Universidade do Minho

Mestrado Integrado em Engenharia Informática

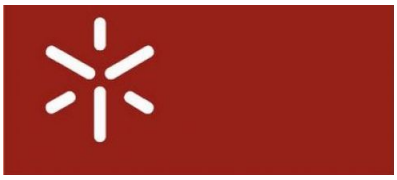
Computação Gráfica

Normais e Texturas

Grupo:
Etienne Costa A76089
Joana Cruz A76270
Rafael Alves A72629
Maurício Salgado A71407

Docente:
António Ramires

18 de Maio de 2019



Conteúdo

1 Introdução

O relatório apresentado diz respeito à quarta fase do projeto proposto no âmbito da unidade curricular de Computação Gráfica. Nesta última fase, o Gerador passou a gerar normais e coordenadas de texturas para cada um dos pontos gerados. Para além disso, o Engine passou a suportar funcionalidades relativas à iluminação do cenário (utilizando as normais geradas anteriormente) e à aplicação de texturas aos modelos.

2 Gerador

Até à fase anterior, o Gerador calculava as coordenadas dos pontos constituintes das primitivas. Nesta fase, o Gerador foi alterado de maneira a que, para além de gerar os pontos das primitivas, gerar também as coordenadas das normais e as coordenadas da textura de cada ponto. Ou seja, para cada primitiva (plano, caixa, esfera, e teapot) são geradas as coordenadas dos pontos que a constituem, como também as coordenadas da normal e textura de cada um desses pontos. As normais dos pontos são geradas para que, ao introduzir-se uma fonte de luz, o OpenGL possa desenhar a sombra da primitiva. 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.

2.1 Normais

Cada uma das primitivas gráficas passou a ter o conjunto das normais dos seus vértices. Assim sendo, o cálculo das normais para cada uma das primitivas gráficas foi inserido no ficheiro gerado .3d.

2.1.1 Plano

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.

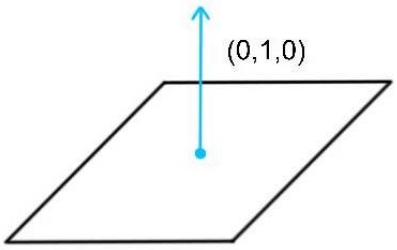


Figura 1: Normal de um plano

2.1.2 Caixa

A geração dos vértices da caixa 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).

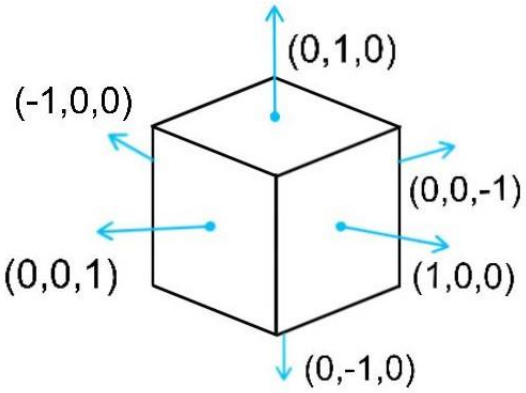


Figura 2: Normais de um cubo

2.1.3 Esfera

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

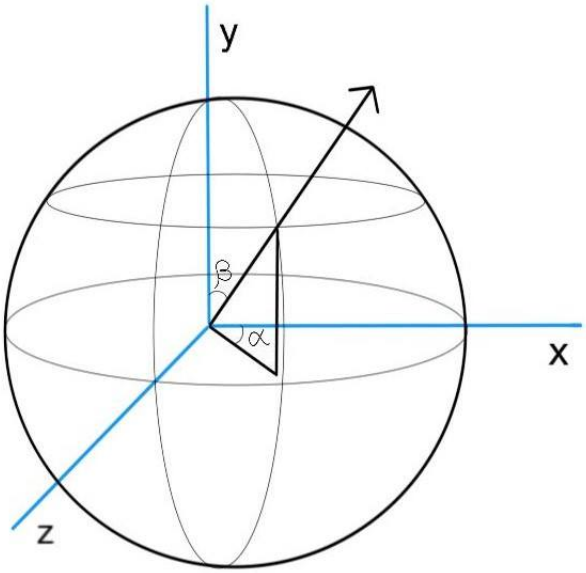


Figura 3: Normal de um ponto de uma esfera

2.1.4 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.

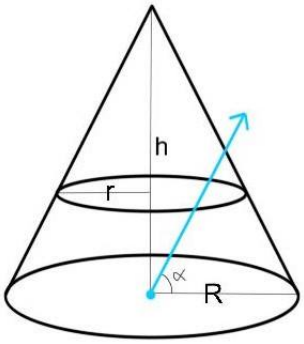


Figura 4: Normal de um ponto da superfície lateral de um cone

2.1.5 Bezier Patches

Os pontos das normais dos Bezier Patches foram os mesmo usados das coordenadas para o desenho dos pontos.

2.2 Texturas

Cada uma das primitivas gráficas passou a guardar as coordenadas correspondentes às texturas que poderão ser posteriormente aplicadas a cada uma delas. Para isso, apenas foi necessrátio, para os pontos de cada primitiva, associar as coordenadas dos pontos correspondentes de uma textura.

2.2.1 Plano

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 Esfera

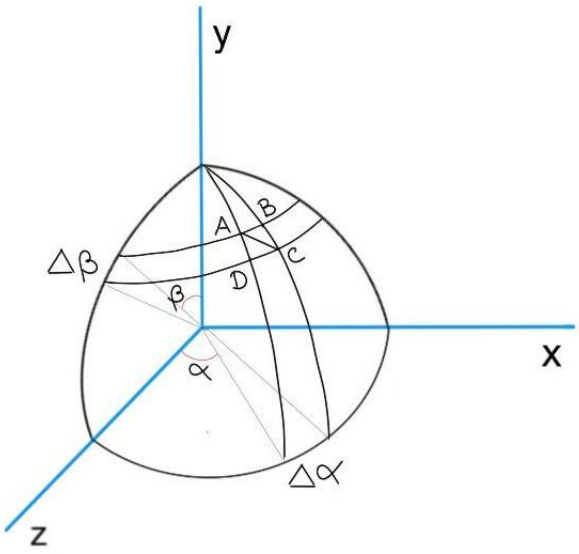


Figura 5: Ilustração de uma intersecção de 4 pontos nums esfera

Relativamente à esfera, os pontos correspondentes da textura são dados por:

Para cada stack i
    Para cada slice j

$tAx = j/slices$
$tAy = i/stacks$

$tBx = (j+1)/slices;$
$tBy = i/stacks$

$tCx = (j+1)/slices$
$tCy = (i+1)/stacks$

$tDx = j/slices;$
$tDy = (i+1)/stacks$

3 Engine

Relativamente à fase anterior, o programa Engine agora está 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. O 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 no nosso programa pode ser um de dois tipos: point e directional. 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.

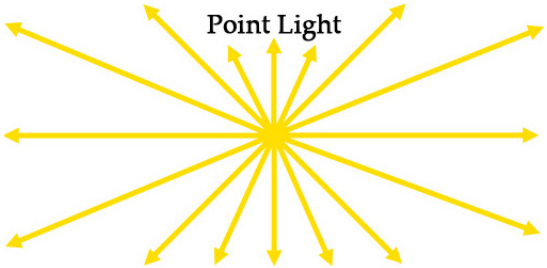


Figura 6:

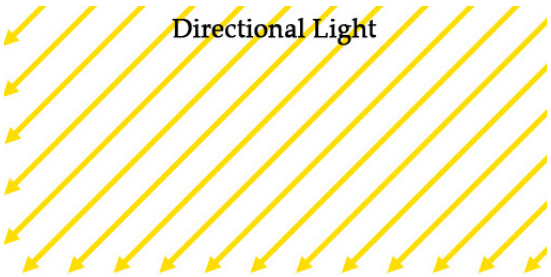


Figura 7:

No ficheiro XML este poderá receber:

```
<lights>
    < light type="point" X=0 Y=0 Z=0 />
    < light type= "directional" X=1 Y=1 Z=1 />
</lights>
```

Além destas propriedades, também é possível definir as cores das componentes ambiente, difusa e especular de luz. A componente ambiente representa uma

fração desta luz que foi recursivamente fragmentada pelo meio, em sucessivas reflexões, sendo praticamente impossível determinar a sua direção inicial. As luzes de fundo e as sombras têm a sua componente de luz ambiente maior que as restantes, exatamente pelo facto de apesar de nenhuma fonte de luz incidir diretamente nestas zonas, estas ainda recebem alguma luz que foi refletida indefinidamente até chegar ao objeto. De forma geral, a componente ambiente é a cor de um objeto quando este se encontra à sombra, mesmo sem que este tenha uma fonte de luz diretamente direcionada para ele. A componente difusa, representa a iluminação geral do objeto, quando este se encontra na direção de uma fonte de luz especifica e bem definida. A luz difusa pode assim ser descrita como uma fonte de luz que vem apenas de uma única direção e na qual a cor da luz define a cor do objeto sobre o qual esta incide. Quando a luz embate numa superfície, esta espalha-se igualmente em todos os sentidos. A iluminação especular, ou specular highlight, representa o "brilho"de um objeto, causado pela reflexão de uma determinada fonte de luz sobre si. Tal como na vida real, esta componente da luz apresenta a cor branca. Para isso, acrescentam-se à tag model

- Os elementos ambientX, ambientY e ambientZ, para a luz ambiente
- Os elementos diffuseX, diffuseY e diffuseZ, para a luz difusa
- Os elementos diffuseANDambientX, diffuseANDambientY e diffuseANDambientZ, para a luz ambiente e difusa
- Os elementos specularX, specularY e specularZ, para a luz especular
- Os elementos emissionX, emissionY e emissionZ, para a luz emissiva

Exemplos de XML:

```
<model file= "primitive.3d" diffuseX=0.5 diffuseY=0.5 diffuseZ=0.5 />
<model file= "primitive.3d" ambientX=0.5 ambientY=0.5 ambientZ=0.5 />
<model file= "primitive.3d" specularX=0.5 specularY=0.5 specularZ=0.5 />
<model file= "primitive.3d" emissionX=0.5 emissionY=0.5 emissionZ=0.5 />
```

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=100 />
```

Com esta nova versão do Engine, é possível aplicar texturas a primitivas. Para isso, utilizam-se as coordenadas de textura geradas pelo Generator.

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

3.1 Alterações nas Estruturas de Dados

Nesta secção iremos abordar as alterações que fizemos às classes já existentes, e as novas classes de apoio criadas.

3.1.1 Vertex

Esta classe representa um ponto num referencial a três dimensões, com as coordenadas x, y e z, o que se torna bastante útil para a representação dos vértices utilizados posteriormente para o desenho dos triângulos. que elaboram as figuras primitivas.

```
class Vertex{
    public:
        float x;
        float y;
        float z;
        Vertex();
        Vertex(float xx, float yy, float zz);
        ~Vertex();
    };
```

3.1.2 Light

Criou-se a classe Light para guardar a informação relativa às origens de luz. Esta classe possui duas, uma para identificar qual o tipo de luz, e outra que guarda a posição ou direção da luz.

```
class Light{
    public:
        Vertex lightP;
        string lightType;
        Light(void);
        Light(Vertex ligthPos, string light);
        ~Light();
        void draw();
    };
```

3.1.3 Material

A classe Material foi criada com o intuito de armazenar a informação obtida no ficheiro XML relativa aos materiais das figuras primitivas. Os materiais, por sua vez, podem possuir as seguintes propriedades:

- Diffuse

- Ambient
- Diffuse and Ambient
- Specular
- Emission
- Shininess

```
class Material{
    public:
        float diffuse[4], ambient[4], diffuseANDambient[4],
        specular[4], emission[4], shininess;
        Material(void);
        Material(Vertex diff, Vertex amb, Vertex diffAm,
        Vertex spec, Vertex emiss, float shi, bool texture);
        ~Material();
        void draw();
    };
```

O método draw ativa o material de acordo com os pontos obtidos do ficheiro XML.

3.1.4 Model

A classe Model já existia anteriormente, com o objetivo de guardar os dados de uma figura retirados do ficheiro XML e sofreu diversas alterações. Agora esta classe possui três vetores de Vertex(coordenadas x, y, z de um vértice) que correspondem às coordenadas dos vértices das primitivas, assim como as coordenadas das normais e das texturas.

- O caminho do ficheiro da textura que ser quer aplicar
- O material que constitui o modelo
- A textura
- As coordenadas dos vértices das primitivas, assim como as coordenadas das normais e das texturas.
- Os buffers correspondes aos vértices, normais e texturas

```
class Model{
    public:
        string fileTexture;
```

```
        vector<Vertex> vertexes;
        vector<Vertex> normals;
        vector<Vertex> textures;
        GLuint buffer[3];
        GLuint texture;
        Vertex color;
        Material material;
        Model();
        Model(string path);
        ~Model();
        void fillBuffers();
        void draw();
        void prepareTexture(string s);
    };
```

Criámos o método **prepareTexture(textura)** de modo a que se possa carregar a textura a partir de um ficheiro, utilizando os métodos presentes na biblioteca IL.h, e posteriormente ativa-se o buffer das texturas. Quando se faz a leitura e processamento do ficheiro XML, preenche-se o buffer com os pontos da figura, os pontos das normais e os pontos da textura. Desta forma, surgiu o método **fillBuffers()** de modo a carregar toda a informação contida em cada posição dos arrays dos vértices, normais e texturas para os buffers (posição 0 contém os pontos, 1 contém as normais e 2 contém as texturas). O método **draw()** deve ser invocado no Engine permitindo um desenho mais rápido e eficiente das figuras.

3.1.5 Group

Esta é a classe que guarda os dados retirados de um grupo do ficheiro XML. Nesta estrutura é possível armazenar todas as informações que estão associadas a um determinado grupo, as respetivas transformações geométricas se existirem, os modelos associados a esse grupo, assim como os grupos filhos contidos.

```
class Group{
    public:
        Vertex rotation;
        float rotationAngle;
        Vertex translation;
        Vertex scale;
        vector<Model> models;
        vector<Group> subGroups;
        vector<Vertex> orbitPoints;
        float translationTime;
        float rotationTime;
        Group(void);
```

```
        Group(Vertex rot, float rotAng, Vertex trans, Vertex scle,
        vector<Model> modls, vector<Group> subs, vector<Vertex> orbitPoint,
        float timeTranslation, float timeRotation);
        ~Group();
    };
```

3.1.6 Scene

Esta classe armazena todas as luzes e os grupos pelo qual o cenário é constituído.

```
class Scene{
    public:
        vector<Group> groups;
        vector<Light> lights;
        string fileScene;
        Scene(void);
        Scene(vector<Group> nGroups, vector<Light> nLights, string file);
        ~Scene();
    };
```

4 Exemplo de Execução

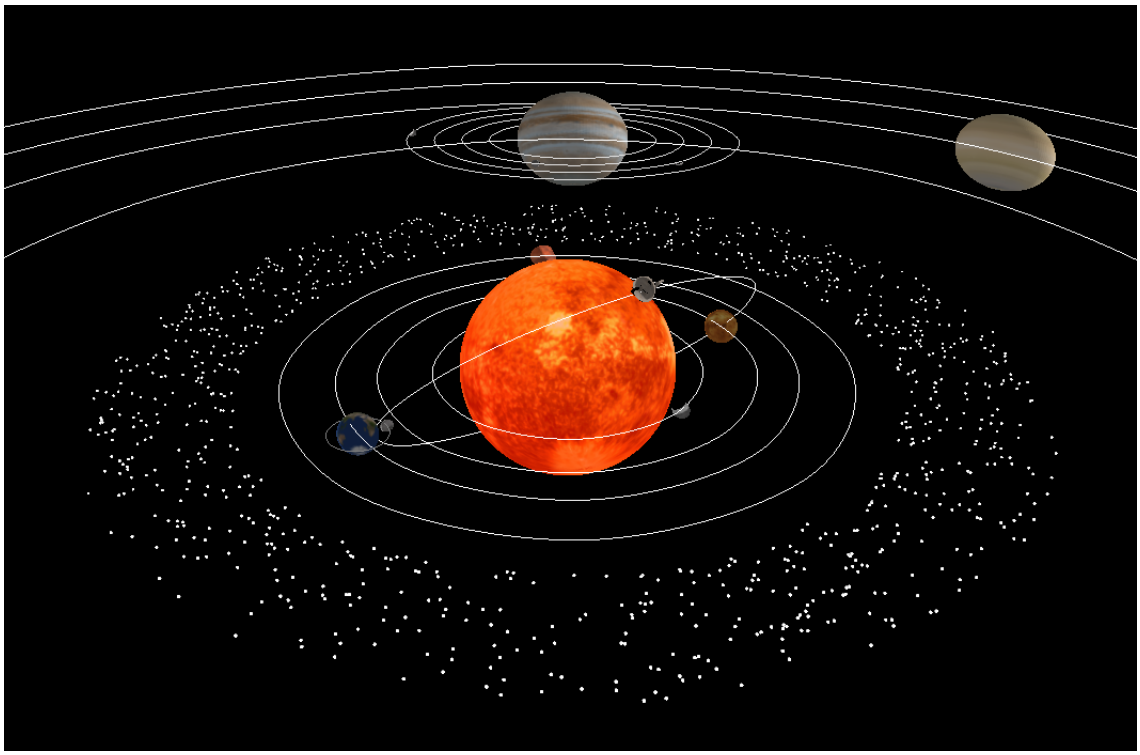


Figura 8: Sistema solar

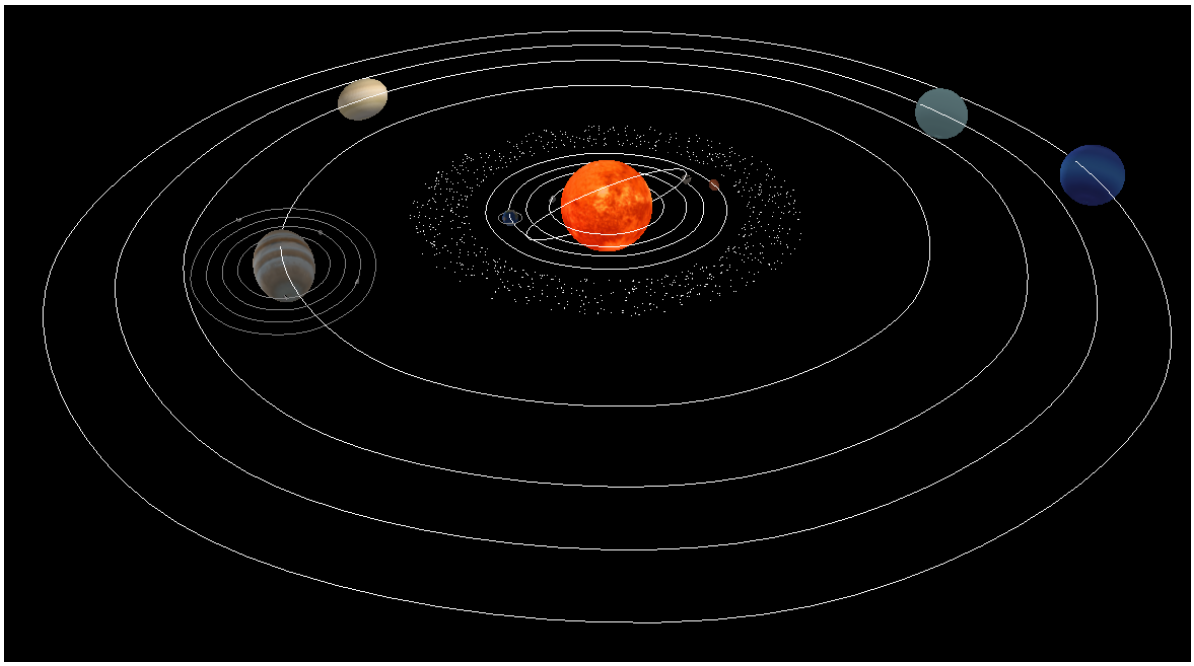


Figura 9: Sistema solar

5 Conclusão

O nosso programa inicialmente apenas erá capaz de gerar os pontos de várias primitivas e, com o Engine, ler um ficheiro XML com a informação sobre os modelos(e respetivos ficheiros com os vértices gerados anteriormente), e desenhá-los. 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 triângulos dos mesmos. Posteriormente, foram implementadas funcionalidades relativas às transformações geométricas, definidas por tags especifícas do ficheiro XML do cenário. Neste fase também implementação o desenho com recurso a VBO's. Na fase seguinte, as rotações e as translações foram melhoradas. No caso da translação, foi adicionada a capacidade de introduzir pontos 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. Assim como, foi implementado o desenho de modelos a partir de patches de Bezier. Esta última fase consistiu na geração adicional das normais dos modelos e de coordenadas para as texturas. Além disso, foi possíveç a implementação de vários tipos de luzes para a iluminação da cena e a aplicação de texturas aos modelos. Nesta última fase do trabalho, tivemos diversas dificuldades para conseguir completar o trabalho no prazo de entrega, dado que submetemos no dia de entrega um trabalho pouco funcional, e entretanto fizemos os possíveis para melhorar o trabalho, assim como realizar o relatório. Com a finalização desta fase, dá-se por terminado o projeto. No entanto, ainda é possível continuar o desenvolvimento deste, pois houveram diversas funcionalidades extra, e algumas primitivas que gostaríamos de ter implementado.