Universidade do Minho
Departamento de Informática

Computação Gráfica

Fase 4 - Grupo 59

26 de maio, 2024

Carlos Ferreira
a89509

Gerson Junior
a88000

Pedro Sousa
a100823

Pedro Viana
a100701

Sumário

1 Introdução

Serve o seguinte relatório para documentar o processo de desenvolvimento da quarta fase do trabalho prático da unidade curricular de Computação Gráfica, do ano letivo 2023/2024.

Nesta fase a aplicação *Generator* deve ser capaz de gerar coordenadas de textura e vetores normais para cada vértice dos modelos.

A aplicação *Engine* deve ser capaz de ler e aplicar as coordenadas de textura e as normais geradas pelo *Generator*.

Para além disto, no ficheiro de configuração XML deve ser possível definir as componentes da cor de um modelo, difusa, especular, emissiva e ambiente, e o brilho do modelo. Deve ser ainda possível definir fontes de luz do tipo ponto, direcional e holofote. O *Engine* deve ser capaz de desenhar estes novos elementos na cena.

2 Generator

A geração de todos os modelos foi reescrita de forma a que cada vértice seja calculado apenas uma vez, evitando trabalho redundante. Anteriormente, cada vértice era calculado tantas vezes quanto o número de triângulos que definia, pelo que o mesmo vértice podia ser calculado até 4 vezes.

Para que o *Engine* seja capaz de desenhar modelos com textura e própria interação com luz, foi necessário expandir *Generator*. Assim, cada vértice de um modelo é agora, não só definido pelas suas coordenadas no espaço, mas também pela sua normal e pelas coordenadas de textura.

Assim, um modelo é agora definido por um conjunto de um novo tipo de dados, o `Vertex`, definido da seguinte forma:

```
struct Vertex {
    Point3 position;
    Vector3 normal;
    Vector2 texCoords;
}
```

2.1 Normais

Uma normal é definida como um `Vector3`.

2.1.1 Plano

Como o plano é definido sobre o plano XZ e está orientada no sentido positivo do eixo Y, as suas normais são todas iguais ao vetor $(0, 1, 0)$;

2.1.2 Caixa

A caixa é definida por 6 faces planas paralelas aos planos coordenados.

As normais da face paralela ao plano XY orientada no sentido positivo de Z são definidas pelo vetor $(0, 0, 1)$ e as normais da face oposta são definidas pelo vetor $(0, 0, -1)$;

As normais da face paralela ao plano YZ orientada no sentido positivo de X são definidas pelo vetor $(1, 0, 0)$ e as normais da face oposta são definidas pelo vetor $(-1, 0, 0)$;

As normais da face paralela ao plano XZ orientada no sentido positivo de Y são definidas pelo vetor $(0, 0, 1)$ e as normais da face oposta são definidas pelo vetor $(0, 0, -1)$;

2.1.3 Esfera

Como o centro da esfera está na origem, as coordenadas dos seus vértices também definem um vetor da origem a cada vértice. Assim, a normal de cada vértice pode ser definida como a normalização do vetor definido pelas coordenadas de cada vértice.

2.1.4 Cone

A base do cone é definido sobre o plano XZ e está orientada no sentido negativo do eixo Y, pelo que as suas normais são todas definidas pelo vetor $(0, 1, 0)$;

Como o eixo do cone coincide com o eixo Y do referencial, as normais podem da superfície lateral podem ser definidas como o vetor polar $(1, \alpha, \beta)$ onde α é o ângulo da *slice* do vértice e β é o ângulo perpendicular à geratriz do cone calculado por

$$\arctan\left(\frac{\text{raio}}{\text{altura}}\right)$$

2.1.5 Superfícies de Bézier

Para calcular a normal de um vértice de uma superfície de Bézier, é necessário calcular a normalização do *cross product* de dois vetores tangentes ao vértice, definidos da seguinte forma:

$$\frac{\partial p(u,v)}{\partial u} = [3u^2 \;\; 2u \;\; 1 \;\; 0] M \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} M^T V^T$$

$$\frac{\partial p(u,v)}{\partial v} = UM \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} M^T \begin{bmatrix} 3v^2 \\ 2v \\ v \\ 0 \end{bmatrix}$$

Segue uma implementação deste cálculo:

```
Vector3 normalUV(Point3 P[16],double UM[4], double VM[4], double dUM[4], double dVM[4])
{
    Vector3 tangentU;
    Vector3 tangentV;

    for (int i = 0; i < 4; i++) {
        Vector3 UMP;
        Vector3 dUMP;
        for (int j = 0; j < 4; j++) {
            UMP += Vector3(P[j * 4 + i]) * UM[j];
            dUMP += Vector3(P[j * 4 + i]) * dUM[j];
        }
        tangentU = tangentU + dUMP * VM[i];
        tangentV = tangentV + UMP * dVM[i];
    }

    return tangentV.cross(tangentU).normalize();
}
```

No projeto, estes cálculos são feitos em conjunto com o cálculo das coordenadas do vértice e da textura, mas esses cálculos foram omitidos por motivos de demonstrativos.

2.2 Coordenadas de Textura

De forma a definir as coordenadas de textura foi definido um novo tipo de dados, o `Vector2`:

```
struct Vector2 {
    float x, y;
}
```

2.2.1 Plano

Uma textura é mapeada no plano com o ponto (0, 0) da textura no vértice $(-\text{size}/2, 0, \text{size}/2)$ do plano.

Sendo "divisions" o número de divisões no plano, o ponto de coordenadas da textura do vértice na linha i, coluna j, é definido por $(j/\text{divisions}, i/\text{divisions})$.

2.2.2 Caixa

A textura é mapeada nas faces da caixa tal como é mapeada no plano.

2.2.3 Esfera

À exceção dos cones nos polos da esfera, uma textura é mapeada na esfera de forma semelhante ao plano. Para cada vértice na *stack* i, slice *j*, o ponto de coordenadas da textura no vértice é definido por $(j/\text{slices}, i/\text{stacks})$.

Os cones nos polos não são compostos por *quads* pelo que é inevitável perder parte da textura entre os triângulos que os definem. De forma a obter resultados idênticos aos *demos* fornecidos com o enunciado foi decidido mapear a textura como na Figura 1.

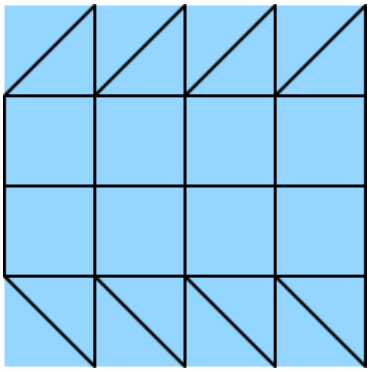

Figura 1: Projeção em 2D do mapeamento de uma textura numa esfera 4×4.

2.2.4 Cone

Como o cone tem duas superfícies distintas, uma textura é mapeada duas vezes para cada superfície.

Na base do cone e na última *stack* da superfície curva, a textura é mapeada de forma semelhante aos polos da esfera, onde parte da textura é perdida entre os triângulos.

Na restante superfície curva, a textura é mapeada de forma semelhante à esfera. Para cada vértice na *stack* i, slice *j*, o ponto de coordenadas da textura no vértice é definido por $(j/\text{slices}, i/\text{stacks})$.

2.2.5 Superfícies de Bézier

Uma textura é mapeada numa superfície de Bézier tal como seria mapeada num plano com tantas divisões como o nível de tesselação da superfície. Assim, as coordenadas de textura de um vértice (u, v) são definidas por $(1 - u, 1 - v)$.

3 Engine

No *Engine* foram ativadas as funcionalidades de luzes e texturas. A leitura dos ficheiros de configuração XML foi expandida para lidar com os novos elementos, luzes, texturas e cores. E a leitura dos ficheiros ".3d" foi desenvolvida para conseguir ler os novos dados (coordenadas de textura e normais).

3.1 Vertex Buffers

De forma a evitar o carregamento redundante de múltiplos modelos iguais foi implementada *memoization* através de um novo tipo de dados, o `VertexBuffers`. Este é utilizado pela classe `Model` e contém as referencias para os *buffers* das posições, normais e coordenadas de textura do um modelo.

Sempre que um novo modelo é carregado, primeiro é verificado se o nome do ficheiro do modelo já foi carregado anterior. Se for o caso, o modelo usa o `VertexBuffers` já carregado, se não, um novo objeto `VertexBuffers` é criado e guardado no mapa.

3.2 Luz

O ficheiro de configuração XML foi expandido para permitir especificar luzes. Uma luz pode ser do tipo direcional, pontual ou holofote.

Segue um exemplo da configuração dos 3 tipos de luzes:

```
<lights>
  <light type="directional" dirx="1" diry="0.7" dirz="0.5"/>
  <light type="point" posx="2" posy="2" posz="-2" />
  <light type="spot" posx="0" posy="2" posz="4" dirx="0" diry="-2" dirz="-4"
cutoff="10"/>
</lights>
```

Para modelar este novo elemento de configuração foi criado um novo tipo de dados, a `Light`, a qual é estendida pelos tipos `DirectionalLight`, `PointLight` e `SpotLight` que modelam cada tipo de luz.

O tipo `Light` contém um `id` do tipo `GLenum` que identifica a luz no OpenGL e especifica que luzes devem implementar um método `void place()`. Este método é responsável por colocar a luz na cena e deve ser chamado na *callback* `renderScene` antes dos modelos serem desenhados. Para esse efeito, a classe `World` contém agora um conjunto de `Light` sobre os quais itera e chama o método `place()` antes de desenhar os modelos.

A luz direcional é definida pela sua direção (`Vector3`). Segue-se a sua implementação:

```
class DirectionalLight : public Light {
    Vector3 dir;

    void place() {
        float dirv[4] = {dir.x, dir.y, dir.z, 0.0f};
        glLightfv(id, GL_POSITION, dirv);
    }
}
```

A luz pontual é definida pela sua posição (`Point3`). Segue-se a sua implementação:

```
class PointLight : public Light {
    Point3 pos;

    void place() {
        float posv[4] = {pos.x, pos.y, pos.z, 1.0f};
        glLightfv(id, GL_POSITION, posv);
```

```
    }
}
```

A luz holofote é definida pela sua posição (`Point3`), a sua direção (`Vector3`) e o ângulo de corte (`float`). Segue-se a sua implementação:

```
class SpotLight : public Light {
    Point3 pos;
    Vector3 dir;
    float cutoff;

    void place() {
        float posv[4] = {pos.x, pos.y, pos.z, 1.0f};
        float dirv[4] = {dir.x, dir.y, dir.z, 0.0f};
        glLightfv(id, GL_POSITION, posv);
        glLightfv(id, GL_SPOT_DIRECTION, dirv);
        glLightf(id, GL_SPOT_CUTOFF, cutoff);
    }
}
```

3.3 Cor

O ficheiro de configuração XML é agora capaz de especificar 5 componentes da cor de um modelo, difusa, ambiente, especular e emissiva com valores RGB de 0 a 255, e brilho entre os valores 0 e 128.

Segue um exemplo de um modelo com a cor especificada:

```
<model file="sphere_1_32_32.3d">
  <color>
    <diffuse R="143" G="85" B="37"/>
    <ambient R="5" G="5" B="5"/>
    <specular R="0" G="0" B="0"/>
    <emissive R="0" G="0" B="0"/>
    <shininess value="0"/>
  </color>
</model>
```

Para modelar este novo elemento de configuração foi criado um novo tipo de dados, o `Color`. Segue-se a implementação do estado deste com os diferentes componentes da cor inicializadas com os seus valores *default*.

```
class Color {
    ColorRGB diffuse{200, 200, 200};
    ColorRGB ambient{50, 50, 50};
    ColorRGB specular{0, 0, 0};
    ColorRGB emissive{0, 0, 0};
    float shininess{0};
}
```

O tipo de dados `ColorRGB` é definido como se segue:

```
struct ColorRGB {
    float r{0), g{0}, b{0};
}
```

A classe `Color` oferece ainda o método `void set()` a ser chamado pelo método `void draw()` da classe `Module`, implementado da seguinte forma:

```
void Color::set() const {
    float d[] = {diffuse.r, diffuse.g, diffuse.b, 1.0f};
```

```
    float a[] = {ambient.r, ambient.g, ambient.b, 1.0f};
    float s[] = {specular.r, specular.g, specular.b, 1.0f};
    float e[] = {emissive.r, emissive.g, emissive.b, 1.0f};
    glMaterialfv(GL_FRONT, GL_DIFFUSE, d);
    glMaterialfv(GL_FRONT, GL_AMBIENT, a);
    glMaterialfv(GL_FRONT, GL_SPECULAR, s);
    glMaterialfv(GL_FRONT, GL_EMISSION, e);
    glMaterialf(GL_FRONT, GL_SHININESS, shininess);
}
```

3.4 Texturas

Para utilizar texturas no *Engine*, foi definido um novo tipo de dados, o `Texture` e, a classe `VertexBuffers` inclui uma referencia a um *buffer object* para as coordenadas de textura do modelo.

O ficheiro da textura é lido a partir da biblioteca `DevIL` e os seus dados são guardados num *buffer* referenciado pelo atributo `id` guardado na classe `Texture`.

Para utilizar a textura esta deve ser "ligada" ao modelo antes de este ser desenhado. Para tal a classe `Texture` oferece o método `bind()` que deve ser chamado no método `draw()` do modelo antes do desenho deste. Esta ligação é opcional e apenas acontece se o modelo tiver uma textura.

4 Demo

Para demonstrar as novas capacidades do *Engine*, foi desenvolvido um demo de um sistema solar.

Neste, na mesma posição do Sol encontra-se uma luz do tipo ponto que simula a luz emita pelo Sol:

```
<lights>
  <light type="point" posx="0" posy="0" posz="0"/>
</lights>
```

O Sol tem uma cor difusa amarelada e uma cor emissiva também amarelada mas mais branca, e uma textura obtida no site https://planetpixelemporium.com/:

```
<model file="sphere_1_32_32.3d">
  <texture file="sunmap.jpg"/>
  <color>
    <diffuse R="253" G="184" B="19"/>
    <ambient R="50" G="50" B="50"/>
    <specular R="0" G="0" B="0"/>
    <emissive R="253" G="251" B="211"/>
    <shininess value="0"/>
  </color>
</model>
```

Todos os outros corpos tem texturas obtidas no mesmo site e utilizam a cor *default* à exceção da cor ambiente que foi definida como (5, 5, 5) de forma a que o lado oposto ao Sol de todos os corpos parecesse estar em escuridão.

Segue-se o resultado deste demo na Figura 2:

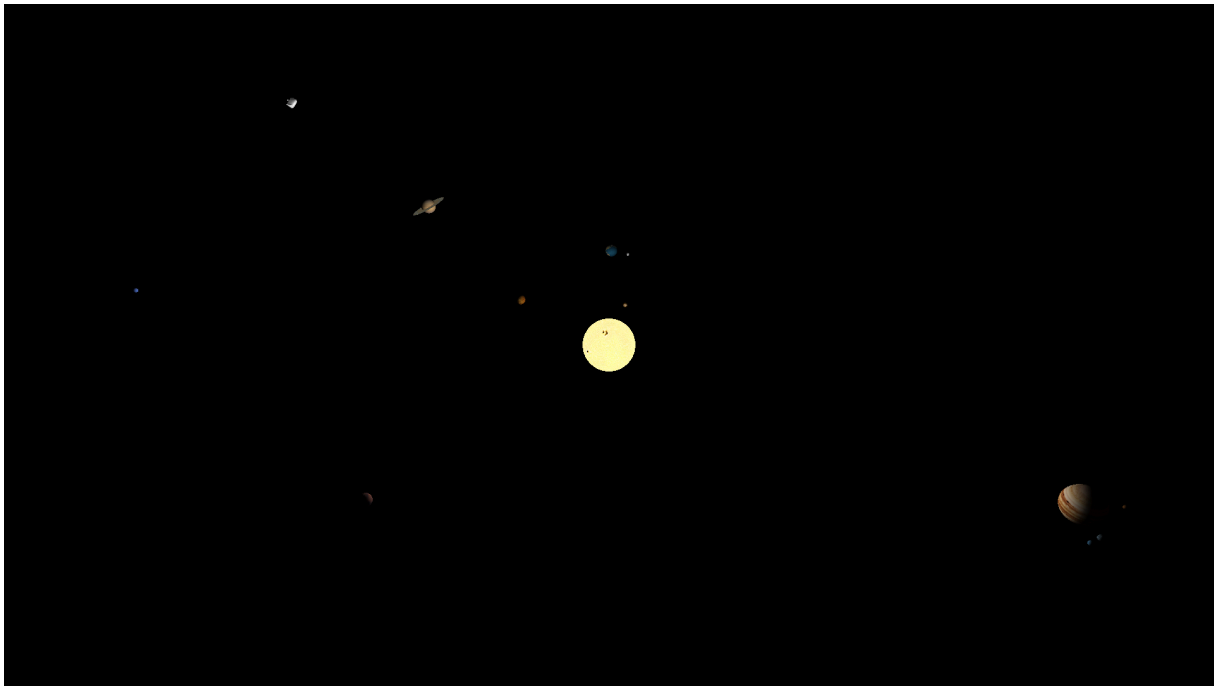

Figura 2: Sistema solar com luzes e texturas.

5 Conclusão

Nesta fase do projeto foi possível identificar e desenvolver as estratégia necessárias para gerar modelos com os dados necessários para que estes interajam apropriadamente com a luz e exibam texturas.

A performance do *Generator* foi melhorada ao evitar cálculos redundantes e a performance da leitura do XML no *Engine* foi melhorada através de *memoization* dos modelos já carregados.

Como o *demo* do sistema solar demonstra, é agora possível desenhar cenas com fontes de luz e com modelos com textura e própria interação com a luz.