Computação Gráfica - 4.ª Fase

Dinis Lourenço Gomes
a87993@alunos.uminho.pt

Rui Miguel Gonçalves Dias
a89480@alunos.uminho.pt

Universidade do Minho - Departamento de Informática

1 Introdução

Nesta fase, o **Generator** deverá ter a capacidade de gerar coordenadas de textura e normais para cada vértice.

No **Engine** serão activadas as funcionalidades de luz e textura, e deverá não só ler, mas também aplicar as coordenadas das normais e das texturas dos respectivos ficheiros **Models**.

É então necessário modificar não só as funções que tratam do *parsing* dos ficheiros *XML*, como também alterar e criar estruturas de modo a implementar estas novas funcionalidades.

2 Implementação

2.1 Estruturas Utilizadas

- **Light**

  Esta estrutura é utilizada para guardar as componentes da fonte de luz do modelo. Dependendo do tipo de luz a estrutura será preenchida com diferentes dados. Por opção do grupo decidiu-se fazer apenas uma classe em vez de separar em múltiplas classes.

```
class Light {
public:
    // POINT | DIRECTIONAL | SPOT -> Spotlight
    string type;

    // Color Ambience
    float ambience[4];

    // Color Defuse
    float defuse[4];

    // POINT -> Coord X, Coord Y, Coord Z, Point <-> W = 0
    float posicao[4];

    // DIRECIONAL -> Coord X, Coord Y, Coord Z, Vector <-> W = 1
    float direcao[4];

    // SPOT -> Utiliza posicao e direcao.
};
```

- **Model**
  A esta estrutura previamente desenvolvida foi adicionado uma *string* com o nome do ficheiro das texturas, dois vectores de *floats* que correspondem às normais dos pontos e aos pontos de textura, o número total de vértices e a informação da respectiva textura(se esta existir). A cor foi alterada e em vez de um só array terá agora três arrays diferentes um para a cor ambiente, outro para a cor difusa e um para a cor emitida, se o objecto não emite luz este array corresponde à cor preta.

```
class Model{
    public:
        string filename; //nome do file de pontos associado.
        string texture; //nome do file de texturas associado.
        float amb[4];
        float dif[4];
        float emis[4];
        int vertexCount;
        vector< float > coordenadas;
        vector <float> normais;
        vector <float> texCoords;
        GLuint texID;

        // Default Constructor
        Model::Model(string f, string text, Color c, bool e, vector<float>
    coord, vector<float> normies, vector<float> texs, GLuint id) {
            filename = f;
            texture = text;
            amb[0] = c.r * 0.15;
            amb[1] = c.g * 0.15;
            amb[2] = c.b * 0.15;
            amb[3] = c.a;
            dif[0] = c.r * 0.85;
            dif[1] = c.g * 0.85;
            dif[2] = c.b * 0.85;
            dif[3] = c.a;
            if (e) {
                emis[0] = c.r;
                emis[1] = c.g;
                emis[2] = c.b;
            }
            else {
                emis[0] = 0;
                emis[1] = 0;
                emis[2] = 0;
            }
            emis[3] = c.a;
            vertexCount = coord.size() / 3;
            coordenadas = coord;
            normais = normies;
            texCoords = texs;
            texID = id;
        }

        void Model::setImage(GLuint id) {
            texID = id;
        }

};
```

2.2 Generator

2.2.1 Cálculo dos Vértices

- **Torus**

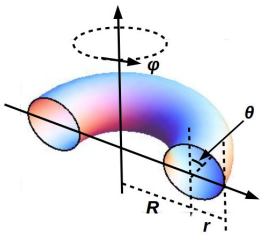

Figure 1: Torus

$$x = R * sin(phi) + r * sin(phi) * cos(theta)$$
$$y = r * cos(theta)$$
$$z = R * cos(phi) + r * cos(phi) * cos(theta)$$

2.2.2 Cálculo das Normais

- **Plano**

  Para as normais do plano, sendo este um plano é elementar calculá-las sendo apenas necessário apresentar um vector perpendicular ao plano.

- **Caixa**

  Análogo ao plano é apenas aplicado a cada face da caixa.

- **Esfera**

  Para a esfera, as normais foram calculadas utilizando a função *normalize* utilizado nas fichas práticas.

- ***Patch***

  Através dos pontos previamente calculados são calculadas as normais utilizando a seguinte fórmula:

  $$\vec{n} = \vec{v} \times \vec{u}$$

- **Cone**

  Para a face lateral do cone o raciocínio é análogo as *Patches* pelo que para cada triângulo é calculado uma normal.

  A base do cone é um plano pelo que o raciocínio é análogo.

- **Torus**

  $$x = sin(phi) * cos(theta)$$
  $$y = cos(theta)$$
  $$z = cos(phi) * cos(theta)$$

2.2.3 Cálculo das Texturas

Para a representação de texturas, as imagens das texturas são carregadas em formato de repetição reflectida.

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_MIRRORED_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_MIRRORED_REPEAT);
```

Os tópicos seguintes representam como os pontos de textura são calculados para cada um dos objectos:

- **Plano**

Figure 2: Plane

- **Caixa**

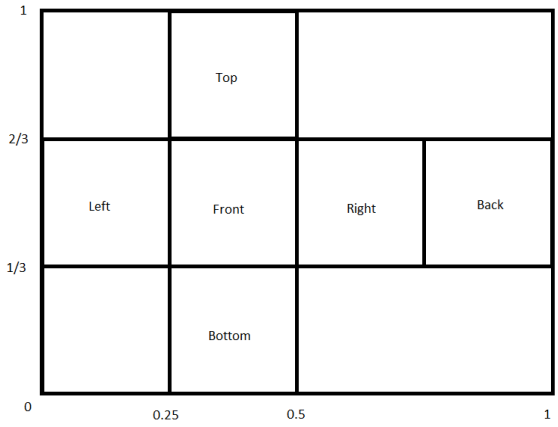

Figure 3: Caixa

- **Cone**

  Nota: O circulo da esquerda será a textura da face lateral enquanto o da direita será a textura da base.

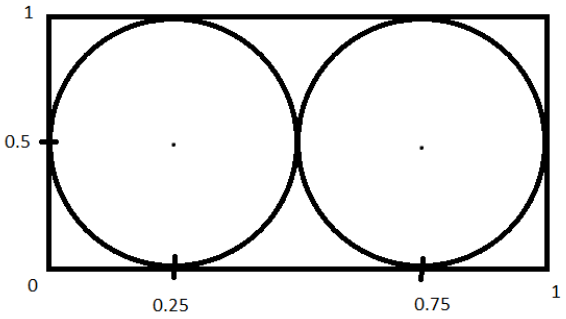

Figure 4: Cone

- **Esfera**

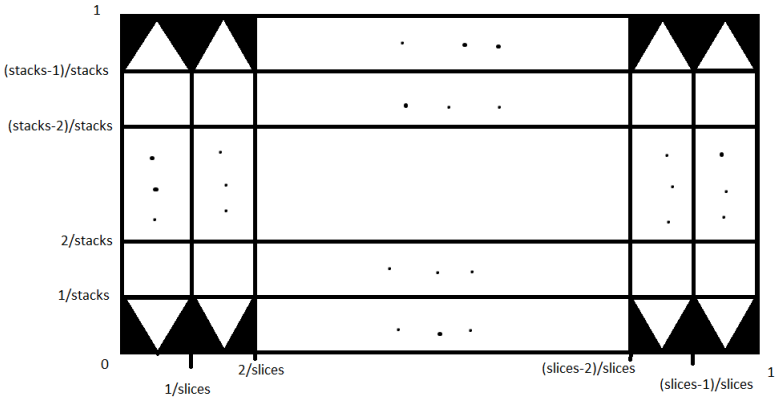

Figure 5: Esfera

- **Patches**

  Nota: A textura repete-se para cada patch.

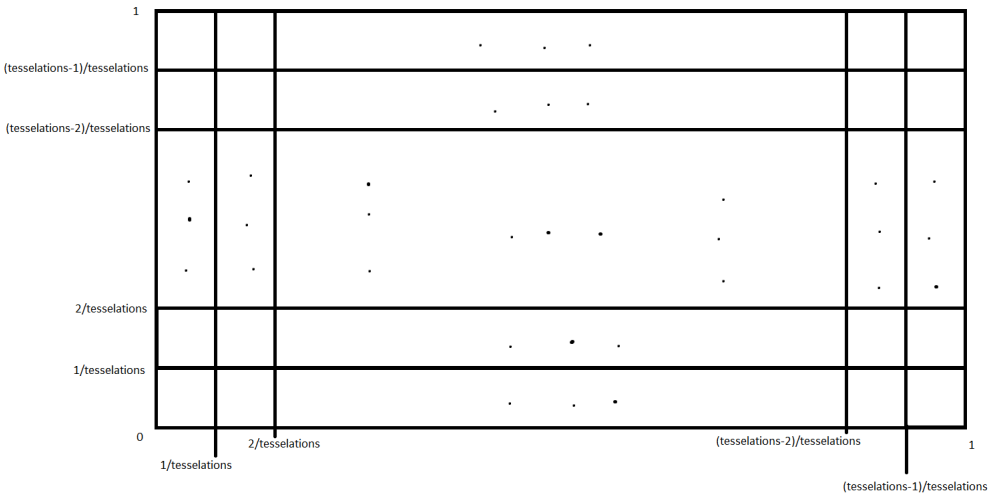

Figure 6: Patches

- **Torus**

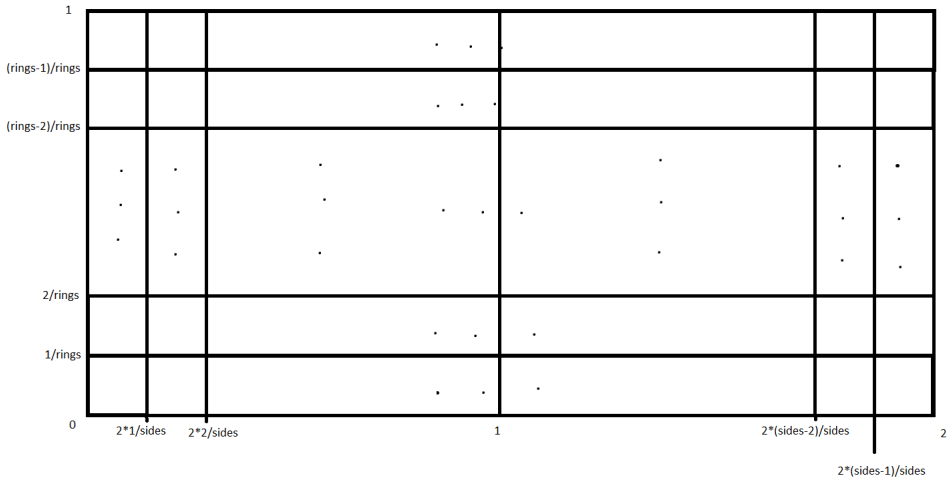

Figure 7: Torus

2.3 Engine

2.3.1 Buffers

Nesta fase, tal como na anterior, é requerido o uso de VBOs porém em vez de apenas um buffer por modelo para os vértices, é necessário três buffers por modelo, um para os vértices, um para as normais e outro para as coordenadas de textura.

A utilização destes buffers tem como requisito a ativação dos respectivos *ClientStates* :

```
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_NORMAL_ARRAY);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
```

São então criados três arrays de tamanho igual ao número de modelos na cena, um para cada Client, sendo que a n-ésima posição dos três arrays corresponde ao três buffers do n-ésimo modelo.

Cada buffer é construído através do respectivo array previamente criado e populado na classe *Model*, numa função chamada antes de começar a renderização da cena.

2.3.2 Light

O grupo desenvolveu uma função que faz *parsing* à secção de **light** com os vários grupos de luz que a cena irá apresentar.

De acordo com o tipo de luz pretendido o grupo de luz será presenteado com diferentes tipos de argumentos, ou seja:

- Point

  Este tipo de iluminação irá apenas receber a posição da fonte de luz e a sua orientação.

- Diffuse

  Este tipo de iluminação irá apenas receber a orientação da luz.

- Spotlight

  Este tipo de iluminação irá receber a posição da fonte de luz e a sua orientação.

2.3.3 Cor

Em adição ao trabalho anteriormente realizado para a cor dos modelos, foi acrescentada a variável que representa a transparência. Com a junção desta nova informação e a previamente estabelecida (variáveis RGB), cria-se os arrays que irão então ser usados para definir *Ambience*, *Diffuse* e *Emission* de cada modelo.

Para esta alteração foi também necessário adicionar as seguintes instruções:

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

É importante salientar que modelos com Transparência diferente de 1 necessitam de ser desenhados por fim.

2.3.4 Câmera

Em prol do movimento da câmera estabelecido em fases passadas, possibilidade de alterar o ponto de foco, foi implementada a renderização de uma esfera no ponto de foco para uma mais fácil navegação pela cena. O tamanho desta esfera depende da distância da posição da câmera e o ponto de foco.

3 Scenes

3.1 Solar System

Nesta fase os anéis de Saturno passaram a ser semi-transparentes e, tal como referido anteriormente, são desenhados em último lugar. Foi também adicionada uma skybox com a textura da via láctea.

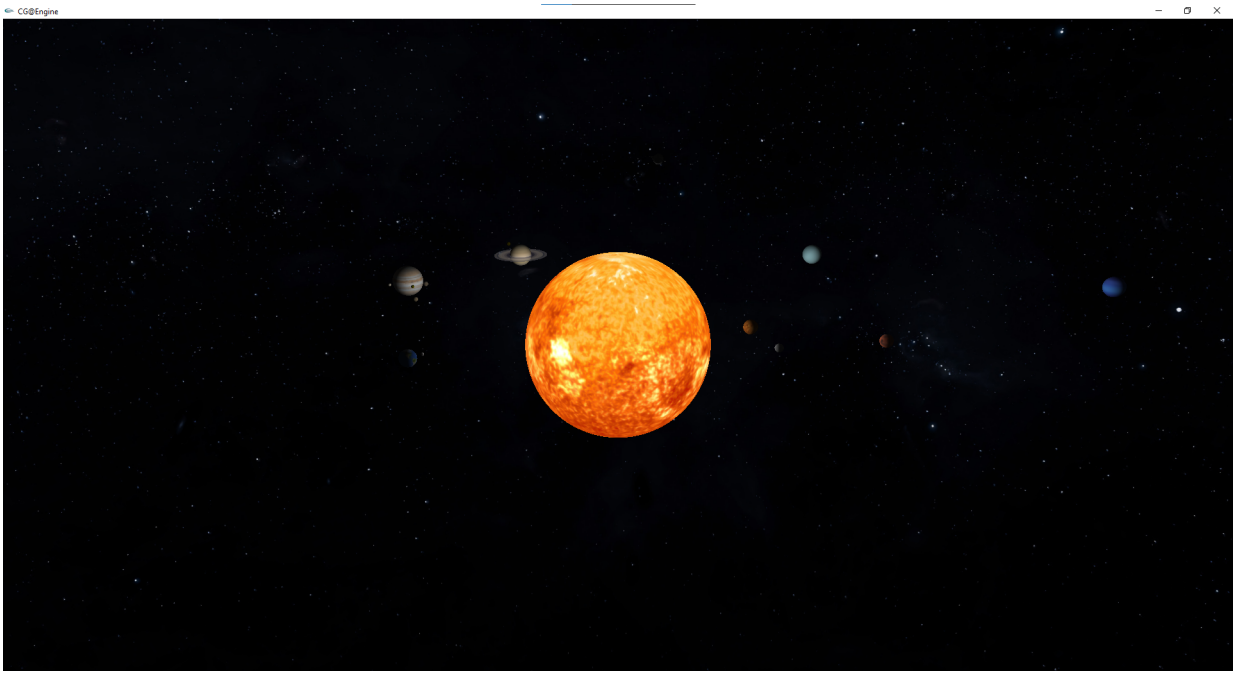

Figure 8: Sistema Solar

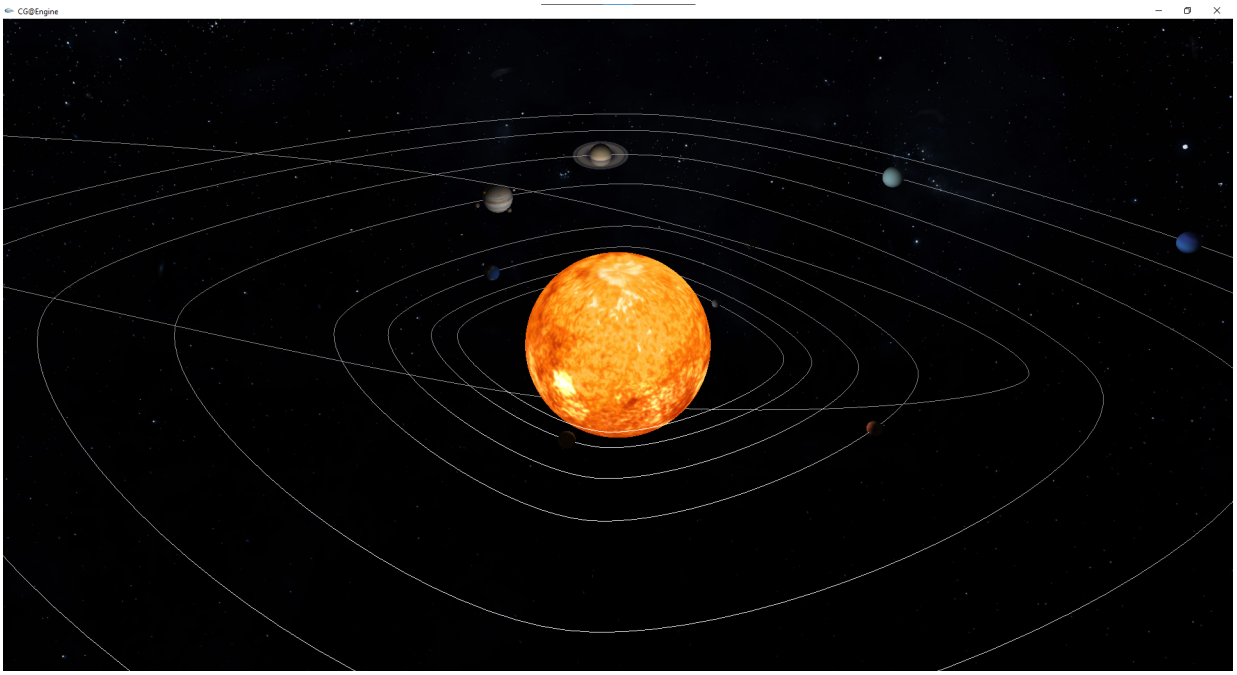

Figure 9: Sistema Solar com trajectórias

4 Conclusão

Com este trabalho prático podemos aplicar os conhecimentos leccionados nas aulas da unidade curricular de computação gráfica permitindo assim aprofundar os nossos conhecimentos de *openGl*.

Como trabalho futuro gostaríamos de implementar *View Frustum Culling* e uma câmara em primeira pessoa.