerdo, onde têm o valor *(0, 0)*. À medida que percorrermos a imagem na horizontal ou na vertical, o valor das coordenadas em x e em y aumenta, respetivamente, até que se chegue a 1, que corresponde à extremidade da imagem. Pode-se verificar isto na **Figura 1**.

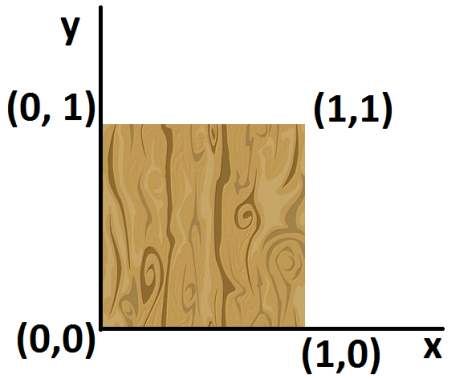

Figura 1 - Modelo de Coordenadas de uma Textura

2.1 Cálculo das Normais

Cada uma das primitivas gráficas passou a ter também um vetor *normals* que possui o conjunto das normais dos seus vértices. Assim sendo, o cálculo das normais para cada uma das primitivas gráficas foi inserido no contexto da função de geração dos seus vértices (*generateVertices*).

2.1.1 *Plane*

Neste caso, como se trata de um plano *XZ*, as normais dos seus vértices são sempre as mesmas e correspondem a (0, 1, 0) para os pontos da face voltada para cima, e (0, -1, 0) para os pontos da face voltada para baixo, tal como descrito na **Figura 2**.

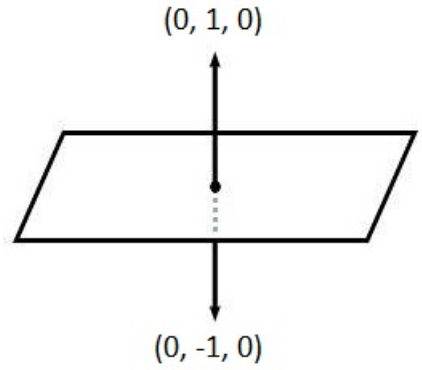

Figura 2 - Normais do *plane*

2.1.2 *Box*

A geração dos vértices da *Box* está dividida em três conjuntos de faces: faces paralelas ao plano *XY*, faces paralelas ao plano *XZ* e faces paralelas ao plano *YZ*.

Para as primeiras, os vértices da face da frente correspondem à normal (0, 0, 1) e os da face de trás a (0, 0, -1). Para as segundas, os vértices da face de cima têm como normal (0, 1, 0) e os da face de baixo (0, -1, 0). Para as terceiras, os vértices da face da direita possuem (1, 0, 0) como normal e os da face da esquerda (-1, 0, 0).

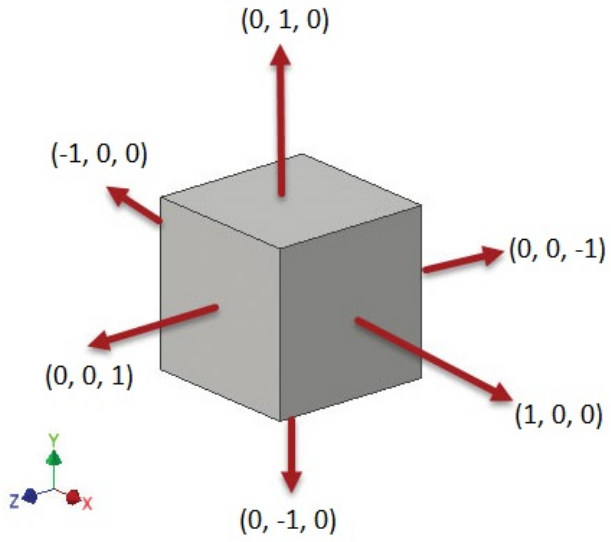

Figura 3 - Normais da *box*

2.1.3 *Cone*

Para os vértices correspondentes à base do cone, a normal correspondente é (0, -1, 0).

Já para os pontos pertencentes à superfície lateral do cone, a normal correspondente é dada por $(r * sin(\alpha), cos(arctg(h/R), r * cos(\alpha))$, em que R corresponde ao raio da base do cone, r ao raio da stack do ponto, h à altura do cone e α ao ângulo da subsecção do cone correspondente a uma slice (**Figura 4**).

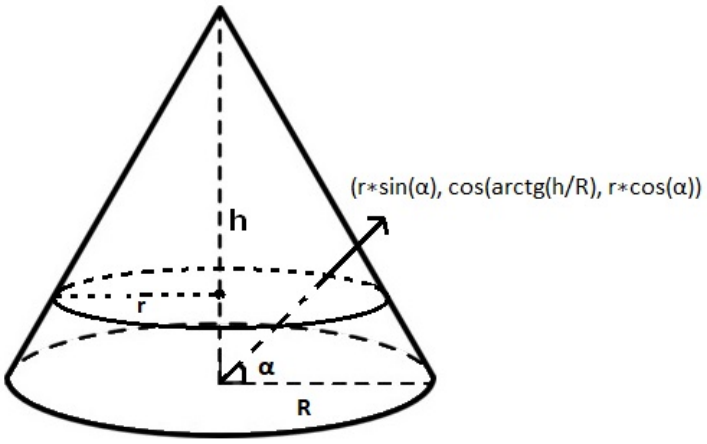

Figura 4 - Normais da *cone*

2.1.4 *Sphere*

Relativamente à esfera, a normal correspondente a um qualquer ponto da superfície da mesma é dada por $(sin(\beta) * cos(\alpha), cos(\beta), sin(\beta) * sin(\alpha))$, de acordo com a figura **Figura 5**.

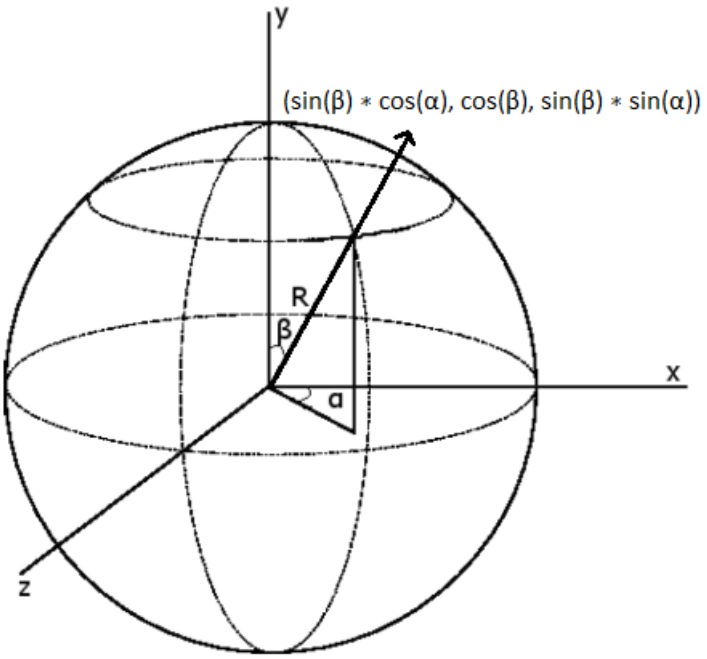

Figura 5 - Normais da *sphere*

2.1.5 *Bezier Patches*

Quando se trata de *Patches* de *Bezier* as normais são calculadas a partir do produto externo entre as derivadas em u e v normalizadas. As fórmulas dos cálculos das derivadas em u e v são as seguintes:

```
// Seja U = [u^3, u^2, u, 1] e V = [v^3, v^2, v, 1]

derivU = [3u^2, 2u, 1, 0] * M * Pa * M^T * V^T

derivV = U * M * Pa * M^T * [3v^2, 2v, 1, 0]
```

Em que M corresponde à matriz de *Bezier*, P aos dezasseis pontos de controlo e $a \in x, y, z$.

2.2 Aplicação das Texturas

Cada uma das primitivas gráficas passou a ter também um vetor *texCoords* responsável por guardar as coordenadas correspondentes às texturas que poderão ser posteriormente aplicadas a cada uma delas. Para isso, apenas foi necessário, para os pontos de cada primitiva, associar as coordenadas dos pontos correspondentes de uma textura.

2.2.1 *Plane*

No caso do plano, o canto superior direito corresponde ao ponto da textura de coordenenadas (1, 1), o canto superior esquerdo corresponde a (0, 1), o canto inferior esquerdo corresponde a (0, 0) e o canto inferior direito corresponde a (1, 0).

2.2.2 *Box*

Relativamente às faces paralelas ao plano XY, os pontos correspondentes da textura são dados por $(a/numDiv, b/numDiv)$, em que a corresponde ao número da divisão no eixo dos xx, b à divisão no eixo dos zz e $numDiv$ ao número total de divisões da face.

Em relação às faces paralelas ao plano XZ, os pontos correspondentes da textura são dados por $(b/numDiv, a/numDiv)$.

No caso das faces paralelas ao plano YZ, os pontos correspondentes da textura são dados por $(b/numDiv, a/numDiv)$.

2.2.3 *Cone*

Para o caso da base do cone, os pontos correspondentes da textura são dados por $(i/slices, 1)$ e $(i/slices, 0)$, sendo i o número da slice do ponto e *slices* o número total de secções verticais do cone.

Já para o caso dos pontos da superfíce lateral do cone, os pontos correspondentes da textura são dados por $((slices - j)/slices, (stacks - i)/stacks)$, em que i corresponde ao número da *slice* do ponto, j ao número da *stack* do ponto e *slices* e *stacks* ao número total de secções verticais e horizontais do cone, respetivamente.

2.2.4 *Sphere*

Relativamente à esfera, os pontos correspondentes da textura são dados por $((slices - j)/slices, (stacks - i)/stacks)$, em que i corresponde ao número da *slices* do ponto, j ao número da *stack* do ponto e *slices* e *stacks* ao número total de secções verticais e horizontais do cone, respetivamente.

2.2.5 *Bezier Patches*

As coordenadas de textura nas *Bezier Patches* segundo x e y, são dadas, repetivamente, por (1, v) e (1, u). Onde u é o valor da tesselagem em u e v é

o valor da tesselagem em v.

3 *Engine*

Relativamente à fase anterior, o programa *Engine* está agora preparado para desenhar as sombras e aplicar texturas às primitivas. Para isso, recorre às coordenadas geradas pelo programa *Generator*.

A sombra de uma primitiva é desenhada automaticamente a partir das suas normais. Contudo, para que se possa tirar proveito disso, será necessário adicionar pelo menos uma fonte de luz de forma a visualizarem-se as sombras. A *Engine* sabe qual é o tipo de luz a partir da *tag lights*. Cada um dos filhos desta terá uma outra *tag light*, sendo indicadas as coordenadas da fonte de luz, como também o tipo. Uma luz pode ser um de três tipos: *point, directional e spotlight*.

Quando uma luz é do tipo *point*, significa que os raios de luz são emitidos em todas as direções a partir de um único ponto. Caso seja *directional*, os raios de luz serão todos paralelos uns aos outros. Se for do tipo *spotlight*, os raios de luz serão emitidos a partir de um ponto, contudo, em vez de irradiarem em todas as direções como no *point*, estão restritos a uma forma tipo cone. Os diferentes tipos de luz podem ser verificados na Figura 6. Por exemplo, se quiséssemos adicionar uma luz de cada tipo no ficheiro *XML* teríamos algo do género:

```
<lights>
    <light posx=0 posy=0 posz=0 type="point"/>

    <light posx=1 posy=1 posz=1 type="directional"/>

    <light type="spot" posx=1 posy=1 posz=1 spotdirx=1
           spotdiry=0 spotdirz=0 cutoff=45 exponent=0/>
</lights>
```

Onde *posx*, *posy* e *posz* são, respetivamente, as coordenadas em x, y e z a partir de onde a luz será emitida, *spotdirx*, *spotdiry* e *spotdirz*, são, repetivamente, é a direção da luz segundo os eixos x, y e z, *cutoff* é o ângulo a partir do centro e *exponent* é a focagem da luz (quanto maior o valor, mais focada esta será).

Além destas propriedades, também é possível definir as cores das componentes ambiente, difusa e especular de luz. Para isso, acrescentam-se à model, os elementos *ambtr*, *ambtg* e *ambtb*, para a cor no formato *RGB* da luz ambiente, os elementos *diffr*, *diffg* e *diffb*, para a cor no formato *RGB* da luz difusa e os elementos *specr*, *specg* e *specb*, para a cor no formato *RGB* da luz especular.

```
<model file="primitive.3d" diffR=0.5 diffG=0.5 diffB=0.5>

<model file="primitive.3d" ambtR=0.5 ambtg=0.5 ambtb=0.5>

<model file="primitive.3d" specR=0.5 specG=0.5 specb=0.5>
```

Também é possível acrescentar uma luz emissiva à superfície de uma primitiva, através dos elementos *emisr, emisg* e *emisb*, na *tag model*, que corres-

pondem à cor da luz no formato *RGB*.

```
<model file="primitive.3d" emisr=1 emisg=1 emisb=1>
```

Acrescentando ainda às propriedades de luz da superfície da primitiva, é ainda possível atribuir um valor à sua *shininess*. Apenas tem que se acrescentar à *tag model*:

```
<model file="primitive.3d" shininess=64>
```

Com esta nova versão da *Engine*, é possível aplicar texturas a primitivas. Para isso, utilizam-se as coordenadas de textura geradas pelo *Generator*. Para carregar uma textura, apenas é preciso adicionar o elemento texture à *tag model*:

Para além das luzes e das texturas, a *Engine* aplica *Frustum Culling* no *rendering* da cena e implementa, também, uma cãmara em terceira pessoa. Estas duas funcionalidades, serão explicadas com mais detalhe na parte final desta secção.

```
<model file="primitive.3d" texture="texture.jpg"/>
```

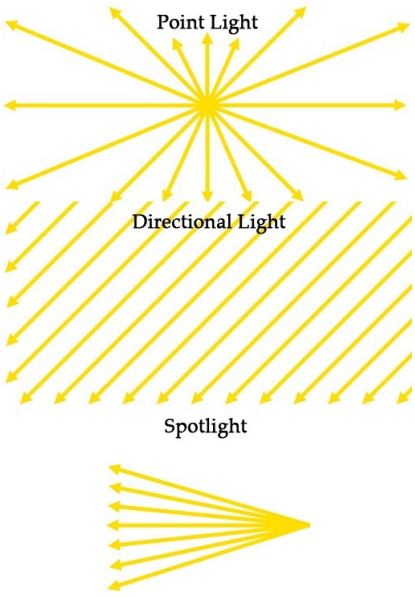

Figura 6 - Diferentes Tipos de Luz

3.1 Luzes

Através de todas as *tags* e atributos relativamente às luzes consegue-se, facilmente, representar um dos três tipos de luz em *OpenGL*.

Uma luz constitui uma operação como qualquer outra em *OpenGL*, sendo armazenada no vetor de operações em *OpenGL* e executada no ciclo de *rendering*. Uma luz representa uma classe abstrata (classe *light*) que herda o método *execute* da classe base *glOperation*. Por sua vez, os três diferentes tipos de luzes (*point*, *directional* e *spotlight*) representam classes derivadas da classe *light*, implementando o método *execute* de acordo com as suas características.

De maneira a permitir múltiplas luzes na mesma cena, mantém-se armazenado o valor da última luz utilizada (*light0, light1, ...*). Sempre que se pretende representar uma nova luz, a número da luz utilizada é incrementado (naturalmente se a última luz utilizada foi a *light7*, então a próxima luz utilizada será a luz *light0*). As diferentes luzes afetam os objetos pertencentes a um grupo e todos os objetos pertencentes a grupos de hierarquias inferiores. Assim, sempre que se executa a função *glPushMatrix* deve-se realizar o *enable* de todas as luzes e, por sua vez, desativar todas as luzes quando se executa a função *glPopMatrix*.

3.2 Aplicação das Texturas

Por predefinição, a *Engine* carrega os ficheiros *XML* relativos ao modelo a ser executado (Sistema Solar), ao modelo para a câmara (Nave) e as respetivas texturas de ambos. Além disso, utiliza uma textura por predefinição para o universo (*background*). Todos estes aspetos podem ser redefinidos, executando a *Engine* com os nomes dos modelos/textura como parâmetros.

Para o caso da textura do *background* (universo), é chamada a função *loadBackground* que carrega o ficheiro da textura e, de seguida, a função *drawBackground* desenha um plano com as dimensões da janela, aplicando a textura retirada da imagem pela função anterior.

Relativamente aos modelos, o ficheiro da textura é opcional e deve estar presente no ficheiro *XML* do modelo em questão (atributo *texture* da tag *model*). A *Engine* apenas gera os *buffers* relativos às *VBOs* uma vez (função *generate*), carregando a textura de forma similar à descrita acima. A posição 2 dos *buffers* das *VBOs* corresponde aos pontos da textura que será desenhada na função de execução (*execute*).

3.3 *Frustum Culling*

Para aumentar a eficiência da renderização de uma cena foi implementada uma técnica de *Frustum Culling*. Esta técnica consiste em não desenhar todos os objetos que não se encontrem dentro do campo visível. O campo visível corresponde a tudo o que se consegue visualizar através da posição da janela e pode ser definido à custa de seis planos (cima, baixo, esquerda, direita, *near* e *far*). Qualquer objeto que se encontre dentro, ou parcialmente dentro do campo é desenhado, tal como exemplificado na **Figura 7**.

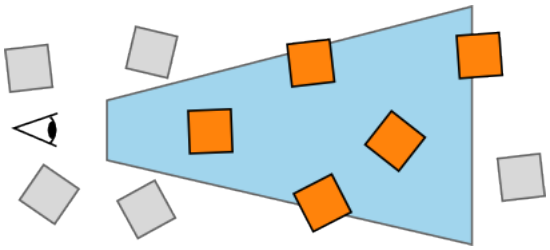

Figura 7 - *Frustum Culling* (apenas os objetos a laranja são desenhados)

Torna-se, portanto, necessário detetar se um objeto está dentro ou fora do campo visível. O primeiro passo deste processo consiste na definição dos seis planos. Estes variam de acordo com a perspectiva da janela e com as definições da câmara. Sempre que se invoca a função *gluPerspective* deve ser igualmente invocada a função *setPerspective* com os mesmos argumentos. Esta função é responsável por armazenar os valores da perspectiva e calcular as dimensões (altura e largura) dos planos *near* e *far*. O calculo destas dimensões é dado pelas seguintes expressões:

```
setPerspective(angle, ratio, near, far)
{

    // RAD = valor de um grau em radianos

    tang = tangente(RAD * angle * 0.5)
    alturaNear = near * tang
    larguraNear = alturaNear * ratio
    alturaFar = far * tang
    larguraFar = alturaFar * ratio
}
```

Para a definição dos seis planos existe uma função chamada *setCam*. Esta função deve ser invocada sempre que é chamada a função *gluLookAt* com exatamente os mesmos parâmetros desta. A função recebe, portanto, três vetores (posição da câmara, posição para onde a câmara está a apontar e o vetor "*up*"). Através destes parâmetros e, juntamente com as dimensões dos planos *near* e *far*, é possível definir os seis planos do campo visível. Um plano pode ser definido à custa de um ponto e de uma normal. As normais dos seis planos são definidas de modo a que apontem para dentro do campo visível. O primeiro passo consiste no cálculo das normais em x, y e z da câmara. Estes vetores são calculados a partir das seguintes expressões:

```
Z = look - pos
normalizar(Z)

X = up * Z
normalizar(X)

Y = Z * X
```

É importante referir que a normal Y corresponde ao verdadeiro vetor "*up*" da câmara. O vetor Z aponta na direção contrária ao vetor *look*. Através deste vetor é possível calcular os pontos dos planos *near* e *far*. Para além disso, este vetor corresponde à normal do plano *far*, logo o inverso deste vetor corresponde à normal do plano *near*.

```
pontoNear = pos - (Z * (posicao near em z))
pontoFar = pos - (Z * (posicao far em z))

planoNear = (Z * (-1), pontoNear)
planoFar = (Z, pontoFar)
```

Finalmente, os pontos e as normais dos restantes planos são calculados através dos vetores X e Y. São também necessários os pontos e as dimensões dos planos *near* e *far*.

```
aux = pontoNear + (Y * alturaNear) - pos
normalizar(aux)
normal = aux * X
planoTopo = (normal, pontoNear + (Y * alturaNear)

aux = pontoNear - (Y * alturaNear) - pos
normalizar(aux)
normal = X * aux
planoBaixo = (normal, pontoNear - (Y * alturaNear)

aux = pontoNear - (X * larguraNear) - pos
normalizar(aux)
normal = aux * Y
planoEsquerda = (normal, pontoNear - (X * larguraNear)

aux = pontoNear - (X * larguraNear) - pos
normalizar(aux)
```

```
normal = Y * aux
planoDireita = (normal, pontoNear - (X * larguraNear)
```

O resultado de todas as expressões listadas anteriormente é ilustrado na seguinte figura:

Os cálculos anteriores permitem a definição dos seis planos, sendo então possível testar se um determinado ponto pertence ou não ao campo visível. Um ponto pertence ao campo visível se o sinal da sua distância a um plano for maior do que zero. Para um ponto pertencer ao campo visível, a sua distância aos seis planos deve ser positiva. No entanto, se a distância a um dos planos for negativa isto significa que o ponto situa-se do lado contrário à normal do plano e, portanto, fora do campo visível. O seguinte algoritmo traduz esta verificação:

```
booleano res = verdadeiro

Para i = 0 ate (i < 6 e res == verdadeiro)
fazer i = i + 1 {
    Se distancia(plano[i], ponto) < 0 fazer
        res = falso
}

return res
```

O algoritmo anterior permite com sucesso verificar se um certo ponto está dentro ou fora do campo visível. No entanto, em cenas que possuam bastantes objetos e que estes, por sua vez, possuam um elevado número de vértices, este algoritmo torna-se bastante dispendioso. Para se resolver este problema um objeto é contido numa esfera que o cobre completamente. Considera-se que cada objeto é desenhado na origem, ou seja, que o seu centro corresponde às coordenadas *(0, 0, 0)*. O raio da esfera de um objeto corresponde, portanto, ao valor absoluto da sua maior coordenada (em x, y ou z). A classe *xmlParser* encarrega-se deste cálculo após a leitura de um conjunto de vértices. Começa por percorrer todos os vértices lidos, calculando os máximos das três coordenadas (x, y e z) em valores absolutos. Estes valores são guardados num mapa associado ao nome do ficheiro correspondente ao objeto. Assim, este cálculo é efetuado apenas uma só vez por ficheiro. No fim, na armazenagem de todos os valores para o desenho de um objeto, é selecionada a coordenada de valor superior, depois desta ser submetida a todas as eventuais escalas que possam ter ocorrido. O algoritmo para testar se uma esfera está contida dentro do campo visível é muito semelhante ao teste de um ponto. Se a distância do seu centro a qualquer um dos planos for inferior ao valor negativo do seu raio então a esfera não está contida dentro do campo visível.

```
booleano res = verdadeiro
real distancia = 0

Para i = 0 ate (i < 6 e res == verdadeiro)
fazer i = i + 1 {
    distancia = distancia(plano[i], centro)

    Se distancia < -raio fazer
        res = falso
}

return res
```

Todos os passos descritos anteriormente permitem que todos os objetos que estejam fora do campo de visão não sejam desenhados independentemente da movimentação da câmara. No entando este método falha caso os objetos sofram transformações geométricas como translações ou rotações. Isto porque ao se efetuar uma translação a origem de um objeto muda relativamente ao campo de visão. É então necessário manterem-se guardados os valores das translações e rotações para posteriormente as aplicar ao centro de um objeto (origem). Para isso simulou-se a matriz *MODEL* presente em *OpenGL*. Esta matriz não foi utilizada pois esta é modificada em conjunto com a matriz *VIEW* não sendo o caso prentendido. Como apenas interessa armazenar os valores das translações e rotações, foram utilizadas duas estruturas de dados: um vetor de três posições para as translações e uma matriz (3 x 3) para armazenar as rotações. Estas estruturas são por sua vez armazenadas em duas *stacks*. Sempre que ocorre uma operação de *glPushMatrix*, devem ser guardadas no topo das *stacks* os valores das translações e das rotações que tenham ocorrido anteriormente. Quando ocorre uma *glPopMatrix* significa que se desce na hierarquia e portanto as translações e rotações efetuadas no grupo atual são removidas das *stacks*. Estas operações são descritas através dos seguintes algoritmos:

```
// Operacao de adicao //

// Nao houve translacao entao adiciona-se a origem
Se stackTranslacoes == vazia fazer
    adicionar(centro)

// Caso contrario o novo grupo herda as translacoes
// efetuadas nas hierarquias superiores
Senao fazer
    adicionar(topoTranslacoes)
```

```
// O processo para as rotacoes e o mesmo
Se stackRotacoes == vazia fazer
    adicionar(matriz identidade)
Senao fazer
    adicionar(topoRotacoes)


// Operacao de remocao //

Se stackTranslacoes != vazia fazer
    removerTopo(stackTranslacoes)

Se stackRotacoes != vazia fazer
    removerTopo(stackRotacoes)
```

Relativamente às rotações, existem três casos possíveis: rotação no eixo dos *xx*, rotação no eixo dos *yy* e rotação no eixo dos *zz*. As seguintes matrizes ilustram, respetivamente, estes três casos:

$$MX = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) \\ 0 & sin(\theta) & cos(\theta) \end{bmatrix}$$

$$MY = \begin{bmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{bmatrix}$$

$$MZ = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Sempre que ocorre a função *glRotate* deve ser chamada a função *rotateCoords* para atualizar a *stack* com a nova matriz de rotação. O pseudocódigo desta função é o seguinte:

```
rotateCoords(axisX, axisY, axisZ, angulo)
{
    // Converter o angulo para graus
    angulo = angulo * PI / 180

    Se axisX != 0 fazer
        matrizRot = getMX(axisX, angulo)
```

```
    Senao Se axisY != 0 fazer
        matrizRot = getMY(axisY, angulo)
    Senao Se axisZ != 0 fazer
        matrizRot = getMZ(axisZ, angulo)

    removerTopo(stackRotacoes)
    adicionar(matrizRot)
}
```

Finalmente, a função *translateCoords* trata dos cálculos necessários para a atualização da origem de acordo com todas as translações que tenham ocorrido. Esta função deve ser invocada sempre que se chamar a função *glTranslate* e o seu pseudocódigo é o seguinte:

```
translateCoords(x, y, z)
{
    vetorAux = { x, y, z }

    res = multMatrixVector(topoStackRotacoes, vetorAux)

    // Soma-se a nova transla  o    transla  o mais
    // antiga na hierarquia
    res = res + topoStackTranslacoes

    removerTopo(stackTranslacoes)
    adicionar(res)
}
```

Para uma melhor compreensão das funções anteriores, considere-se a seguinte hierarquia:

```
<scene>
    <group>
        <translate x=5/>
        <models>
            <model file="sphere.3d"/>
        <models>
    </group>
    <group>
        <rotate axisY=1 angle=90/>
        <translate z=-10/>
        <models>
```

```
            <model file="box.3d"/>
        </models>
    </group>
</scene>
```

A cena descrita é composta por dois grupos (ou seja, duas chamadas das funções *glPushMatrix* e *glPopMatrix*), onde cada grupo possui apenas um modelo. Consideremos o primeiro grupo que consiste no desenho de uma esfera transladada no exio dos *xx*. Após a leitura da *tag group*, é adicionado um vetor com três posições correspondentes à origem (ou seja, *(0, 0, 0)*), pois ainda não houve qualquer translação. Depois de se processar a *tag translate*, a função *translateCoords* é invocada com os seguintes argumentos: *(x = 5, y = 0, z = 0)*. Como não houve qualquer rotação nesse grupo, o resultado da multiplicação da matriz de rotação por este vetor será o próprio vetor (já que a matriz de rotação corresponde à identidade). Ou seja, a origem do modelo da esfera corresponde às coordenadas *(5, 0, 0)*.

No grupo seguinte são, novamente, adicionados o vetor *(0, 0, 0)* e a matriz identidade às *stacks*, já que este grupo é independente do anterior. Após a leitura da *tag rotate*, é carregada a matriz correspondente à rotação no eixo dos *yy* com o respetivo ângulo. Como este grupo possui uma rotação antes de uma translação, o resultado da sua origem passa a depender destas duas componentes e corresponde ao resultado da multiplicação da matriz de rotação pelo vetor *(0, 0, -10)*.

Através de todos estes passos, a classe *TrianglesDrawing* passa a poder determinar se deve desenhar um modelo ou não. O teste é o seguinte:

```
origem = topoStackTranslacoes

// O raio da esfera que cobre todo o modelo ja foi
// previamente calculado como foi dito anteriormente
Se dentroCampoVisao(origem, raio) == verdadeiro fazer
    desenhar()
```

3.4 Câmara em Terceira Pessoa

Uma nova funcionalidade acrescentada ao programa *Engine* consiste na interação com câmara em terceira pessoa. A câmara em terceira pessoa foi implementada com recurso à câmara em primeira pessoa, alterando os valores dos vetores *pos* e *look*.

A câmara em primeira pessoa possui duas funcionalidades: rotação da câmara e movimentação. A primeira diz respeito à interação com o rato e *ARROW KEYS*, podendo rodar no eixo dos *xx, yy* ou *zz*. Conforme a rotação pretendida, incrementa-se ou decrementa-se o ângulo de rotação em cinco graus. As fórmulas de rotação da câmara em cada um dos três eixos são as seguinte:

```
Se rodarEsquerda fazer
    camAngle = camAngle + 5
Senao Se rodarDireita fazer
    camAngle = camAngle - 5
Senao Se rodarCima fazer
    camAngleY = camAngleY - 5

    Se camAngleY < -90 fazer
        camAngleY = -90
Senao Se rodarBaixo fazer
    camAngleY = camAngleY + 5

    Se camAngleY > 90 fazer
        camAngleY = 90


lx = px + sin(camAngle)
ly = py + sin(camAngleY)
lz = pz + cos(camAngle)
```

Como se pode verificar, o ângulo da câmara em y é limitado entre -90º e 90º para uma melhor visualização da cena.

Enquanto que a componente de rotação da câmara apenas altera o vetor *look*, a componente de translação da câmara altera os vetores *look* e *pos*. As expressões utilizadas para a manipulação destes dois vetores são as seguinte:

```
dx = lx - px
dy = ly - py
dz = lz - pz
rx = -dz
rz = dx

//Deslocacao para a frente
px = px + k * dx
lx = lx + k * dx

py = py + k * dy
ly = ly + k * dy

pz = pz + k * dz
lz = lz + k * dz

//Deslocacao para tras
```

```
px = px - k * dx
lx = lx - k * dx

py = py - k * dy
ly = ly - k * dy

pz = pz - k * dz
lz = lz - k * dz

//Deslocacao para a esquerda
px = px - k * rx
lx = lx - k * rx

pz = pz - k * rz
lz = lz - k * rz

//Deslocacao para a direita
px = px + k * rx
lx = lx + k * rx

pz = pz + k * rz
lz = lz + k * rz
```

Como se pode verificar, as expressões anteriores podem ser reduzidas à seguinte expressão::

```
pa = pa + k * da
la = la + k * da
```

Onde k corresponde à velocidade do deslocamento (negativo ou positivo), d ao valor do deslocamento e $a \in \{x, y, z\}$.

A câmara em terceira pessoa também utiliza as mesmas componentes de rotação e translação da câmara em primeira pessoa. A diferença consiste no desenho da entidade que corresponde à terceira pessoa. Essa entidade deve estar sempre posicionada em frente à câmara, independentemente da movimentação desta. Isto é facilmente conseguido em *OpenGL*, desenhando uma qualquer primitiva correspondente à entidade da câmara antes de se invocar a função *gluLookAt*. Desta maneira a entidade é posicionada à frente da câmara, podendo ser sujeita a qualquer tipo de transformação geométrica.

De modo a aumentar a interatividade com a câmara, esta apenas acompanha o movimento da entidade quando ocorre uma determinada "força". Como exemplo, considere-se uma movimentação da câmara para a frente, onde a origem do desenho da entidade corresponde às coordenadas *(0, 0, -3)*. A entidade

é desenhada em relação à câmara e as coordenadas da câmara correspondem à verdadeira origem *(0, 0, 0)*. Ou seja, a entidade é desenhada inicialmente três unidades à frente da câmara (as coordenadas segundo z são trocadas em *OpenGL*). Uma movimentação para a frente faz com que a entidade se afaste da câmara, ou seja, sofra uma transladação negativa no eixo dos *zz*. Por sua vez, foi estipulado que a câmara apenas acompanha o movimento da entidade quando se atinge o valor de uma determinada força. Admite-se assim que esta força é igual ao deslocamento da entidade, então temos:

```
//Movimentacao para a frente
fz = fz - dz

Se fz < -1 fazer
    fz = -1
    moverCameraFrente()
```

Em que *fz* representa a força no eixo dos *zz* e *dz* o deslocamento no eixo dos *zz*. Neste caso, a entidade move-se desde a sua posição inicial até à posição *(0, 0, -4)*, sem que a câmara a siga. A partir dessa posição, a câmara acompanha o movimento da entidade, impedindo que esta se afaste mais da câmara. Como a entidade foi transladada em relação à câmara, a câmara deve mover-se de maneira a que a entidade volte ao estado inicial. Para isso foi definida uma fatia de tempo *dTime*. É atribuído o valor zero sempre que se pressione uma tecla de movimentação da câmara (*w, a, s* ou *d*). Sempre que a função *renderScene* for invocada, efetua-se uma medição de tempo e soma-se ao tempo decorrido. Se o tempo decorrido for superior à fatia de tempo, então procede-se à estabilização da câmara. A estabilização da câmara consiste no movimento da câmara na direção pretendida e na diminuição da força (e, por consequência, do deslocamento) da entidade. Para o exemplo anterior, teríamos uma movimentação da câmara para a frente e o deslocamento da entidade desde a posição *(0, 0, -4)* até à posição *(0, 0, -3)*, que corresponde à posição inicial da entidade. Este deslocamento da entidade é necessário, pois não basta mover a câmara para se alterar a posição do desenho da entidade. Se uma entidade está transladada quatro unidades em relação à câmara, ela vai ser sempre desenhada nessa posição, independentemente da movimentação da câmara. Estes processos são análogos para todas as movimentações da câmara.

3.5 Interação com o programa *Engine*

O programa *Engine* pode ser executado com argumentos. O primeiro argumento diz respeito ao ficheiro em formato *XML* que possui a hierarquia de uma cena. O segundo argumento corresponde à entidade da câmara (este ficheiro também deve estar num formato válido em *XML*). O terceiro argumento diz respeito a uma imagem que será carregada em *background*. O programa

pode também ser executado sem argumentos. Neste caso, o programa carrega os ficheiros por defeito (*solarSystem.xml*, *tiefighter.xml* e *univ.jpg*).

A pasta *demos* possui inúmeros ficheiros em formato *XML* que representam o desenho de primitivas individuais (plano, cone, esfera, ...), testes de formatos inválidos em *XML* e cenas complexas. Entre as cenas complexas destacam-se os modelos do sistema solar com e sem texturas (*solarSystem.xml* e *solarSystem2.xml*, respectivamente) e o modelo de um farol. O modelo do farol permite exemplificar o funcionamento da luz do tipo *spotlight* e está ilustrado na **Figura 8**.

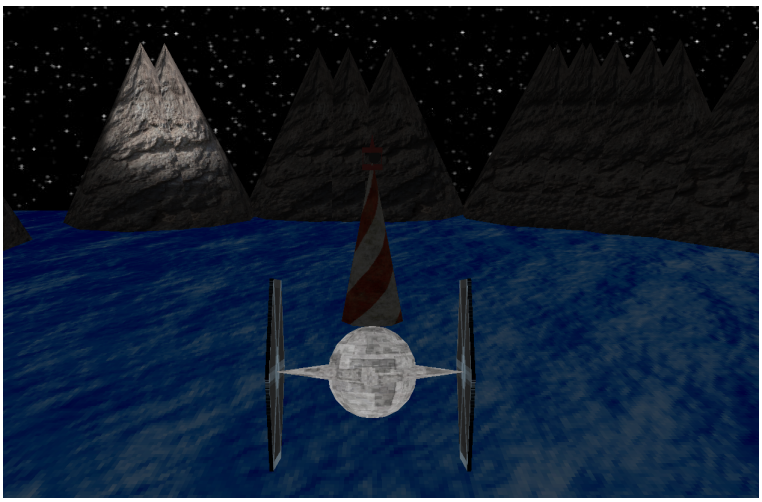

Figura 8 - Modelo do Farol

Para além destes componentes, a pasta *demos* também possui várias imagens que podem ser utilizadas como texturas de primitivas e vários ficheiros com a extensão "*.3d*" que possuem os dados necessários para o desenho de uma primitiva.

Finalmente, é possível interagir com a câmara em terceira pessoa através do teclado e do rato. Ao mover-se o rato após se pressionar o botão do lado esquerdo deste efetua-se uma rotação da câmara. Já a interação com o teclado é seguinte:

- *UPARROW* - Roda a câmara para baixo.
- *DOWNARROW* - Roda a câmara para cima.
- *LEFTARROW* - Roda a câmara para a esquerda.
- *RIGHTARROW* - Roda a câmara para a direita.
- *w* - Move a câmara para a frente.

- *a* - Move a câmara para a esquerda.
- *s* - Move a câmara para trás.
- *d* - Move a câmara para a direita.

4 Conclusão

O nosso programa começou por apenas ser capaz de, através do *Generator*, gerar os pontos de várias primitivas e, com a *Engine*, ler um ficheiro *XML* com a informação sobre os modelos (e respetivos ficheiros com os vértices gerados anteriormente), e desenhá-los em modo imediato.

Esta primeira fase foi importante para perceber a representação eficiente de modelos, utilizando para tal a regra da mão direita para desenhar a parte da frente dos mesmos.

Posteriormente, foram implementadas funcionalidades relativas à rotação, translação e escala de modelos, definidas por *tags* específicas do ficheiro *XML* da cena. Com o auxílio destas novas funcionalidades e com a implementação de novas noções hierárquicas em *XML* (e.g. *tag group*), foi possível criar uma cena relativa ao Sistema Solar, com planetas, Sol e luas.

Nesta fase percebemos a importância do polimorfismo do *C++*, da utilização de *containers* e das operações de *popMatrix* e *pushMatrix* para a definição de grupos hierárquicos complexos e a aplicação correta de transformações geométricas.

Na fase seguinte, as rotações e as translações foram melhoradas. No caso da translação, foi adicionada a capacidade de introduzir pontos parciais que definem uma curva do tipo *Catmull-Rom*, bem como o número de segundos para completar este mesmo percurso. No caso da rotação, passa a ser possível usar tempo em vez de um ângulo. Além disso, foi implementado o desenho de modelos a partir de *patches* de *Bezier* e a substituição do desenho em modo imediato das primitivas por *VBOs* que é um método bastante mais eficiente quando os índices são bem escolhidos.

Esta última fase consistiu na geração adicional das normais dos modelos e de coordenadas para as texturas. Assim, foi possível a implementação de vários tipos de luzes para a iluminação da cena e a aplicação de texturas aos modelos. Além disso, implementou-se uma câmara em terceira pessoa que permitiu uma navegação mais real pela cena gerada, bem como o método de *Frustrum Culling* que foi responsável por uma renderização da cena bem mais eficiente pois apenas são desenhados os modelos dentro do campo de visão do utilizador.