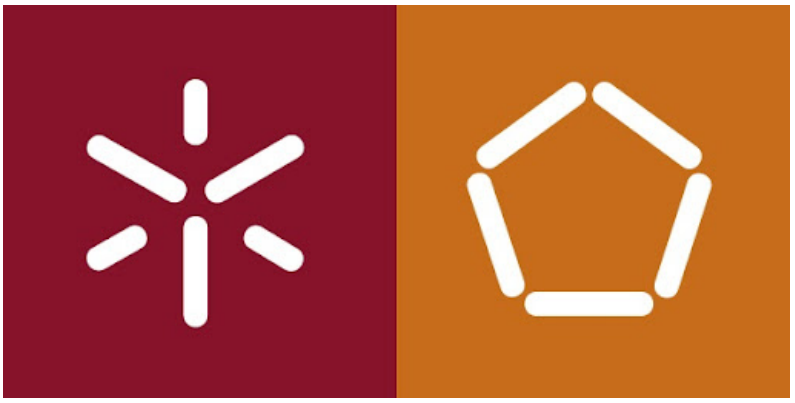
Universidade do Minho
Escola de Engenharia



Normals and Texture Coordinates

Computação Gráfica

Relatório do Trabalho Prático

Grupo nº44
Alexandre Eduardo Vieira Martins A93242
José Eduardo Silva Monteiro Santos Oliveira A100547
Pedro Afonso Moreira Lopes A100759
Pedro Silva Ferreira A97646

26 de maio de 2024

Índice

Introdução

Nesta última fase do trabalho, abordam-se os temas da aplicação de luz e de texturas ao programa, de forma a conseguir gerar cenários de maior complexidade e realismo.

Houve complexidade adicionada aos ficheiros XML. Visto duma forma abstrata:

Material/Cor do Modelo

Encapsulada em cada modelo, temos informação sobre a textura que ele vai usar e o comportamento da superfície perante a luz (as suas propriedades materiais), sendo que os campos definem a cor emitida pelo irradiar do tipo específico de iluminação na superfície.

- ***iluminação difusa:*** provém de uma determinada fonte pontual (como o Sol) e atinge as faces com uma intensidade consoante estarem viradas para ou contra a luz, e irradiará igualmente em todas as direcções.

- ***iluminação ambiente:*** vinda uniformemente de todas as direções contra os polígonos da cena. É uma aproximação à luz do céu, analogamente, logo poderia ser atribuída a uma skybox no contexto dum cenário gráfico.

- ***highlight especular:*** refere-se ao foco algo mais claro consequente da reflexão da luz na superfície.

- ***iluminação emitida:*** a cor irradiada pelo próprio modelo uniformemente para fora, que neste caso vai ser mais útil para o Sol.

- ***brilho*****:** a magnitude com a qual se vai notar o highlight especular, ou seja, o controlo do seu raio e de quão espalhado vai estar pela superfície.

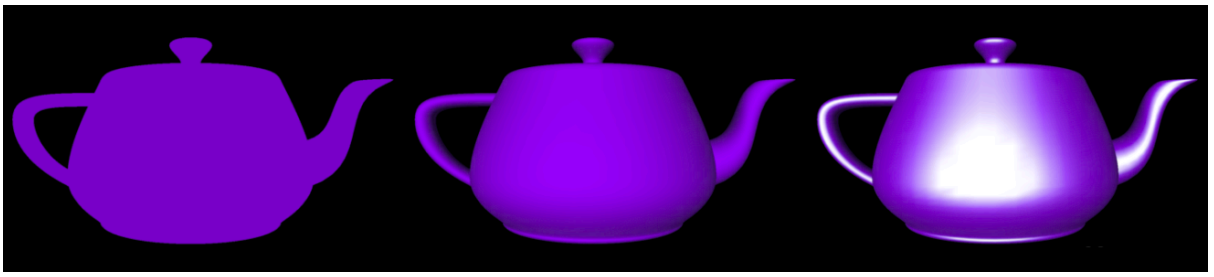


Figura 1 *- combinação das iluminações. só ambiente, + difusa, + especular.*

Tipos das Luzes

- ***luz de ponto:*** caracterizada por uma iluminação omnidirecional a partir duma origem definida, análoga à lâmpada dum candeeiro. Vai atenuando à medida que a distância desse ponto origem aumenta.

- ***luz direcional:*** caracterizada por um vetor de direção apenas. A inexistência duma origem concreta faz com que a iluminação não enfraqueça com a distância, pois não tem um começo propriamente dito. Igualmente, faz com que não haja o conceito de distância entre a luz e uma superfície iluminada por esta.

- ***luz de holofote:*** combina as características das duas anteriores - tem uma origem definida, mas também irradia segundo um vetor direção. Estas duas propriedades fazem com que a luz siga uma forma de cone em termos volumétricos, em que a distância origem-superfície serve de altura e a base circular (a área iluminada) baseia-se tanto nela como no ângulo de cutoff (definido como o ângulo entre o eixo de altura do cone e o seu lado), que garante que só as superfícies a menor ângulo para com o vetor são abrangidas.

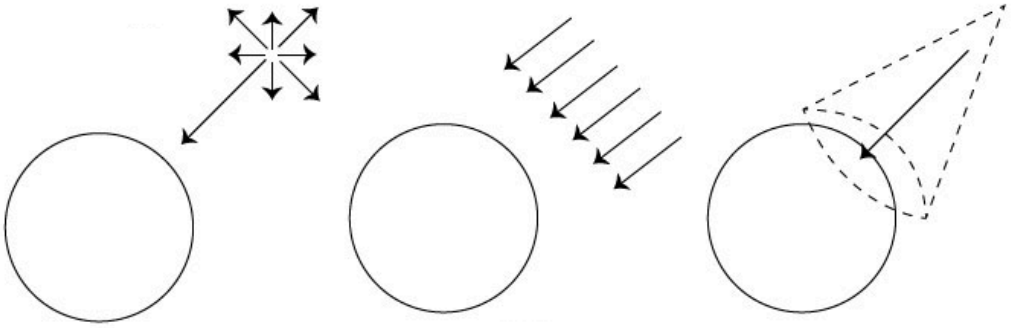


Figura 2 *- luzes de ponto, direcional e de holofote, nesta ordem.*

Generator

Cálculo das Normais & Mapeamento das Texturas

1. **Planos**

```
Coordenadas normalCoords1 = {0.0f,1.0f,0.0f};
normal.pontos.push_back(normalCoords1);
normal.pontos.push_back(normalCoords1);
normal.pontos.push_back(normalCoords1);
normal.pontos.push_back(normalCoords1);
normal.pontos.push_back(normalCoords1);
normal.pontos.push_back(normalCoords1);
normals.push_back(normal);

Coordenadas texCoords1 = {x_tex,y_tex,0};
Coordenadas texCoords2 = {x_tex,y_tex - lado_text_length,0};
Coordenadas texCoords3 = {x_tex + lado_text_length,y_tex - lado_text_length,0};
Coordenadas texCoords4 = {x_tex + lado_text_length,y_tex,0};
texts.pontos.push_back(texCoords1);
texts.pontos.push_back(texCoords2);
texts.pontos.push_back(texCoords3);
texts.pontos.push_back(texCoords3);
texts.pontos.push_back(texCoords4);
texts.pontos.push_back(texCoords1);
texCoords.push_back(texts);
```

***Figura 3** - cálculo das normais & mapeamento em planos.*

É bastante direta a obtenção das normais no contexto do plano, pois são sempre *(0,1,0)* para todos os vértices, dada a sua orientação. É Igualmente simples o mapeamento da textura, pois resume-se a associar os seus extremos aos da divisão de momento a ser abordada na iteração. Essencialmente, havendo 6 pontos nos dois triângulos que formam uma divisão, fica-se com o equivalente destes pontos no plano da textura em si, ficando assim a textura concisa em cima do plano.

Cálculo das Normais & Mapeamento das Texturas (cont.)

2. **Caixas**

```
// Face de baixo //
Coordenadas normalCoords1 = {0,-1,0};
normals.pontos.push_back(normalCoords1);
normals.pontos.push_back(normalCoords1);
normals.pontos.push_back(normalCoords1);
normals.pontos.push_back(normalCoords1);
normals.pontos.push_back(normalCoords1);
normals.pontos.push_back(normalCoords1);
normalCoords.push_back(normals);

Coordenadas texCoords1 = {x_tex,y_tex,0};
Coordenadas texCoords2 = {x_tex,y_tex - lado_text_length,0};
Coordenadas texCoords3 = {x_tex + lado_text_length,y_tex - lado_text_length,0};
Coordenadas texCoords4 = {x_tex + lado_text_length,y_tex,0};
texts.pontos.push_back(texCoords3);
texts.pontos.push_back(texCoords2);
texts.pontos.push_back(texCoords1);
texts.pontos.push_back(texCoords1);
texts.pontos.push_back(texCoords4);
texts.pontos.push_back(texCoords3);
textureCoords.push_back(texts);
```

***Figura 4** - cálculo das normais & mapeamento em caixas.*

Como uma caixa acaba por ser um conjunto de planos, cada coleção de vértices pertencentes a um único plano terão normais iguais. Se no plano ficavam todos (0,1,0), aqui ficarão…

…(0,1,0) para os de **cima**.
…(0,-1,0) para os de **baixo**.
…(0,0,1) para os da **frente**.
…(0,0,-1) para os de **trás**.
…(-1,0,0) para os da **esquerda**.
…(1,0,0) para os da **direita**.

Uma única textura cobrirá todas as faces da caixa. Sendo assim, acontece que para cada vértice *(i,j)* inserido numa face corresponderá a coordenada *(i / nr divisões, j / nr divisões)* da textura.

Cálculo das Normais & Mapeamento das Texturas (cont.)

3. Esferas

```
const float texture_x_shift = 1.0 / slices;
const float texture_y_shift = 1.0 / stacks;

float x = radius * sin(nextStack) * sin(nextSlice);
float y = (radius * cos(nextStack));
float z =(radius * sin(nextStack) * cos(nextSlice));
float vLen = sqrtf(x*x+y*y+z*z);

Coordenadas p0 = { x , y, z};
Coordenadas p0n = {x / vLen,y / vLen ,z / vLen};
Coordenadas t0 = {next_texture_x, next_texture_y,0.};

x = (radius * sin(nextStack) * sin(currentSlice));
z = (radius * sin(nextStack) * cos(currentSlice));
Coordenadas p1 = { x, y, z };
Coordenadas p1n = {x / vLen,y / vLen ,z / vLen};
Coordenadas t1 = {texture_x, next_texture_y,0.};

x = (radius * sin(currentStack) * sin(nextSlice));
y = (radius * cos(currentStack));
z = (radius * sin(currentStack) * cos(nextSlice));
Coordenadas p2 = { x, y, z };
Coordenadas p2n = {x / vLen,y / vLen ,z / vLen};
Coordenadas t2 = {next_texture_x, texture_y,0.};

x = (radius * sin(currentStack) * sin(currentSlice));
z = (radius * sin(currentStack) * cos(currentSlice));
Coordenadas p3 = { x , y, z };
Coordenadas p3n = {x / vLen,y / vLen ,z / vLen};
Coordenadas t3 = {texture_x, texture_y,0.};
```

Figura 5 *- cálculo das normais & mapeamento em esferas.*

No caso das esferas, as normais são obtidas pela normalização das componentes nas coordenadas de cada vértice. Nas coordenadas da textura, associa-se…

… *índice da slice / total de slices* para **x**.

… *índice da stack / total de stacks* para **y**.

Cálculo das Normais & Mapeamento das Texturas (cont.)

4. Cones

Apesar de termos tentado a aplicação da textura para o cone, não foi conseguida uma implementação correta das texturas no cone a tempo da entrega. Resumidamente, tentamos seguir esta linha de pensamento demonstrada na imagem seguinte, em que dividíamos a textura pelas slices e stacks do cone, tendo assim as texturas para cada retângulo do cone e íamos atribuindo cada vértice a uma coordenada do gráfico das texturas.

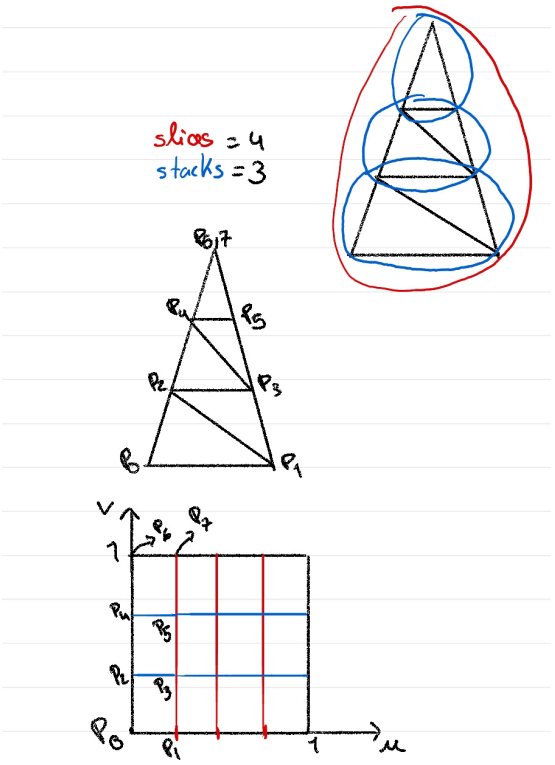


Figura 6 - *Gráfico para a aplicação das texturas no cone*

Cálculo das Normais & Mapeamento das Texturas (cont.)

5. **Modelos Bézier**

```
for (int v = 0; v ⩽ tesselation; v++)
            {
                float vf = float(v) / tesselation;
                float v_vetor[4] = {vf * vf * vf, vf * vf, vf, 1};
                float v_vetor_deriv[4] = {vf * vf * 3, vf * 2, 1, 0};

                Coordenadas u_vetorXcalculada[4], puv;
                Coordenadas u_vetorXcalculada_uderiv[4], puv_uderiv, puv_vderiv;
                for (auto i = 0; i < 4; i++)
                {
                    u_vetorXcalculada[i] = multMatrixVector(u_vetor, matrix_MPM_trans[i]);
                    u_vetorXcalculada_uderiv[i] = multMatrixVector(u_vetor_deriv, matrix_MPM_trans[i]);
                }
                puv = multMatrixVector(v_vetor, u_vetorXcalculada);
                pontosFinais[u][v] = puv;
                puv_uderiv = multMatrixVector(v_vetor, u_vetorXcalculada_uderiv);
                puv_vderiv = multMatrixVector(v_vetor_deriv, u_vetorXcalculada);
                Coordenadas normal = puv_vderiv.get_cross_product(puv_uderiv);
                normal.normalize();
                normaisFinais[u][v] = normal;
            }
```

***Figura 7** - cálculo das normais para vértices em modelos de patches Bézier.*

Para o cálculo das normais para os modelos baseados em patches Bézier, foi seguido o seguinte raciocínio mencionado nas aulas, pelas seguintes fórmulas, tal como mencionado na fase anterior:

$$\frac{\partial B(i,j)}{\partial i} = [3i^2, 2i, 1, 0].M.P.M^T.V^T$$

$$\frac{\partial B(i,j)}{\partial j} = U.M.P.M^T.[3j^2, 2j, 1, 0]^T$$

Figura 8 - fórmulas para o cálculo dos vetores tangentes.

…obtendo estas derivadas parciais, o vetor normalizado será o resultado normalizado do cross product dos vetores tangentes ao ponto. Adicionalmente, a textura *(i,j)* corresponderá ao ponto *(i,j)*, sendo que ambos vão variar entre 0 e 1 dada a normalização.

Infelizmente, atingimos um equívoco onde quase todas as normais foram calculadas corretamente, à exceção de uma porção pequena que, sem identificável razão, davam valores muito estranhos, apesar de todos os cálculos estarem a seguir a mesma lógica.

Engine

Parsing

As novidades no parsing andam à volta da assimilação das cores/materiais, das texturas e das luzes. Foram criadas novas classes para cada um destes, transferindo para instâncias a informação vinda do XML.

```
void processTextureElement(tinyxml2::XMLElement* textureElement, Model& m) {

    const char* file = textureElement->Attribute("file");
    filePath = "../build/textures/" + std::string(file);
    filePaths.push_back(filePath);

    ...

    Texture texture = Texture(filePath);
    texture.prep();
    m.texture = texture;
    m.hasTexture = true;
}
```

***Figura 9** - parsing das texturas para instância respetiva*

```
void processLightElement(tinyxml2::XMLElement* child) {
    float px, py, pz, dx, dy, dz, sx, sy, sz, sdx, sdy, sdz, c;

    const char* childName = child->Attribute("type");
    Light light = Light(light_id);
    light_id++;

    if (strcmp(childName, "point") == 0) {
        // guardar parametros
    }
    else if (strcmp(childName, "directional") == 0) {
        // guardar parametros
    }
    else if (strcmp(childName, "spot") == 0) {
        // guardar parametros
    }

    light.prep();
    lights.push_back(light);

}
```

***Figura 10** - parsing e armazenamento das luzes em vetor*

Primeiramente é feito o parsing dos parâmetros do group light no XML. Aqui lemos os seus parâmetros e guardamos em variáveis globais para posteriormente aplicar as luzes ao desenhar o modelo.

Gestão das Texturas & Cores

Foram organizadas as cores e as texturas recebidas no XML em classes. A classe *Color* permite conter as coordenadas duma instância de cor passada de input e chamar *glMaterial()* de forma mais abstraída. Na classe *Texture,* tem-se o método *Texture.prep()* que tem a responsabilidade de carregar a textura em questão e adaptá-la para uso em OpenGL.

```
class Color{
    public:
        Coordenadas diffuse, ambient, specular, emissive;
        bool hasDiffuse, hasAmbient, hasSpecular, hasEmissive, hasShininess;
        float shininess;
    public:
        Color(){
            hasDiffuse = false;
            hasAmbient = false;
            hasSpecular = false;
            hasEmissive = false;
        }

    void apply(){
        float diffuseArr[4] = {diffuse.x/255, diffuse.y/255, diffuse.z/255, 1.0};
        float ambientArr[4] = {ambient.x/255, ambient.y/255, ambient.z/255, 1.0};
        float specularArr[4] = {specular.x/255, specular.y/255, specular.z/255, 1.0};
        float emissiveArr[4] = {emissive.x/255, emissive.y/255, emissive.z/255, 1.0};
        if (hasDiffuse) glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuseArr);
        if (hasAmbient) glMaterialfv(GL_FRONT, GL_AMBIENT, ambientArr);
        if (hasSpecular) glMaterialfv(GL_FRONT, GL_SPECULAR, specularArr);
        if (hasEmissive) glMaterialfv(GL_FRONT, GL_EMISSION, emissiveArr);
        if (hasShininess) glMaterialf(GL_FRONT, GL_SHININESS, shininess);
    }
};
```

```
class Texture{
    public:
        GLuint texture_id;
        unsigned int t, tw, th, tex;
        unsigned int texID;
        unsigned char *texData;
        std::string path;
    public:
        Texture(){};

        Texture(std::string pathName){
            path = pathName;
        }
...
```

***Figuras 11 e 12** - classes Color & Texture*

Havendo agora um campo Color e um campo Texture na classe Model, a deteção de um modelo com uma textura leva à vinculação da mesma no VBO, com as funções *glBindBuffer()* e *glBindTexture()* (esta última abstraída por *Texture.apply()*). Adicionalmente, existe a opção de serem desenhadas as normais com a tecla N, que torna *draw_normals* verdadeira.

```
void draw_model(Model &m){

    ........

    if(m.hasColor){
        glEnable(GL_LIGHTING);
        m.color.apply();
    }
    ........

    if (m.hasTexture){
        glBindBuffer(GL_ARRAY_BUFFER, m.vbo_ids[1]);
        m.texture.apply();
        glTexCoordPointer(2, GL_FLOAT, 0, 0);
    } else glDisableClientState(GL_TEXTURE_COORD_ARRAY);
    ........
```

```
    ........

    if(draw_normals){

        glDisable(GL_LIGHTING);
        glColor3f(1.0f, 0.0f, 0.0f);
        glBegin(GL_LINES);

        for(int i = 0; i < m.coords[0].size(); i+=3){
            glVertex3f(m.coords[0][i], m.coords[0][i+1],
                       m.coords[0][i+2]);

            glVertex3f(m.coords[0][i] + m.coords[2][i],
                       m.coords[0][i+1] + m.coords[2][i+1],
                       m.coords[0][i+2] + m.coords[2][i+2]);
        }
        glEnd();
    }
}
```

***Figuras 13 e 14** - função draw_model(), agora com testes para textura e para cor e opcional desenho das normais*

Gestão das Luzes

Para auxiliar a aplicação das luzes, criamos uma classe *Light*. Esta classe contém variáveis booleanas que nos indicam se existe fonte de luz ou não, qual o seu tipo e campos que guardam as propriedades da luz em questão.

```
class Light{
    public:
    Coordenadas position;
    Coordenadas direction;
    float cutoff;
    bool hasPosition, hasDirection;
    GLenum gl_light;

    Light(GLenum lightID){
        hasPosition = false;
        hasDirection = false;
        gl_light = lightID;
    }
}
```

Figura 15 *- classe Light*

O método Light.prep() faz a configuração do ambiente necessária para aplicar as luzes, enquanto que o Light.apply() aplica as definições de luz predefinidas no ambiente, tal como se pode ver nas seguintes imagens.

```
class Light{
 void prep(){
        float amb[4] = {1.0f, 1.0f, 1.0f, 1.0f};
        glLightModelfv(GL_LIGHT_MODEL_AMBIENT, amb);
        glEnable(GL_LIGHT0 + gl_light);
        glLightfv(GL_LIGHT0 + gl_light, GL_DIFFUSE, white);
        glLightfv(GL_LIGHT0 + gl_light, GL_SPECULAR, white);
    }
}
```

Figura 16 *- método Light.prep()*

Gestão das Luzes (cont.)

```
class Light{
 void apply(){
        direction.normalize();
        if (hasDirection && hasPosition) {
            float posArr[4] = {position.x, position.y, position.z, 1};
            glLightfv(GL_LIGHT0 + gl_light, GL_POSITION, posArr);
            float dirArr[3] = {direction.x, direction.y, direction.z};
            glLightfv(GL_LIGHT0 + gl_light, GL_SPOT_DIRECTION, dirArr);
            glLightfv(GL_LIGHT0 + gl_light, GL_SPOT_CUTOFF, &cutoff);
        }
        else if (hasPosition){
            float posArr[4] = {position.x, position.y, position.z, 1};
            glLightfv(GL_LIGHT0 + gl_light, GL_POSITION, posArr);
        }
        else if (hasDirection){
            float dirArr[4] = {direction.x, direction.y, direction.z,0};
            glLightfv(GL_LIGHT0 + gl_light, GL_POSITION, dirArr);
        }
    }
}
```

Figura 17 *- método Light.apply()*

Demo Scene

Para a Demo Scene desta fase, de maneira a dar por concluído o cenário do sistema solar, aplicamos então as texturas todas aos seus respetivos planetas e uma luz para o sol, de forma a este ser a estrela que é, tendo esta cena então ficado da seguinte forma:

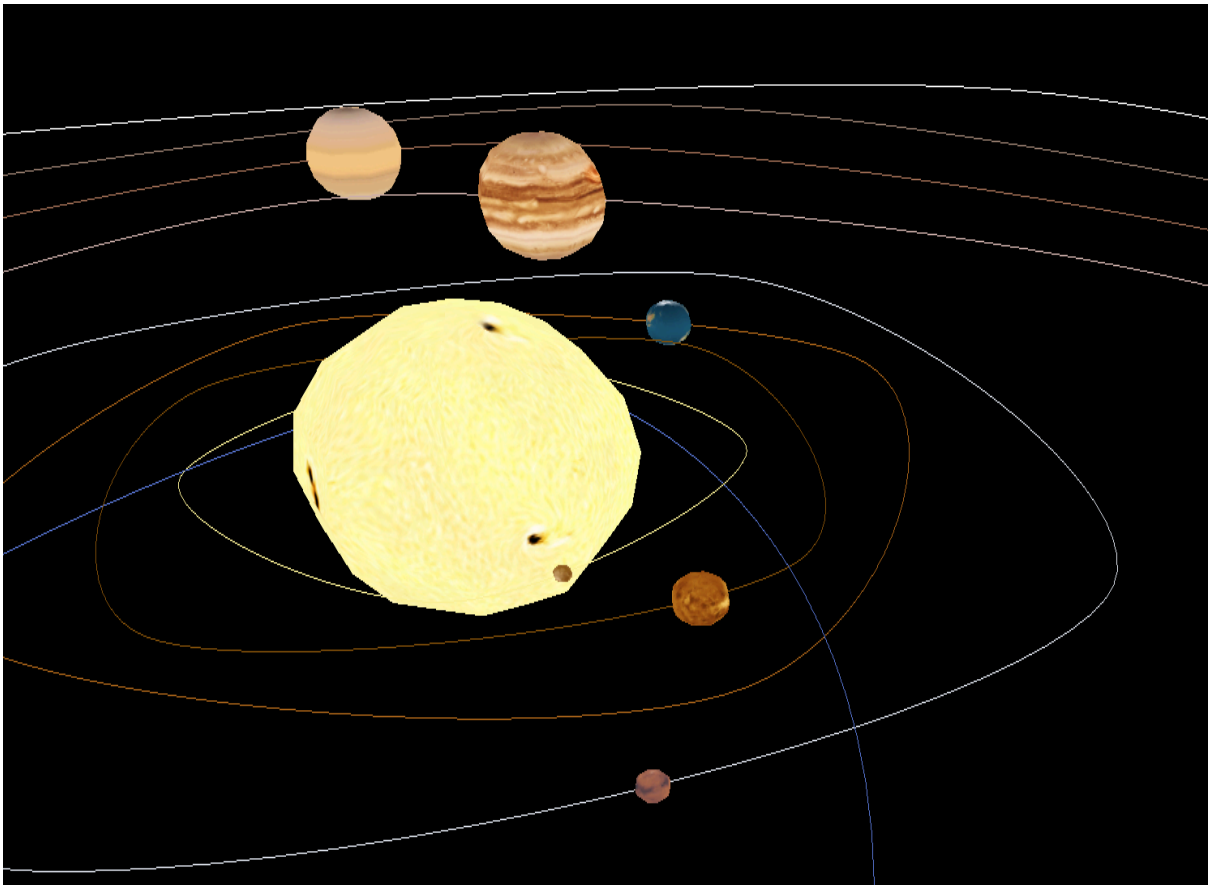


Figura 18 *- Demo Scene adaptada para esta fase*

Considerações Finais

Figura 19 *- Ponte Velha de Ponte de Lima*

Consideramos que o projeto teve sucesso em expor-nos aos princípios básicos do OpenGL e à programação de gráficos no geral. Apesar de terem havido alguns percalços e imperfeições na implementação, temos por opinião que foi um bom exercício para afinar as nossas habilidades na matéria.

Para futuro aperfeiçoamento, teríamos como foco:

- Resolver a anomalia verificada nas normais dos Bezier patches, que se expõem na renderização do Teapot.

- Desenvolver uma boa câmara First Person, já que foi algo que tentamos fazer na última fase e seria um extra muito útil, não só para melhor detecção de problemas como também para servir como um modo extra de câmara, mas, graças a frustrações com DevIL e outros problemas inerentes, acabamos por não ter tempo para explorar e aplicar esta componente no projeto.

- *Um overhaul* à aplicação de texturas ao cone, já que apesar de sabermos a teoria não conseguimos acabar a sua implementação a tempo da entrega.