Computação Gráfica (3º ano)

Universidade do Minho
Escola de Engenharia

Trabalho Prático

Relatório

João Miguel Faria Leite (A94285)
Joel Vieira Barros (A94692)
Sandro Manuel Fernandes Duarte (A94731)

3 de junho de 2023

Resumo

Este documento apresenta a continuação e a conclusão do trabalho prático realizado no âmbito da Unidade Curricular de Computação Gráfica. Esta última parte tem como objetivo a implementação de texturas nos modelos e a iluminação da cena.

Esta fase provou-se mais desafiante, tendo sido obtidos uma boa parte dos resultados pretendidos. O sistema está montado de acordo com todas as especificações pedidas, apesar de apresentar alguns problemas que fazem com que não funcione a 100%.

Conteúdo

Capítulo 1

Introdução

Na primeira fase deste projeto foram desenvolvidas duas aplicações: o gerador, que permite criar primitivas gráficas; e o motor, que lê ficheiros .xml de configuração, utilizando modelos criados pelo gerador, para exibir um modelo gráfico com a API OpenGl.

Para a segunda fase do Trabalho Pratico foi implementado um sistema de transformações geométricas, que posiciona os modelos na cena.

Com terceira fase foi implementado o uso de VBOs (Vertex Buffer Objects), superfícies cúbicas e animação usando curvas de Catmull-Rom.

Nesta última fase foram propostas novas modificações ao gerador. Este deve agora produzir coordenadas de textura e vetores normais para cada modelo. Com isto, o motor deve ser modificado de forma a desenhar a textura especificada, bem como a ativar a iluminação da cena. Estes parâmetros são passados através do ficheiro de configuração xml.

Capítulo 2

Gerador

De modo a produzir coordenadas de textura e vetores normais, foi necessário modificar cada primitiva.

2.1 Plane

Os vetores normais utilizados nesta primitiva são sempre os mesmos. Neste caso, serão sempre $(0, 1, 0)$, pois este é o vetor perpendicular a cada ponto.

As coordenadas de textura necessitam ser calculadas. Para isso, iteramos por cada vértice de cada triângulo do plano e mapeamos a textura. De notar que as texturas têm como coordenadas pontos 2D, abrangido pontos desde $(0, 0)$ até $(0, 1)$.

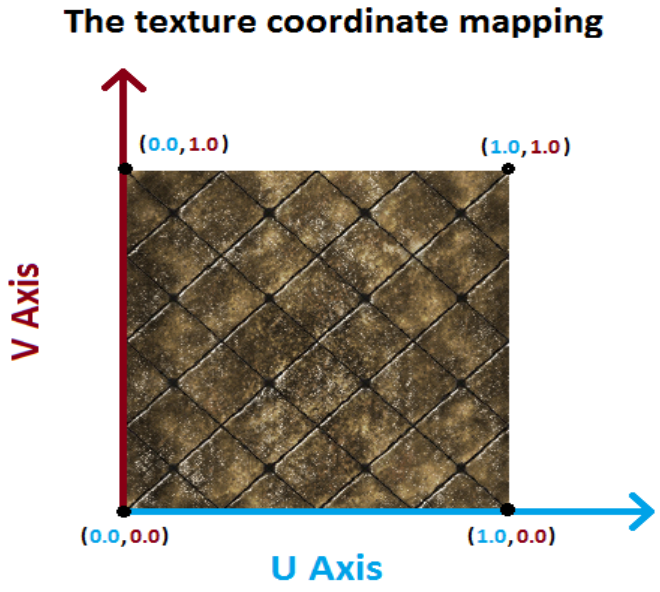

Figura 2.1: *Sistema de coordenadas de texturas*

Como o nosso gerador começa por gerar triângulos no canto superior esquerdo, necessitamos de percorrer a textura pela mesma ordem. Assim, iniciamos a travessia pelo ponto $(0, 1)$ da textura e seguimos a mesma

abordagem que utilizamos para encontrar os vértices. Fazemos triângulos, desta vez 2D, com as coordenadas de textura.

```
    float xTexture = 0.0f;
    float yTexture = 1.0f;

    float delta = 1.0f / static_cast<float>(divisions);

    for (int i = 0; i < divisions; ++i) {
        for(int j = 0; j < divisions; ++j) {
            textureCoordinates.emplace_back(xTexture, yTexture,0);
            textureCoordinates.emplace_back(xTexture, yTexture - delta,0);
            textureCoordinates.emplace_back(xTexture + delta, yTexture - delta,0);

            textureCoordinates.emplace_back(xTexture, yTexture,0);
            textureCoordinates.emplace_back(xTexture + delta, yTexture - delta,0);
            textureCoordinates.emplace_back(xTexture + delta, yTexture,0);

            xTexture += delta;
        }
        xTexture = 0.0f;
        yTexture -= delta;
    }
```

Listing 2.1: Prrocesso de encontrar coordenadas de textura de um plano

2.2 Box

Esta primitiva também contém vetores normais que podem ser pré-calculados. A face de cima tem os mesmos vetores normais que o plano, ou seja, $(0, 1, 0)$. A de baixo é o inverso da de cima, $(0, -1, 0)$. A face esquerda e da direita são $(-1, 0, 1)$ e $(1, 0, 0)$. Por último, a face da frente e de trás são $(0, 0, 1)$ e $(0, 0, -1)$.

O processo de encontrar as coordenadas de textura, no nosso caso, é o mesmo de que o do Plano. Cada face repete a figura da textura.

2.3 Sphere, Cone e Bezier

Os vetores normais destas primitiva são o resultado da normalização de cada vértice do modelo.

Com a Sphere, sendo esta produzida simetricamente a partir do meio da esfera, necessitamos de começar pelo meio da textura, ou seja, o ponto $(0, 0.5)$.

Nas restantes, para encontrar as coordenadas, utilizamos o mesmo método do que o usado para encontrar os pontos dos triângulos.

```
    for (int i = 0; i < slices; ++i) {
        for (int j = 0; j < halfStacks; ++j) {
            float upperX = xTexture + delta;
            float upperY = yTexture - delta;

```

```
6             textureCoordinates.emplace_back(xTexture, 1 - yTexture, 0.0f);
7             textureCoordinates.emplace_back(upperX, 1 - yTexture, 0.0f);
8             textureCoordinates.emplace_back(xTexture, 1 - upperY, 0.0f);
9
10            textureCoordinates.emplace_back(upperX, 1 - yTexture, 0.0f);
11            textureCoordinates.emplace_back(upperX, 1 - upperY, 0.0f);
12            textureCoordinates.emplace_back(xTexture, 1 - upperY, 0.0f);
13
14            float lowerY = yTexture + delta;
15
16            textureCoordinates.emplace_back(xTexture, 1 - yTexture, 0.0f);
17            textureCoordinates.emplace_back(xTexture, 1 - lowerY, 0.0f);
18            textureCoordinates.emplace_back(upperX, 1 - lowerY, 0.0f);
19
20            textureCoordinates.emplace_back(upperX, 1 - lowerY, 0.0f);
21            textureCoordinates.emplace_back(upperX, 1 - yTexture, 0.0f);
22            textureCoordinates.emplace_back(xTexture, 1 -yTexture, 0.0f);
23
24            yTexture -= delta;
25        }
26        xTexture += delta;
27        yTexture = 0.5f;
28    }
```

Listing 2.2: Processo de encontrar coordenadas de textura de uma esfera

Capítulo 3

Motor

3.1 Iluminação

Foi proposto que, agora, o motor conseguisse interpretar fontes de luz do ficheiro de configuração. Existem três tipos de luz: Direcional, *point* e *spotlight*.

De maneira a iluminar a nossa cena, decidimos criar uma classe abstrata que representa uma fonte de luz. Foram criadas subclasses, de modo a implementar os três diferentes tipos de luz.

No momento de arranque do motor, depois de ter sido lido o ficheiro de configuração, verificamos a presença de fontes de iluminação. Caso estas sejam encontradas, procedemos à ativação das mesmas. Cada luz contém um identificador (um valor maior do que 0x4000, de modo a que não haja conflito com outras macros do OpenGL) e é com este que dizemos ao OpenGL qual deve ser acionada.

```
1     if (gWorld->GetLightsSize() != 0) {
2         glEnable(GL_RESCALE_NORMAL);
3         glEnable(GL_LIGHTING);
4
5         GLfloat dark[4] = {0.2,0.2,0.2,1.0};
6         GLfloat white[4] = {1.0,1.0,1.0,1.0};
7
8         float amb[4] = { 1.0f, 1.0f, 1.0f, 1.0f };
9         glLightModelfv(GL_LIGHT_MODEL_AMBIENT, amb);
10
11
12        for (auto const light: *gWorld->GetLights()) {
13            int lightId = light->GetId();
14
15            glEnable(lightId);
16
17            glLightfv(lightId, GL_AMBIENT, dark);
18            glLightfv(lightId, GL_DIFFUSE, white);
19            glLightfv(lightId, GL_SPECULAR, white);
20        }
21    }
22
23    void DirectionalLight::TurnOn() {
```

```
24          glLightfv(mId, GL_POSITION, mDirection);
25      }
26
27      void PointLight::TurnOn() {
28          glLightfv(mId, GL_POSITION, mPosition);
29      }
30
31      void SpotLight::TurnOn() {
32          glLightfv(mId, GL_POSITION, mPosition);
33          glLightfv(mId, GL_SPOT_DIRECTION, mDirection);
34          glLightfv(mId, GL_SPOT_CUTOFF, &mCutOff);
35      }
36  }
```

Listing 3.1: Ativação de fontes de luz

Foi criada uma classe chamada 'Material', que representa as propriedades de como o modelo reage a uma fonte de luz.

3.2 Texturas

Para carregar texturas para o motor usamos a livraria devIL. Utilizando a função LoadTexture, carregamos em memória a imagem e obtemos um identificador que fica associado a esta. O modelo guarda este identificador e, aquando o desenho, utiliza um VBO com as coordenadas de textura.

```
1       void Model::Draw() const {
2           mVertexBuffer.Bind();
3           glVertexPointer(3, GL_FLOAT, 0, 0);
4
5           mNormalBuffer.Bind();
6           glNormalPointer(GL_FLOAT, 0, 0);
7
8           if (mTextureId > 0) {
9               glEnable(GL_TEXTURE_2D);
10
11              // bind the texture
12              glBindTexture(GL_TEXTURE_2D, mTextureId);
13
14              mTextureBuffer.Bind();
15              glTexCoordPointer(2, GL_FLOAT, 0, 0);
16          }
17
18          mMaterial->SetProperties();
19
20          glDrawArrays(GL_TRIANGLES, 0, mPositionArray.GetVertexCount());
21
22          if (mTextureId > 0) {
23              glBindTexture(GL_TEXTURE_2D, 0);
24              glDisable(GL_TEXTURE_2D);
25          }
26
27      }
```

Listing 3.2: Função de desenho do modelo

3.3 Material

A classe Material foi criada para ser usada pelo modelo de forma a definir as propriedades do material de um modelo 3D. As propriedades do material, como ambiente, difusa, mancha especular, emission e brilho determinam como o modelo interage com a luz numa cena.

No método Model::Draw(), as propriedades são aplicadas usando mMaterial¿ =SetProperties(). Isto aciona o material utilizado pelo OpenGL de acordo com os valores especificados.

Usando a classe Material , conseguimos ter controlo sobre como o modelo se parece na cena, sendo possivel obter diferentes efeitos visuais.

```
1 void Material::SetProperties() const {
2     glMaterialfv(GL_FRONT, GL_AMBIENT, mAmbient);
3     glMaterialfv(GL_FRONT, GL_SPECULAR, mSpecular);
4     glMaterialfv(GL_FRONT, GL_DIFFUSE, mDiffuse);
5     glMaterialfv(GL_FRONT, GL_EMISSION, mEmission);
6     glMaterialf(GL_FRONT, GL_SHININESS, mShininess);
7 }
```

Listing 3.3: Função de aplicação de propriedades do material no modelo

Capítulo 4

Conclusão

Com o desenvolvimento da última fase do projeto, foi-nos possível consolidar os conhecimentos adquiridos nas aulas, nomeadamente, a utilização de texturas e iluminação.

Ao longo desta fase, enfrentamos alguns obstáculos, sobretudo na implementação da iluminação e na ativação das propriedades do material do modelo. A cena fica iluminada com base na fonte de luz e o material do modelo reage de maneira adequada à iluminação. Foram testados os ficheiros fornecidos pelos docentes para verificar se o resultado era o pretendido. Persiste ainda a falha na orientação do modelo face a uma curva de Catmull-Rom. A esfera em certos cenários apresenta a textura certa e noutros não.

O modelo do sistema mantém-se animado, com os planetas a fazer rotação em volta do sol, sobre eles próprios e a reagirem a luz emitida pelo sol. A luz nesta cena está na origem.