Computação Gráfica: Fase 4 – Normals and Texture Coordinates

Afonso Faria,
André Ferreira,
Bruno Campos e
Ricardo Gomes

Universidade do Minho, Braga, Portugal
{a83920,a93211,a93307,a93785}@alunos.uminho.pt

Resumo Nesta fase foi-nos proposta a implementação de iluminação e mapeamento de texturas em modelos.

1 Introdução e Contextualização

O presente relatório diz respeito à quarta fase do projeto da Unidade Curricular de Computação Gráfica.

Nesta fase foi-nos proposta a implementação de coordenadas normais e coordenadas de texturas no generator para todos os modelos criados (cubo, plano, cone , esfera e superfícies de bezier). Foi também proposto a implementação de iluminação e de texturas.

Nas seguintes secções do relatório iremos descrever a abordagem tomada na resolução dos problemas e os obstáculos que surgiram durante a realização do projeto.

2 *Generator*

2.1 *Plano*

Coordenadas Normais: Para as coordenadas normais de um plano a implementação é instantânea pois todos os vértices têm normais N = (0,1,0).

Coordenadas de Textura: Para as coordenadas de textura de um plano dividimos a textura numa rede igual ao modelo do plano e aplicamos o resultado.

2.2 *Cubo*

Coordenadas Normais: Para as coordenadas normais de um cubo a implementação é quase igual à do plano, apenas repetindo o processo 6 vezes e alterando o vetor normal dependendo da face atual.

Coordenadas de Textura: Para as coordenadas de textura de um cubo repetimos o processo do plano para cada face do cubo.

2.3 *Esfera*

Coordenadas Normais: Para as coordenadas normais duma esfera apenas temos de considerar uma segunda esfera unitária (raio 1) e a partir desta obtemos facilmente o vetor normal de cada vértice de qualquer esfera.

Coordenadas de Textura: Para as coordenadas de textura da esfera passamos os ângulos de slices (entre 0^{o} e 360^{o}) e stacks (entre -90^{o} e 90^{o}) para um domínio de o a 1, utilizando depois essas coordenadas nas texturas para obter o mapeamento utilizado.

2.4 *Cone*

Coordenadas Normais: Para as coordenadas normais do cone usamos 3 pontos de cada face lateral para calcular 2 vetores pertencentes ao plano da face, calculando seguidamente o vetor normal ao plano e normalizando-o. No caso da base é evidente o uso do vetor V = (0,-1,0).

Coordenadas de Textura: Para as coordenadas de textura do cone utilizamos a mesma lógica que usamos na esfera para obter o mapeamento de texturas das faces laterais, obtidas através das slices e stacks. No caso da base utilizamos como lógica uma circunferência centrada no meio da textura com raio 0.5.

2.5 *Bezier Patches*

Coordenadas Normais: Para as coordenadas normais de um Bezier Patch utilizamos as derivadas no vértice de cada das componentes de tesselação (u e v) para obter 2 vetores, a partir destes calculamos o vetor normal, normalizando este para finalizar.

Coordenadas de Textura: Para as coordenadas de textura de um Bezier Patch utilizamos a tesselação u e v para obter uma rede no mapeamento da textura, sendo este processo similar ao do plano.

3 *Engine*

3.1 Iluminação

Para as luzes foi especulado adicionar um vetor de luzes, onde cado luz pode ter 3 tipos, Point,Directional ou Spotlight como esta ilustrado a seguir;

```
struct Light{
    string type;
    glm::vec3 pos;
    glm::vec3 dir;
    float cutoff;
};

struct Point{
    glm::vec3 pos;
};
struct Directional{
    glm::vec3 dir;
};
struct Spotlight{
    glm::vec3 pos;
    glm::vec3 dir;
    float cutoff;
};

auto Renderer::set_lights(Lights& l) -> Renderer& {
    int luzes = 0 ;
    GLfloat white[4] = { 1.0, 1.0, 1.0, 1.0 };
    for (auto const& luz : l) {
        glLightfv(GL_LIGHT0+luzes, GL_DIFFUSE, white);
        glLightfv(GL_LIGHT0+luzes, GL_SPECULAR, white);
        if (type == point){
            glLightfv(GL_LIGHT0,GL_POSITION,luz->pos);
        }
        if (type == direct){
            glLightfv(GL_LIGHT0,GL_SPOT_DIRECTION,luz->dirct);
        }
        if (type == spot){
            glLightfv(GL_LIGHT0,GL_POSITION,luz->pos);
            glLightfv(GL_LIGHT0,GL_SPOT_DIRECTION,luz->dirct);
            glLightfv(GL_LIGHT0,GL_SPOT_CUTOFF,luz->cutoff);
        }
    }
}
```

Assim seria possível inicilziar as luzes tendo em conta que seriam adicionadas os comandos glEnableClientState(GL_NORMAL_ARRAY) , glEnable(GL_LIGHTING), glEnable(GL_LIGHT0), glEnable(GL_LIGHT); ... e assim por diante.

Na inicialização do mundo, além de mandarem-se os pontos para a GPU mandou-se também as suas normais com
glGenBuffers(500,state::bind[1]);
glBindBuffer(GL_ARRAY_BUFFER, state::bind[1][i]);
glBufferData(GL_ARRAY_BUFFER,sizeof(float) * normals.size()*3, normals.data() ,GL_STATIC_DRAW);

E seria definido que todas as luzes seriam brancas, ou seja, com 1,1,1,1 como cor. E para se conseguir ver objetos que não tenham componente de luz usou-se uma luz ambiente para o modelo todo com
float amb[4] = 1.0f, 1.0f, 1.0f, 1.0f ;
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, amb);

3.2 Modelos

Cor Para definir as cores, ou seja, os materiais seria algo do género:

```
glMaterialfv(GL_FRONT, GL_AMBIENT, model->ambient);
glMaterialfv(GL_FRONT, GL_DIFFUSE, model->difuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, model->specular);
glMaterialf(GL_FRONT, GL_SHININESS, model->shininess);
glMaterialf(GL_FRONT, GL_EMISSION, model->emession);
```

Onde model->x , o x seria o ser valor guardado em cada model. Isto seria acrescentado na parte do render, no desenho de cada modelo.

Textura Para utilizar as texturas primeiramente foi necessário acrescentar os comandos para permitir coordenadas de textura glEnableClientState(GL_TEXTURE_COORD_ARRAY);glEnable(GL_TEXTURE_2D);
E na inicialização, tendo em conta que as coordenadas de textura estariam no model após a leitura do ficheiro .3d utilizaria estas linhas na inicialização do mundo

```
glGenBuffers(500,state::bind[2]);
glBindBuffer(GL_ARRAY_BUFFER, state::bind[2][i]);
glBufferData(GL_ARRAY_BUFFER,sizeof(float) * TEXT DO MODELO.size(), TEXT DO MODELO.data() ,GL_STATIC_DR
```

Era também esperado que ao modelo estivesse associado um valor idText durante a leitura do ficheiro do .3d com a função e que caso não houvesse textura fosse -1:

```
auto static  loadTexture(std::string s)noexcept -> void {

    unsigned int t,tw,th;
    unsigned char *texData;
    unsigned int texID;

    ilInit();
```

```
    ilEnable(IL_ORIGIN_SET);
    ilOriginFunc(IL_ORIGIN_LOWER_LEFT);
    ilGenImages(1,&t);
    ilBindImage(t);
    ilLoadImage((ILstring)s.c_str());
    tw = ilGetInteger(IL_IMAGE_WIDTH);
    th = ilGetInteger(IL_IMAGE_HEIGHT);
    ilConvertImage(IL_RGBA, IL_UNSIGNED_BYTE);
    texData = ilGetData();

    glGenTextures(1,&texID);

    glBindTexture(GL_TEXTURE_2D,texID);
    glTexParameteri(GL_TEXTURE_2D,    GL_TEXTURE_WRAP_S,        GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D,    GL_TEXTURE_WRAP_T,        GL_REPEAT);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D,    GL_TEXTURE_MAG_FILTER, GL_LINEAR);

    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, tw, th, 0, GL_RGBA, GL_UNSIGNED_BYTE, texData);
    //glGenerateMipmap(GL_TEXTURE_2D);


    glBindTexture(GL_TEXTURE_2D, 0);



    state::idText = texID;


}
```

E assim, durante a fase do render, ao desenhar um modelo bastaria:

```
if(modelo->tex != -1){
    glBindBuffer(GL_ARRAY_BUFFER, state::bind[2][iii]);
    glTexCoordPointer(2,GL_FLOAT,0,0);

}
```

Inacabado Como não conseguimos implementar o engine completo seguem-se exemplos de mapeamento de texturas e iluminação testados("hard-coded"):

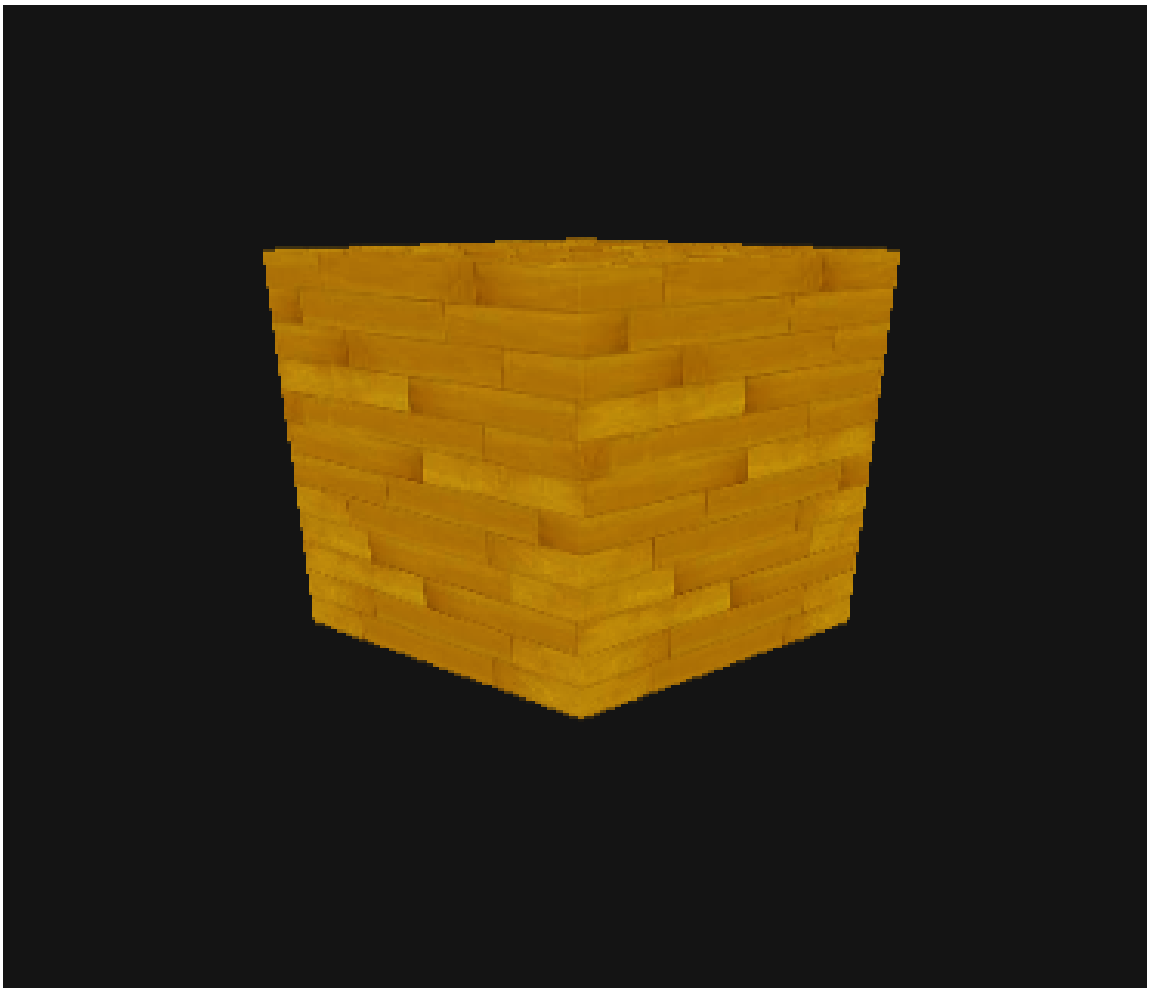

Figura 1. Exemplo 1: Mapeamento de um Cubo

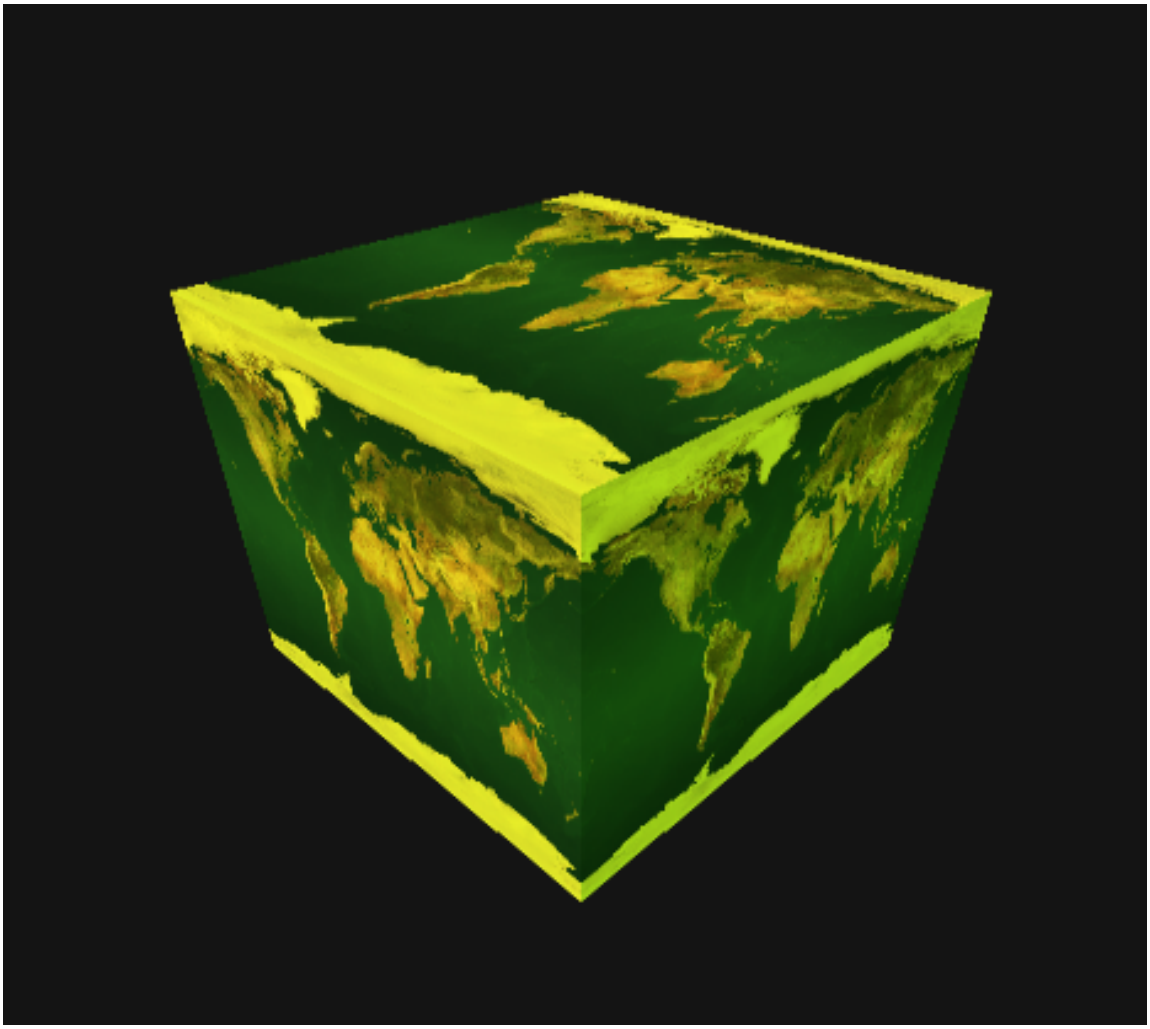

Figura 2. Exemplo 2: Mapeamento de um Cubo com textura da terra

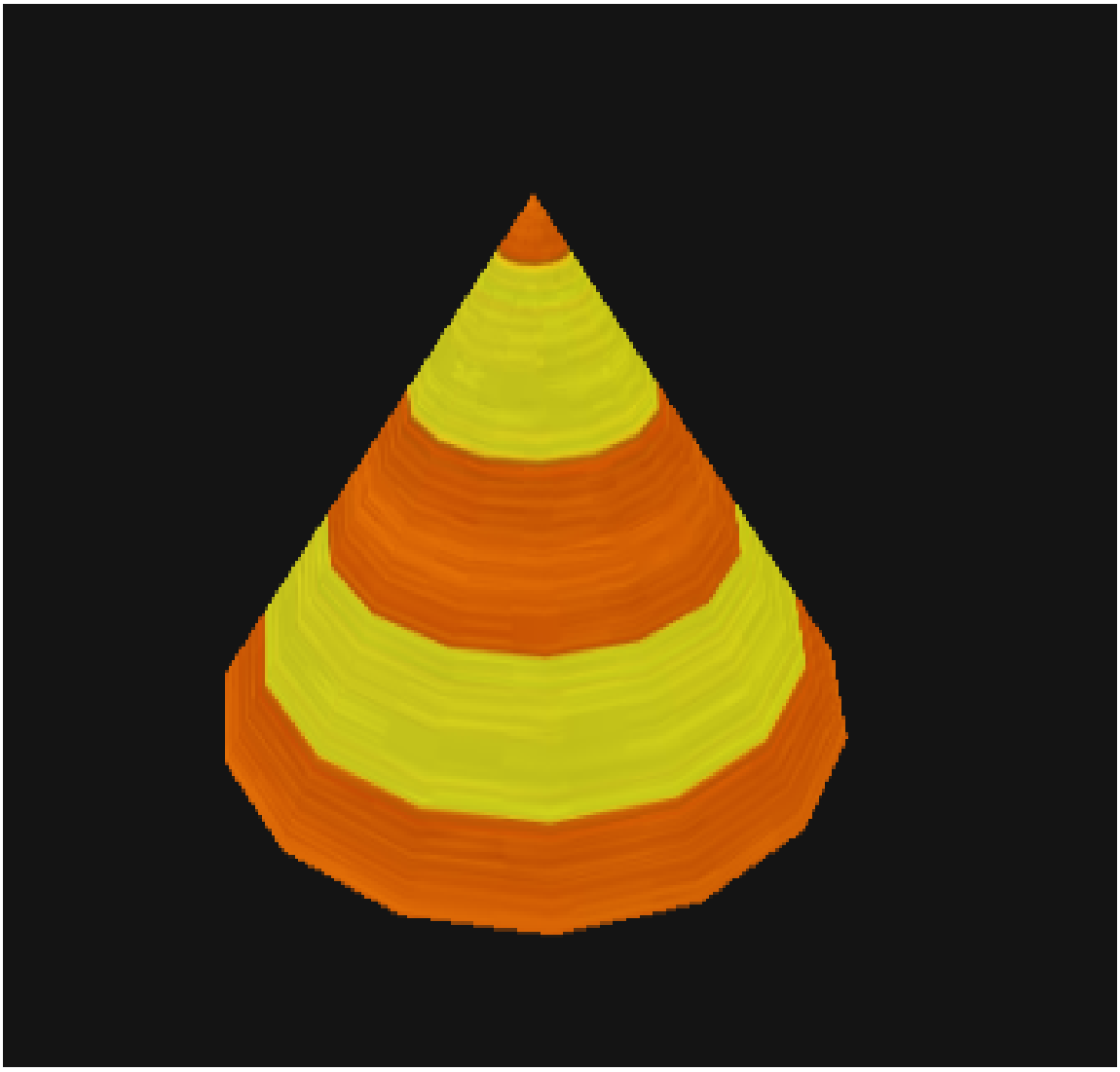

Figura 3. Exemplo 3: Mapeamento de um Cone

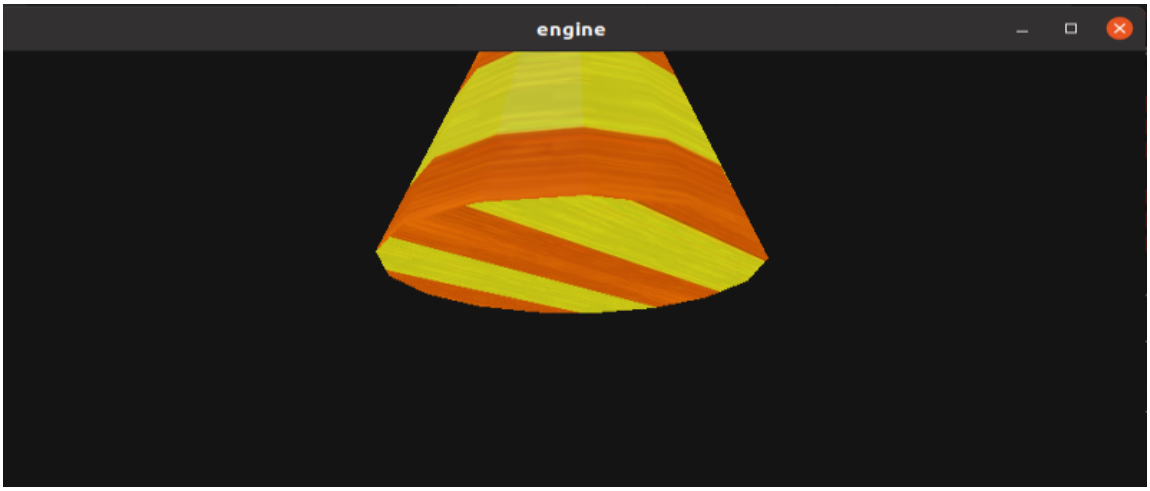

Figura 4. Exemplo 4: Mapeamento da base de um Cone

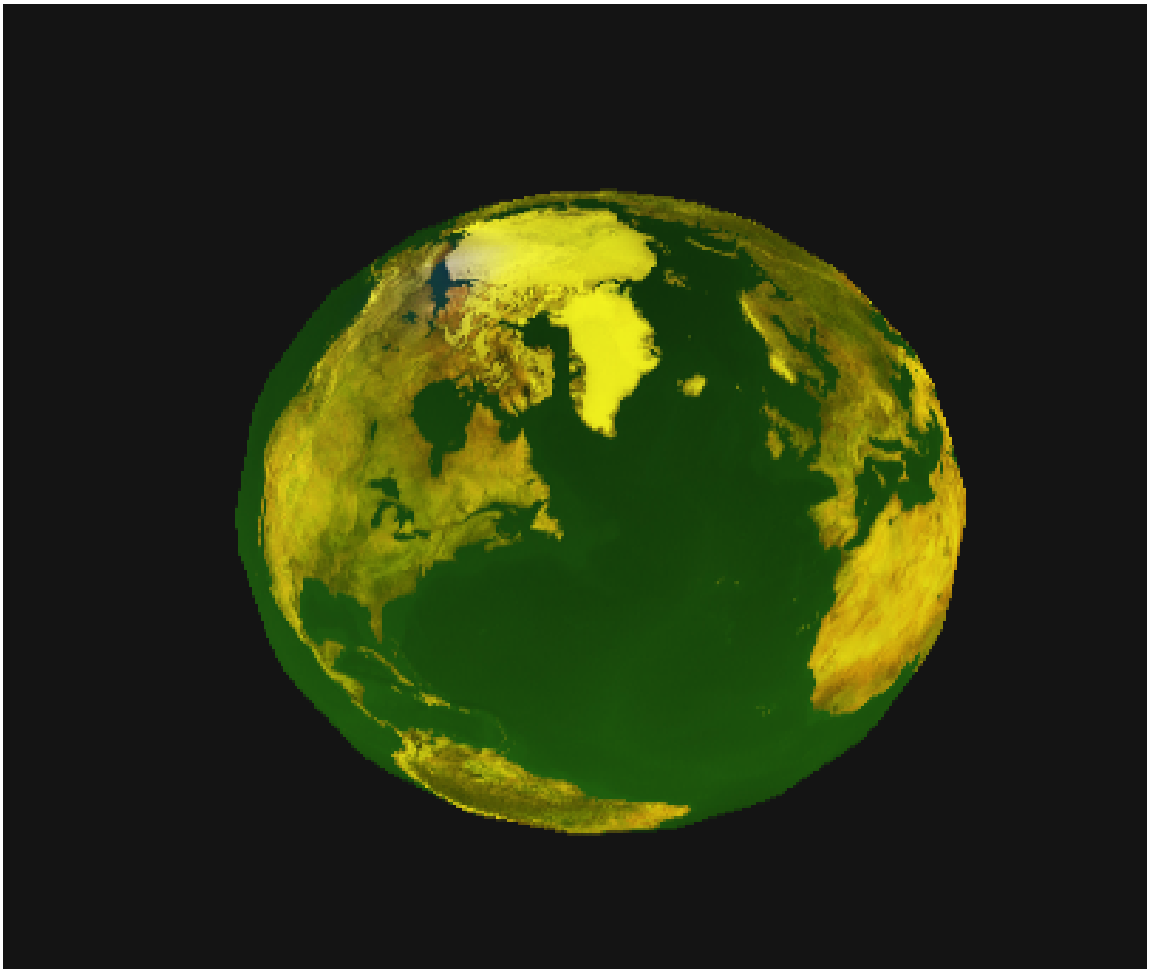

Figura 5. Exemplo 5: Mapeamento de uma Esfera

4 Resultados

Por falta de tempo esta secção encontra-se vazia.

5 Conclusão

O projeto e todos os testes foram desenvolvidos em Linux. Várias funcionalidades implementadas tiram partido de bibliotecas standard de C++20. Também recorremos às bibliotecas:

- **{fmt}**, que facilita operações de *input/output* com formatação;
- **glm**, que oferece tipos de dados e funções de matemática para OpenGL;
- **rapidxml**, para efetuar a leitura de ficheiros XML;
- **result**, para efetuar *value-based error handling*;
- **speedlog**, para imprimir informação de *logging*;
- **tinyobjloader**, para efetuar a leitura de ficheiros Wavefront .obj.

Apesar de não termos conseguido chegar aos resultados esperados, por não termos lido com sucesso e alterado as estruturas como tínhamos mencionado no relatório, acreditamos que conseguimos calcular corretamente as coordenadas de textura e de normais, tendo conseguido apresentar testes hard-coded de alguns modelos.