Universidade do Minho

Computação gráfica

Departamento de Informática

Fase 4

Grupo 40:
João Abreu — A84802
Tiago Magalhães — A84485
Hugo Matias — A85370
Sérgio Gomes — A67645

31 de Maio de 2020

Conteúdo

1. Arquitetura do código

1.1 Aplicações

Devido ao facto de este trabalho ser no seguimento dos anteriores, tendo sido feitas diversas alterações necessárias a cada umas das seguintes aplicações a serem cumpridos os novos requisitos.

1.1.1 Generator

Indo ao encontro do que foi feito nas fases anteriores, o generator destina-se a gerar vários pontos constituintes das várias primitivas gráficas conforme os parâmetros fornecidos. Nesta fase, para além das primitivas gráficas anteriores e o método de construção de modelos com base nas curvas de Bézier, foram introduzidos cálculos de vetores normais e de textura para ser possível a implementação de **luz** e **textura** no nosso projeto.

1.1.2 Engine

O objectivo desta aplicação continua a ser o mesmo: permitir a apresentação de uma janela e exibição dos modelos requisitados. Além disso, permite, também, a interação com os mesmos a partir de certos comandos. Tal como na versão anterior, existe um ficheiro XML que vai ser interpretado. No entanto, para esta fase de projeto, a arquitetura deste ficheiro evolui da maneira a cumprir requisitos. Desta forma, foram feitas algumas alterações nos métodos de parsing e consequentemente alterações nas medidas de armazenamento e também de renderização.

2. Generator

De maneira a podermos aplicar texturas e luz, apareceu a necessidade de calcular normais e coordenadas de textura, par isso calculamos nas funções que estavam a calcular os pontos para cada primitiva também as normais e texturas

2.1 Plano

Neste caso o calcula da normal, é intuitiva bastou apenas considerar a normal (0,1,0) para todos os pontos voltados para cima e (0,-1,0) para os pontos voltados para baixo, as coordenadas da textura no plano o canto superior direito será (1,1), o superior esquerdo (0,1), o inferior esquerdo (0,0) e o inferior direito será (1,0).

2.2 Cubo

Para calcular as normais do cubo as normais da face da frente serão (0,0,1), da face de trás serão (0,0,-1), da direita será (1,0,0) e o da face esquerda será (-1,0,0). Para calculo da coordenadas da textura serão dados por (i/numero_de_divisões) e (j/numero_de_divisões)

2.3 Esfera

Para calcular as normais da esfera apenas temos de ter em conta a direção que apresenta sem dar importância ao valor do raio. Para as coordenadas das texturas só temos de pensar nas coordenadas do quadrado formado pelos 2 triângulos em questão e encontrar o local exato da esfera multiplicando esse valor por 1.0 /stacks ou 1.0 / slices.

2.4 Bezier Patch

Para calcular as normais da superfície de *Bezier*, bastou apenas calcular as derivadas parciais em ordem a **u** e **v**, após isto aplicar o produto vetorial e normalizar, para calculo da derivadas parciais aplicamos a formula matricial já para gerar os pontos mas com a derivada de **u** no caso da derivada parcial em ordem a **u** e **v** no caso da derivada parcial em ordem a **v** já a coordenada das texturas é o próprio **u** e **v**.

3. Engine

3.1 Class Model

Ao longo do processo de desenvolvimento desta fase tivemos a necessidade de inserir uma nova estrutura de dados ao nosso projeto, que englobaria toda a informação precisa para definir um modelo. Em vez de só precisarmos dos vértices de um modelo, agora necessitamos também de lhe associar uma **luz** e **textura**.

As alterações podem ser vistas nos pedaços de código abaixo, aparecendo primeiro a versão da Fase 3 seguida da Fase 4.

```
-------------------------------------------------------------------------
struct VBO {
    int size;
    GLuint vertices;
};

class Grupo{
    vector<Transformacao*> transformations;
    vector<VBO> models;
    vector<Grupo> childgroups;
}
-------------------------------------------------------------------------
struct VBO {
    int size_vertices;
    GLuint vertices;
    int size_normals;
    GLuint normals;
    int size_tex;
    GLuint texCoords;
};
class Model{
    int texFlag; // 0 -> color, 1 -> texture
    VBO v;
    GLuint texture;
    float ambient[4], diffuse[4], specular[4], emissive[4], shininess;
}
class Grupo{
    vector<Transformacao*> transformations;
    vector<Model*> models;
    vector<Grupo> childgroups;
}
-------------------------------------------------------------------------
```

3.2 Class Light

Distinguimos a luz associada a modelos e a fonte de luz criando a class Light guardando nela o último tipo de luz. Nela guardamos o tipo da luz, a sua identifiação e os campos dos vetores como podemos ver em baixo:

```
int type; //0->point, 1->direction, 2->spot
int number;
float position[4], ambient[4], diffuse[4], specular[4], spotDirection[4], cutoff;
```

Guardamos todas as luzes lidas num vector e, antes de fazer o desenho dos modelos, acendemos todas as luzes da seguinte maneira:

```
void renderScene(){
    (...)
    for (auto light : lights) light.turnOn();
    g.drawGroup(time);
    (...)
}
```

Acendemos as luzes com o método mostrado de seguida:

```
void Light::turnOn() {
    if (type == 1) position[3] = 0;
    else position[3] = 1;

    glLightfv(number + GL_LIGHT0, GL_POSITION, position);
    glLightfv(number + GL_LIGHT0, GL_AMBIENT, ambient);
    glLightfv(number + GL_LIGHT0, GL_DIFFUSE, diffuse);
    glLightfv(number + GL_LIGHT0, GL_SPECULAR, specular);

    if (type == 2) {
        glLightfv(number + GL_LIGHT0, GL_SPOT_DIRECTION, spotDirection);
        glLightf(number + GL_LIGHT0, GL_SPOT_CUTOFF, cutoff);
    }
}
```

3.3 VBOs

Uma das mudanças feitas ao nosso projeto foi implementar os VBOs para desenhar os diferentes modelos com luz e textura. Os Virtual Buffer Objects são uma funcionalidade oferecida pelo OpenGL, os quais nos permitem inserir informação sobre os vértices diretamente na placa gráfica do nosso dispositivo. Estes fornecem-nos uma performance bastante melhor, devido ao facto de a renderização ser feita de imediato pois a informação já se encontra na placa gráfica em vez de no sistema, diminuindo assim a carga de trabalho no processador e assim os pontos em vez de estarem a ser desenhados um a um, são passados para um buffer. Passando para a explicação do que foi alterado no nosso código, as alterações foram feitas na classe Model, cuja função de desenho foi alterada para a seguinte versão:

```
void Model::drawModel() {

glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuse);
```

```
    glMaterialfv(GL_FRONT, GL_SPECULAR, specular);
    glMaterialfv(GL_FRONT, GL_AMBIENT, ambient);
    glMaterialfv(GL_FRONT, GL_EMISSION, emissive);
    glMaterialf(GL_FRONT, GL_SHININESS, shininess);

    glBindBuffer(GL_ARRAY_BUFFER, v.vertices);
    glVertexPointer(3, GL_FLOAT, 0, 0);
    glBindBuffer(GL_ARRAY_BUFFER, v.normals);
    glNormalPointer(GL_FLOAT, 0, 0);
    if(texFlag == 1){
        glBindTexture(GL_TEXTURE_2D, texture);
        glBindBuffer(GL_ARRAY_BUFFER, v.texCoords);
        glTexCoordPointer(2, GL_FLOAT, 0, 0);
    }
    //  Draw
    glDrawArrays(GL_TRIANGLES, 0, (v.size_vertices));
    glBindTexture(GL_TEXTURE_2D, 0);
}
```

Para que cada um dos ficheiros.3d tivesse o seu próprio conjuntos de pontos VBO com as normais e texturas, tivemos de alterar na classe engine a função que lê os ficheiros. Acrescentamos o seguinte à função readFile:

```
    glGenBuffers(1, &vecbuf);
    glBindBuffer(GL_ARRAY_BUFFER, vecbuf);
    glBufferData(GL_ARRAY_BUFFER, sizeof(float)*vertices.size(), vertices.data(), GL_STATIC_DRAW);
    glGenBuffers(1, &normbuf);
    glBindBuffer(GL_ARRAY_BUFFER, normbuf);
    glBufferData(GL_ARRAY_BUFFER, sizeof(float)*normal.size(), normal.data(), GL_STATIC_DRAW);
    glGenBuffers(1, &texbuf);
    glBindBuffer(GL_ARRAY_BUFFER, texbuf);
    glBufferData(GL_ARRAY_BUFFER, sizeof(float)*textura.size(), textura.data(), GL_STATIC_DRAW);

    v.size_vertices = vertices.size()/3 ;
    v.vertices = vecbuf;
    v.size_normals = normal.size()/3;
    v.normals = normbuf;
    v.size_tex = textura.size() / 2;
    v.texCoords = texbuf;
```

4. Resultados obtidos

4.1 Sistema Solar

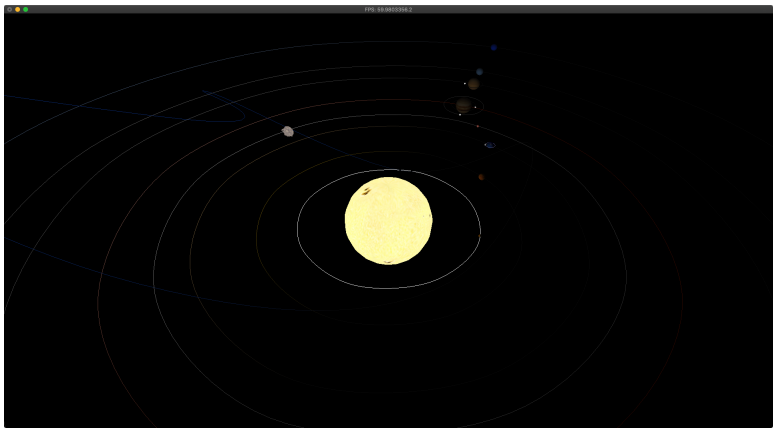

Figura 4.1: Demonstração sistema solar = 0.

4.2 Outros modelos

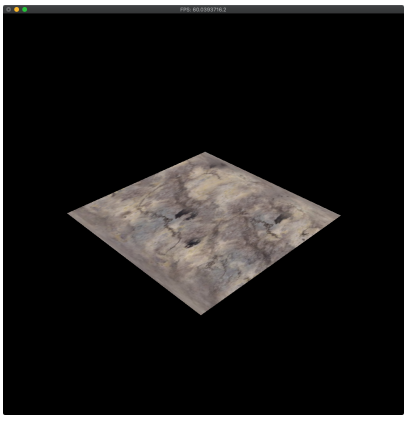

Figura 4.2: Demonstração plano.

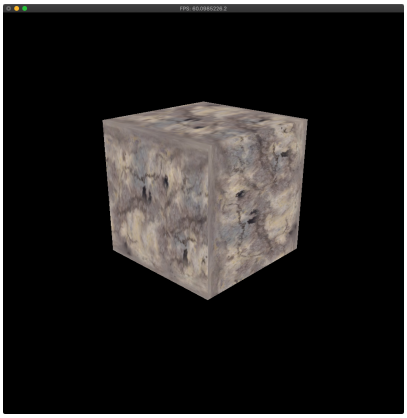

Figura 4.3: Demonstração cubo.

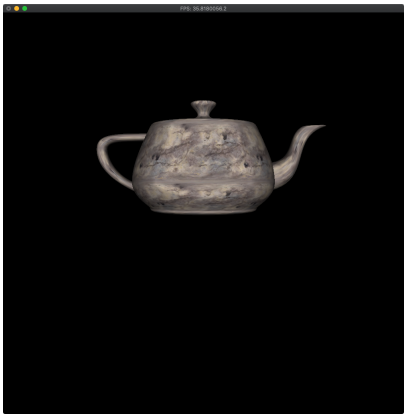

Figura 4.4: Demonstração teapot shininess = 0.

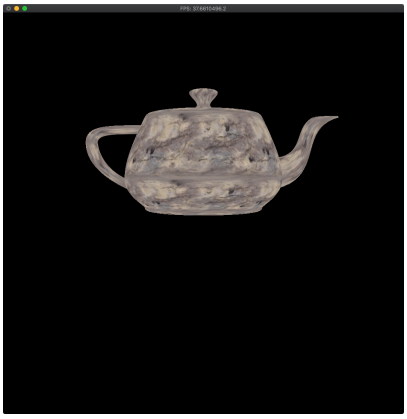

Figura 4.5: Demonstração teapot shininess = 128.

5. Conclusão

O nosso programa começou por apenas ser capaz de, através do Generator, gerar os pontos de várias primitivas e, com a Engine, ler um ficheiro XML com a informação sobre os modelos (e respectivos ficheiros com os vértices gerados anteriormente), e desenhá-los em modo imediato.

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 mesmos.

Posteriormente, foram implementadas funcionalidades relativas à rotação, translação e escala de modelos, definidas por tags específicas do ficheiro XML da cena. Com o auxílio destas novas funcionalidades e com a implementação de novas noções hierárquicas em XML (e.g. tag group), foi possível criar uma cena relativa ao Sistema Solar, com planetas, Sol e luas.

Nesta fase percebemos a importância do polimorfismo do C++, da utilização de containers e das operações de popMatrix e pushMatrix para a definição de grupos hierárquicos complexos e a aplicação correta de transformações geométricas.

Na fase seguinte, as rotações e as translações foram melhoradas. No caso da translação, foi adicionada a capacidade de introduzir pontos parciais 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. Além disso, foi implementado o desenho de modelos a partir da patches de Bezier e a substituição do desenho em modo imediato das primitivas por VBOs que é um método bastante mais eficiente quando os índices são bem escolhidos.

Esta última fase consistiu na geração adicional das normais dos modelos e de coordenadas para as texturas. Assim, foi possível a implementação de vários tipos de luzes para a iluminação da cena e a aplicação de texturas aos modelos. Além disso, implementou-se uma câmara em terceira pessoa que permitiu uma navegação mais real pela cena gerada.