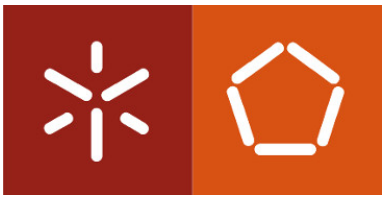

Computação Gráfica

Trabalho Prático - Fase 4

30 de maio de 2021

Patrícia Pereira, A89578

Meriem Khammasi, A85829

Jaime Oliveira, A89598

Luís Magalhães, A89528

Conteúdo

1 *Abstract*

Quarta fase do trabalho prático realizado no âmbito da Unidade Curricular de Computação Gráfica da Universidade do Minho. Esta fase consiste na adição das últimas funcionalidades eexigidas para completar o trabalho apresentado.

As principais diferencças a anotar serão a adição de luz e de texturas ao cenário que será visualizado no trabalho quando este for executado.

2 Introdução

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

Iremos também descrever e explicar as alterações feitas ao *generator* e ao *engine*, bem como o seu funcionamento apresentando algumas imagens de exemplo para facilitar a compreensão. Iremos também especificar os diversos raciocínios utilizados para ultrapassar as barreiras encontradas.

3 Organização do Código

Tendo em conta a implementação realizada na primeira fase do projeto, optamos por seguir a abordagem estabelecida anteriormente, pelo que continuamos a ter duas aplicações, o *Generator* e o *Engine*.

3.1 *Generator*

Nesta fase não houve a implementação de novas primitivas, houve no entanto várias mudanças que foram efetuadas a certas primitivas para que estas agora consigam organizar nos seus ficheiros "*3d*"os dados representam as normais de todos os pontos de cada objeto. As formas que foram alteradas nesta fase foram: a esfera, o *torus*, e o *teapot*.

As normais neste trabalho são estruturadas de forma semelhante aos pontos que são guardados nos ficheiros *3d*. Através do *generator* é criado uma lista do vetores que irão representar cada normal de cada ponto, e cada uma é composta por três coordenadas (x, y e z), depois todas as normais serão armazenadas no mesmo ficheiro onde serão guardados os pontos (mantendo a mesmo ordem).

3.2 Esfera

O cálculo das normais nos pontos da esfera é feito de uma maneira relativamente simples, tendo em conta que a esfera quando é criada inicialmente no *generator* não é afetada por nenhuma transformação (principalmente de translação ou de alteração de escala) então

a esfera irá ter como posição inicial a origem do referêncial e todos os pontos da forma geométrica serão equidistantes da origem.

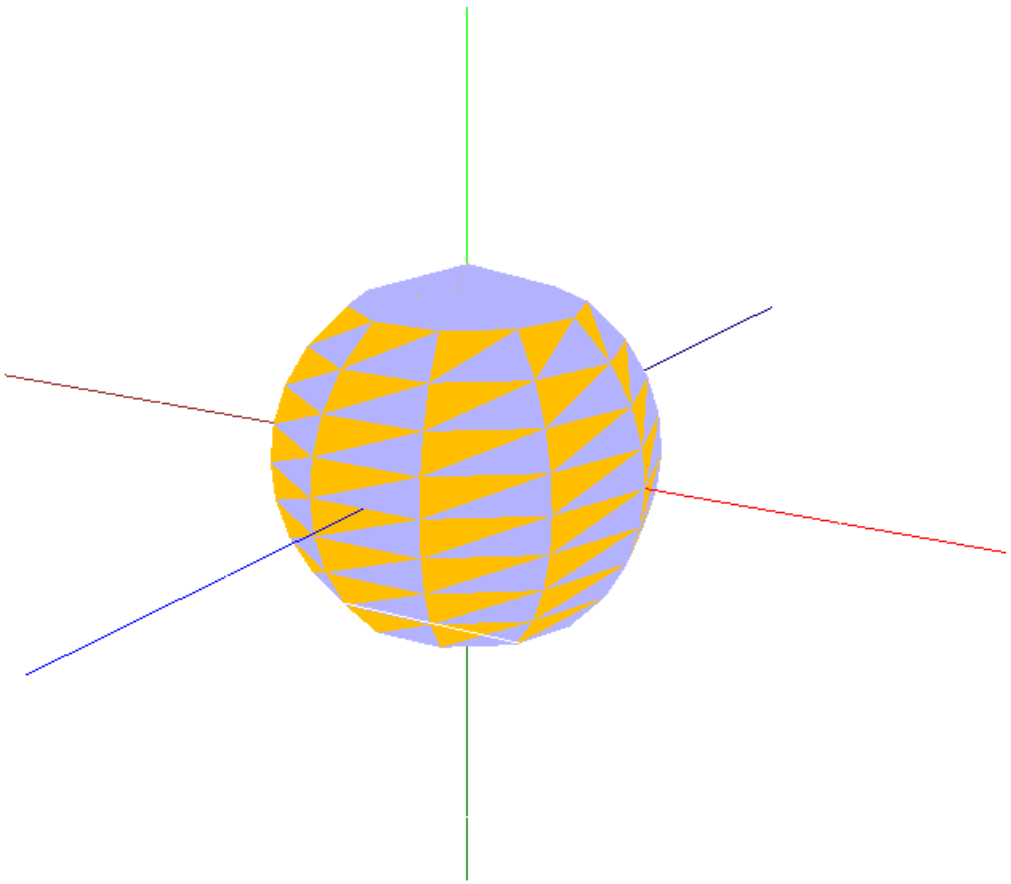

Figura 1: Esfera no centro do referêncial

Tendo estes dados em consideração, podemos ver que é muito simples obter a normal de cada ponto, visto que esta consiste apenas em subtrair as coordenadas de cada ponto com as coordenadas da origem (x = 0.0, y = 0.0, z = 0.0). Ou seja, as normais irão assumir exatamente os mesmos valores que os respetivos pontos.

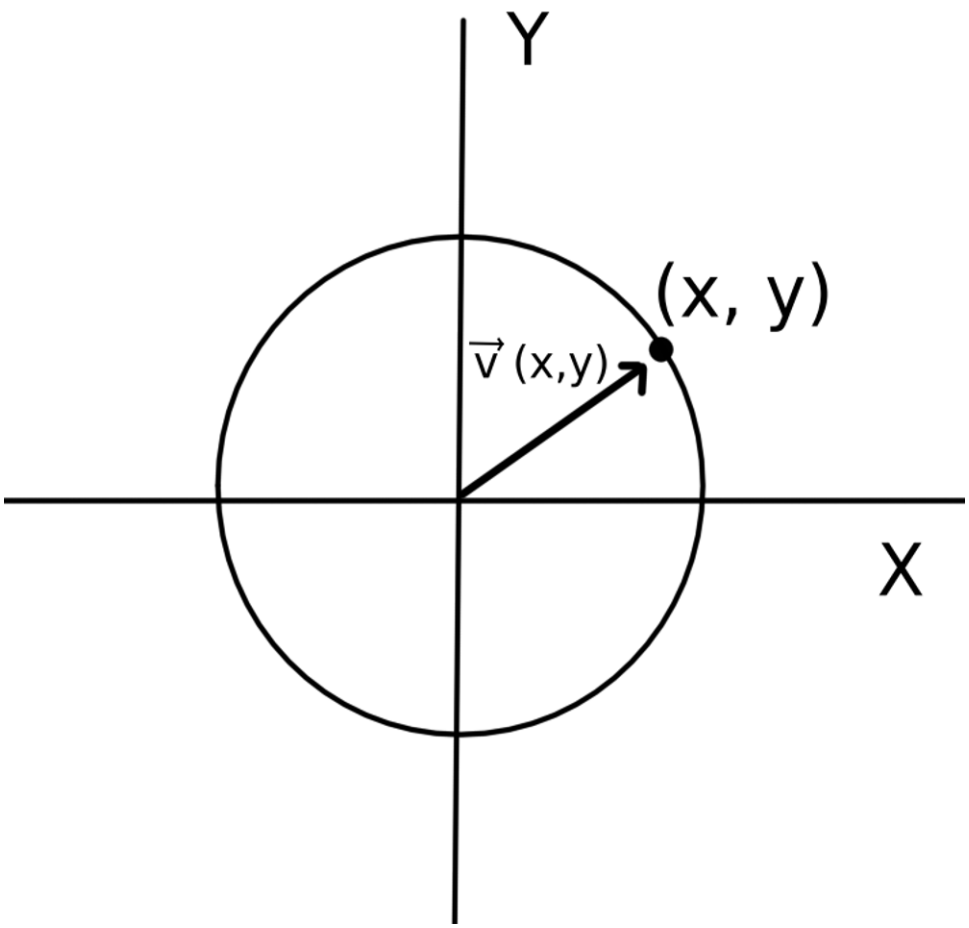

Figura 2: Exemplo do racioncínio efetuado (mas apenas com duas dimensões)

3.3 *Torus*

Com o torus não podemos usar o mesmo método que foi usado para calcular as normais da esfera, visto que nem todos os pontos são equidistantes da origem. Contudo, o método usado para obter os vetores perpendiculares da superfície do *torus* também tem uma solução simples.

Para desenhar os pontos do *torus* é necessário que haja um conjunto de pontos de referência que se situam no mesmo plano e que ao mesmo tempo descrevam uma circunferência no espaço (esta circunferncia ficaria contida no interior da superfície do *torus*).

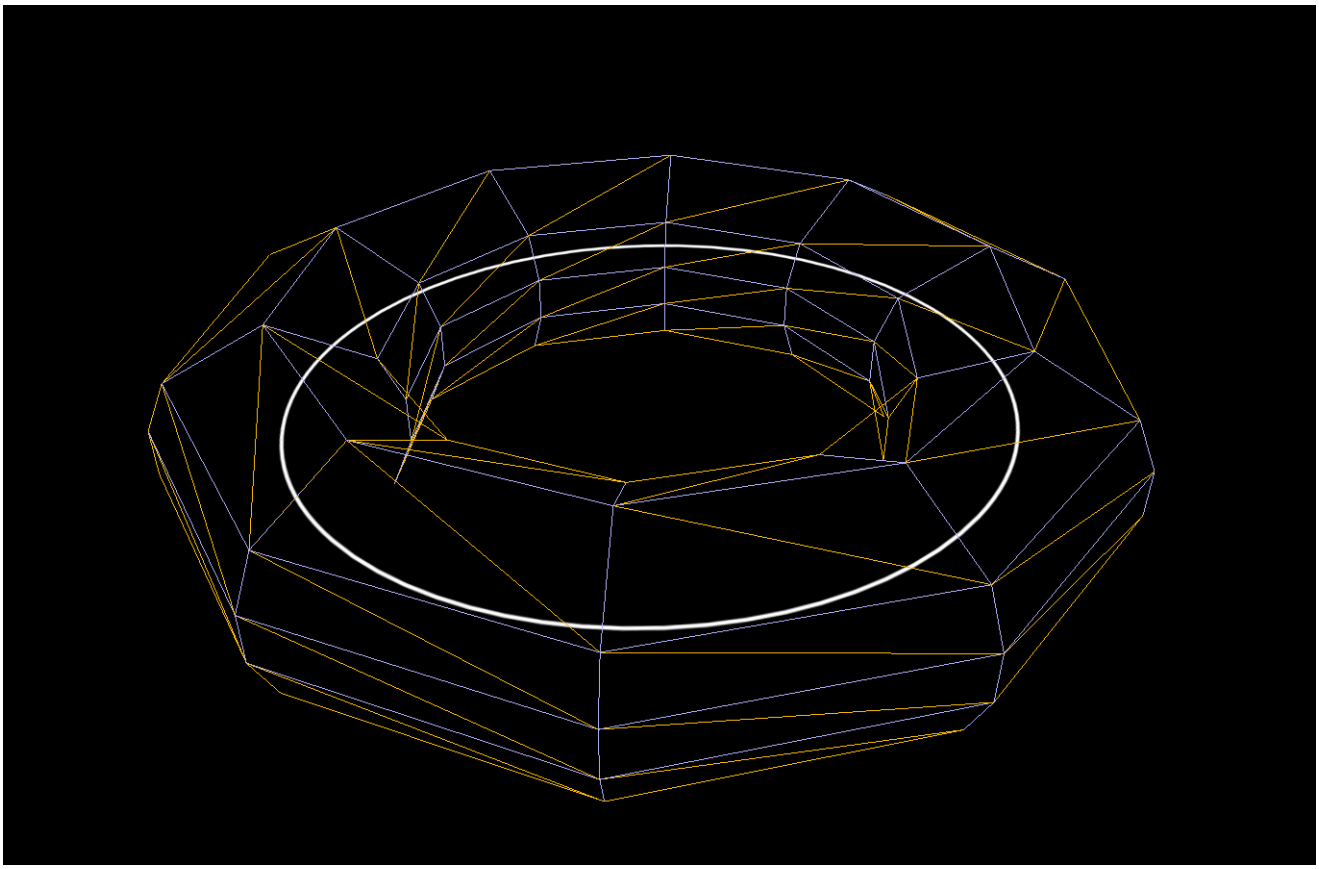

Figura 3: Demonstração do *torus* e da sua cinrcunferência interior

Depois de obter esse conjunto de pontos de referência podemos começar a calcular os pontos da superfície do *torus*, apenas certificando que todos os pontos de cada anel do *torus* são equidistantes de um certo ponto da circunferência que foi mencionada.

Tendo isto em consideração, podemos usar os pontos de referência dessa circunferência para obter um vetor perpendicular à superfície da figura na posição de um certo ponto.

```
p1.x = radius * sin(ang)  + or1 * sin(ang);
p1.y = oy;
p1.z = radius * cos(ang) + or1 * cos(ang);

p2.x = radius * sin(ang + delt) + or1 * sin(ang + delt);
p2.y = oy;
p2.z = radius * cos(ang + delt) + or1 * cos(ang + delt );

p3.x = radius * sin(ang) + or2 * sin(ang);
p3.y = oy + h;
p3.z = radius * cos(ang) + or2 * cos(ang);

points.push_back(p1);
points.push_back(p2);
points.push_back(p3);
```

```
p1.x = or1 * sin(ang);
p1.y = oy;
p1.z = or1 * cos(ang);

p2.x = or1 * sin(ang + delt);
p2.y = oy;
p2.z = or1 * cos(ang + delt );

p3.x = or2 * sin(ang);
p3.y = oy + h;
p3.z = or2 * cos(ang);

normals.push_back(normalizeVec(p1));
normals.push_back(normalizeVec(p2));
normals.push_back(normalizeVec(p3));
```

Figura 4: Comparação do código para calcular pontos (esquerda) e para calcular normais (direita)

3.4 *Teapot*

Até agora o calculo das normais tem sido feito em função de cada primitiva, pois cada uma utiliza um conjunto de regras pré-determinadas para obter as normais da mesma primitiva.

A superfície do *teapot* comporta-se de forma diferente, pois é esta é determinada em função de um ficheiro *patch* e por causa disso a abordagem para obter as normais terá que ser diferente. Da mesma forma que a função que calcula os pontos de uma superfície de **Bezier** têm que funcionar para qualquer ficheiro *patch*, então a função que calcula as normais da superfície também tem que funcionar para qualquer ficheiro que lhe seja apresentado.

Todas as funções do *generator* que são usadas para calcular as normais de uma superfície de *Bezier* são baseadas nas funções já existem para calcular os respetivos pontos. A função *createBezierSurfNormals* possúi uma estrutura quase idêntica à função *createBezierSurf*, só que em vez de armazenar pontos de superfície armazena vetores normais, e faz isso obtendo dois vetores que são tangentes à superfície criada por um *Bezier patch* num determinado ponto, depois de obter os dois vetores é possível obter a normal calculando o produto externo dos dois vetores.

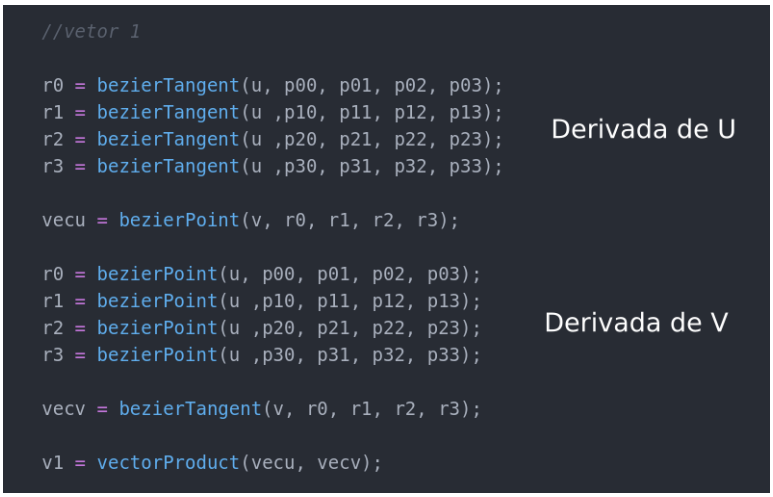

Figura 5: Cálculo de um vetor normal dentro da função *createBezierSurfNormals*

A função *bezierPoint* calcula um ponto de uma curva de *bezier* e a função *bezierTangent* calcula um vetor tangente à curva de *bezier* em função de um certo valor de "t".

Cada vetor é obtido derivando ou os valores de "u"ou os valores de "v"e as funções *bezierTangent* e *bezierPoint* vão alternando dependendo da situação.

```
Point bezierPoint(float t, Point p0, Point p1, Point p2, Point p3){

    Point p;
    float it = 1.0f - t;
    float at = pow(it, 3);
    float bt = 3 * t * pow(it, 2);
    float ct = 3 * pow(t, 2) * it;
    float dt = pow(t, 3);

    p.x = (at*p0.x) + (bt*p1.x) + (ct*p2.x) + (dt*p3.x);
    p.y = (at*p0.y) + (bt*p1.y) + (ct*p2.y) + (dt*p3.y);
    p.z = (at*p0.z) + (bt*p1.z) + (ct*p2.z) + (dt*p3.z);

    return p;
}

Point bezierTangent(float t, Point p0, Point p1, Point p2, Point p3){

    Point p;
    float at = (-3 * t * t) + (6 * t) - 3;
    float bt = 3 - (12 * t) + (9 * t * t);
    float ct = (6 * t) - (9 * t * t);
    float dt = 3 * t * t;

    p.x = (at*p0.x) + (bt*p1.x) + (ct*p2.x) + (dt*p3.x);
    p.y = (at*p0.y) + (bt*p1.y) + (ct*p2.y) + (dt*p3.y);
    p.z = (at*p0.z) + (bt*p1.z) + (ct*p2.z) + (dt*p3.z);

    return p;
}
```

Figura 6: Comparação entre as funções *bezierPoint* e *bezierTangent*

4 Engine

Nesta fase foi necessário implementar funcionalidades de iluminação e textura.

4.1 Parser

=> Ficheiros de Input

O formato dos ficheiros foi alterado pelo que necessitamos de adaptar o parsing dos ficheiros XML para representar um determinado caso. De seguida, é apresentado um exemplo de escrita em formato XMLv para cada uma das novas funcionalidades:

-> Iluminação : devem ser indicadas cada uma das luzes

```
<lights>
        <light type="POINT" posX="-50" posY="25" posZ="0" />
    </lights>
```

-> Texturas : ddevem ser indicadas a localizção da textura bem como o respetivo ficheiro.

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

-> Materiais : as primitivas podem estar associadas a materiais

```
<model file="box.3d"
        texture="../../texture/cube.jpg" emiR="1" emiG="1" emiB="1" />
</models>
```

Assim, decidimos implementar funções responsaveis pelo parsing de iluminação e de materiais Para além disoo uma outra para de modo a permitir a aplicação de texturas no projeto.

4.2 Point

Nesta aplicação mantivemos a estrutura Point já desenvolvida numa fase anterior.

4.3 Transformations

Esta classe, tal como na fase anterior, é responsável por guardar todas as informa- ções relativas a uma dada transformação. Para além disso, nesta são implementadas as curvas de Catmull-Rom.

4.4 Light

De modo a possibilitar a iluminação dos objetos caso esta lhes incida, desenvolve-mos esta classe, que conterá a informação de uma determinada luz.

=> Cor : Tendo em conta que é possível associar diferentes cores a uma luz, optamos por utilizar as seguintes:

- GL_AMBIENTE : intesidade de uma luz ambiente que uma fonte permite num dado cenário.
- GL_DIFUSE : uma fonte pode projetar uma luz direcional. Quando esta atinge outro objeto espalha-se de forma uniforme pela superficie.
- GL_ESPECULAR tem impacto na cor do destaque especular de um dado objeto.

=> Posição Para além das diferentes cores, incluímos dois tipos de luzes, uma pontual e uma direcional.

- POINT : As coordenadas indicadas representam a posição exata da luz, emitindo a luz para todas as direções a partir deste ponto. Para uma coordenada x, y e z, teremos *position = x,y,z,1*
- DIRECTIONAL Neste caso as coordenadas caracterizam a direção da luz. Para uma coordenada x, y e z, teremos *position = x,y,z,0*

De modo a aplicar a cada luz as propriedades indicadas elaboramos a draw.

```
void Light::apply(GLenum number){
    glLightfv(GL_LIGHT0+number, GL_POSITION, info);

    for (const int atrb : attributes){
        switch(atrb){
            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;
        }
    }
}
```

4.5 Texture

Nesta classe são introduzidos parâmetros necessários à representação de cores produzidas através de vetores com a informação respetiva. Assim temos os vetores *diffuse ambient, specular e emission*, Estes são representados através da primitiva *Transformations*. Desta forma, o conteúdo de cada será:

- GL_DIFFUSE : Aplicado o material e a luz branca, define-se a cor primária do modelo.
- GL_EMISSION : Emite a luz do próprio material após este ser aplicado.
- GL_SPECULAR: A superficie parece brilhante após a aplicação do material.
- GL_AMBIENT: Aplicado o material, o modelo reflete a cor em questão. Em todas as superfícies, a luz de incidência é igual.

De modo a aplicarmos os materiais implementamos a função draw.

```
void Texture::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);
}
```

4.6 Shape

Para melhorar o desempenho do programa na fase anterior implementamos VBO's para as primitivas. Devido à alteração dos ficheiros .3d, que agora para além das primitivas contêm as normais e as coordenadas das texturas, foi necessário modificar a *prepareBuffer*:

```
void Shape::prepareBuffer(...){
    ...
    glGenBuffers(3,buffer);
    glBindBuffer(GL_ARRAY_BUFFER, buffer[0]);
    glBufferData(GL_ARRAY_BUFFER,
                sizeof(float) * numVertex[0] * 3,
                vertexs,
                GL_STATIC_DRAW);

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

    glBindBuffer(GL_ARRAY_BUFFER, buffer[2]);
    glBufferData(GL_ARRAY_BUFFER,
                sizeof(float) * numVertex[2],
                &(texture[0]),
                GL_STATIC_DRAW);
    ...
}
```

Desta forma, a draw também teve de sofrer alterações:

```
void Shape::draw(){
    glBindBuffer(GL_ARRAY_BUFFER, buffer[0]);
    glVertexPointer(3, GL_FLOAT, 0, 0);
```

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

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

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

Durante o processo de leitura do ficheiro de configuração é possível carregar uma textura, pelo que recorremos à loadTexture caso o modelo tenha alguma textura associada. Esta permite carregar em memória os dados.

4.7 geoTransforms

Tal como nas fases anteriores esta classe é responsável por guardar toda a informação de um grupo, nomeadamente as transformações geométricas, as primitivas e ainda os grupos filhos.

4.8 Scene

De modo a facilitar o desenho do cenário optámos por criar esta classe que contém as luzes bem como o grupo principal.

```
class Scene {
    private:
        vector<Light*> lights;
        geoTransforms *mainGroup;
    ...
}
```

Nesta definimos a *applyLights* que irá auxiliar na aplicação das luzes:

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

Desta forma para aplicarmos as luzes, na engine, basta recorrer à *applyLights* após a aplicação das propriedades da câmera. O desenho do resto do cenário é realizado de forma semelhante à fase anterior.

5 Conclusão

De uma perspetiva geral, consideramos que a realização desta fase foi relativamente mal sucedida, visto que não fomos capazes de cumprir todos os requisitos estabelecidos exigidos.

Acabou por ser um projeto bastante complexo e que requer uma grande capacidade de planeamento, de organização e de compreensão pelas bibliotecas que se usam para trabalhar. Aos olhos do grupo, o projeto começou com um andamento constante, mas eventualmente acabou por mostrar ser algo bastante complicado, de certa forma nos mostrou que o desenvolvimento de gráficos é um conceito que exige um grande entendimento de várias áreas e que nenhuma delas deve ser subestimada.

Em suma, após terminar a realização deste projeto, concluímos que as matérias lecionadas deviam ter sido mais aprofundadas para que fosse possível realizar este trabalho com uma melhor eficiência.