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)

5 de maio de 2023

Resumo

Este documento apresenta a continuação do trabalho prático realizado no âmbito da Unidade Curricular de Computação Gráfica, que tem como objetivo a implementação de VBOs, curvas e superfícies cúbicas.

Como forma de teste destas novas funcionalidades, foi pedido um modelo de demonstração do sistema solar com animação.

Após a conclusão desta fase foram obtidos quase todos os resultados pretendidos. Os modelos são carregados em VBOs e conseguem seguir curvas de Catmull-Rom, levando o tempo pedido a completar a curva. Além disso, é possível reconhecer ficheiros contendo superfícies de Bezier.

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.

Nesta terceira fase foi proposta a implementação de VBOs (Vertex Buffer Objects) de modo a aumentar a performance da aplicação na altura de desenhar os triângulos compostos pelo modelo. Além disso, foi necessário adicionar uma nova funcionalidade ao gerador, que passa por criar um novo tipo de primitiva, que consegue ler um Bezier Patch de modo a criar uma superfície cúbica. Por fim, o motor passou a ser capaz de criar uma animação, usando curvas de Catmull-Rom.

Para a demonstração desta estrutura foi pedido um modelo dinâmico do sistema solar, incluindo o sol, os planetas e as suas respetivas luas.

Capítulo 2

VBO

Um VBO (Vertex Buffer Object) consiste na criação de um array contiguo, que guarda os vértices de cada triângulo do modelo a ser desenhado.

Assim, cada modelo passsa a ter um array de vértices, o qual está associado a um VBO.

Aquando da criação de um modelo, este array cria o seu buffer, copiando os valores de cada vértice.

```
1  /**
2   * @brief Constructs a VertexBuffer object.
3   * @details Generates a buffer object name.
4   * mRendererID is the renderer in our case is openGL.
5   */
6  VertexBuffer::VertexBuffer() {
7      mRendererID = 0;
8      glGenBuffers(1, &mRendererID);
9  }
10
11 /**
12  * @brief Copies data to the buffer.
13  * @param data pointer to the first element of the array
14  * @param size size of the array in bytes
15  */
16 void VertexBuffer::SetData(const void *data, unsigned int size) {
17     glBindBuffer(GL_ARRAY_BUFFER, mRendererID);
18     glBufferData(GL_ARRAY_BUFFER, size, data, GL_STATIC_DRAW);
19 }
20
21 /**
22  * @brief Binds the buffer.
23  * @details Binds the buffer to the GL_ARRAY_BUFFER target.
24  */
25 void VertexBuffer::Bind() const {
26     glBindBuffer(GL_ARRAY_BUFFER, mRendererID);
27 }
```

Listing 2.1: VertexBuffer

Sempre que queremos utilizar o buffer necessitamos de chamar VertexBuffer::Bind(), de modo a que o

OpenGl saiba qual está a ser utilizado.

Para desenhar os vértices necessitamos de especificar qual é o buffer que está a ser desenhado, que tipo de dados ele guarda (3 floats que equivalem ao x, y, z) e que tipo de primitiva se vai desenhar (no nosso caso, os modelos são conjuntos de triângulos), tal como se pode ver abaixo.

```
1      /**
2       * @brief Draws the model
3       * @details Draws the model using the vertex buffer
4       */
5      void Model::Draw() const {
6          glColor3f(1.0f, 1.0f, 1.0f);
7          mVertexBuffer.Bind();
8          glVertexPointer(3, GL_FLOAT, 0, 0);
9          glDrawArrays(GL_TRIANGLES, 0, mVertexArray.GetVertexCount());
10     }
11 }
```

Listing 2.2: Group::Draw() com uso de VBO

Com esta alteração, é aumentada a rapidez com que é processado cada modelo, uma vez que todos os vértices se encontram num array, que depois é enviado diretamente para a placa gráfica.

Capítulo 3

Superfícies Cúbicas

De modo a que o gerador consiga produzir superfícies cúbicas, necessitamos de criar uma nova primitiva. Esta chama-se Bezier, que reconhece Bezier Patches de um ficheiro estruturado.

```
    32
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
    ...
    290
     1.4, 0, 2.4
    ...
}
```

Listing 3.1: teapot.patch

Cada ficheiro contém o número de patches na primeira linha. Cada patch é uma matriz 4 x 4 de índices. Cada índice corresponde a um ponto da lista de pontos de controlo. No final da lista de índices encontramos o número de pontos de controlo. Cada ponto são três floats. Resumindo, este ficheiro tem 32 matrizes 4 x 4 de pontos.

A primitiva tem como parâmetro o número de segmentos que a superfície vai conter.

```
  for (int p = 0; p < patchSize; ++p) {
       for (int tv = 0; tv < mTesselation; ++tv) {
           float v = static_cast<float>(tv)/static_cast<float>(mTesselation);

           for (int tu = 0; tu < mTesselation; ++tu) {
               float u = static_cast<float>(tu)/static_cast<float>(mTesselation);

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

               vertices.push_back(BezierPoint(p, (u + delta), (v + delta)));
               vertices.push_back(BezierPoint(p, u, (v + delta)));
               vertices.push_back(BezierPoint(p, u, v));

               vertices.push_back(BezierPoint(p, u, v));
               vertices.push_back(BezierPoint(p, (u + delta), v));
               vertices.push_back(BezierPoint(p, (u + delta), (v + delta)));
```

Listing 3.2: Código para encontrar os trinângulos da superficie

Este código é semelhante ao da primitiva Plane, primitiva utilizada anteriormente para gerar um plano. Assim, queremos igualmente iterar sobre uma grelha, de modo a encontrar os vértices de cada triângulo.

O loop externo itera sobre cada patch, enquanto os dois loops internos iteram sobre cada sub-patch(retângulo). Para cada sub-patch, calculamos os 6 pontos de cada triângulo.

Ao variar os parâmetros u e v, podemos calcular uma grelha de pontos na superfície. Utilizamos delta para encontrar os pontos vizinhos desse ponto.

Para calcular cada ponto utilizamos os polinómios de Bernstein.

$$B(u,v) = \begin{bmatrix} u^3 & u^2 & u \end{bmatrix} MPM^T \begin{bmatrix} t^3 \\ t^2 \\ t \end{bmatrix}$$

Como cada patch é uma matriz 4 x 4 de vetores (x, y, z), aquando da inicialização pré-calculamos MPM^T, usando uma matrix 4 x 4 para cada elemento do vetor. Ou seja, uma matriz com todos os valores de x, uma com todos os valores de y e outra com todos os valores de z, de cada patch.

No momento de calculo do ponto em si, calculamos separadamente cada matriz para obter as coordenadas do novo ponto.

Foi criada uma classe Mat que representa uma matriz de floats que executa a multiplicação de matrizes.

Esta fase encontra-se exemplificada com a implementação apresentada de seguida.

```
1  BezierPatch *BezierPatch::Create(const char *pathToFile, uint32_t tesselation) {
2      // ...
3      // Precalculate the M * P * M^T for each patch
4      for (int i = 0; i < totalPatches; ++i) {
5          vector<float> xValues;
6          vector<float> yValues;
7          vector<float> zValues;
8
9          for (int j = 0; j < PATCH_SIZE; ++j) {
10             Vec3f point = controlPoints[indices[i*PATCH_SIZE + j]];
11             xValues.push_back(point.GetX());
12             yValues.push_back(point.GetY());
13             zValues.push_back(point.GetZ());
14         }
15
16         Mat xMatrix(4, 4, xValues);
17         Mat yMatrix(4, 4, yValues);
18         Mat zMatrix(4, 4, zValues);
19
20         Mat xResult = BezierMatrix * xMatrix * BezierMatrixTranspose;
21         Mat yResult = BezierMatrix * yMatrix * BezierMatrixTranspose;
22         Mat zResult = BezierMatrix * zMatrix * BezierMatrixTranspose;
23
24         res->mPatches.push_back({xResult, yResult, zResult});
25     }
26     //...
27 }
28
```

```
29 Vec3f BezierPatch::BezierPoint(uint32_t p, float u, float v) {
30     // p(u,v) = [u^3 u^2 u 1] * M * P * M^T * [v^3 v^2 v 1]^T
31
32     Mat bu(1, 4, {u*u*u, u*u, u, 1});
33     Mat bv(4, 1, {v*v*v, v*v, v, 1});
34     vector<Mat> patch = mPatches.at(p);
35
36     Mat mx = bu * patch.at(0)  *  bv; //mpx
37     Mat my = bu * patch.at(1)  *  bv; //mpy
38     Mat mz = bu * patch.at(2)  *   bv; //mpz
39
40     return {mx.Get(0), my.Get(0), mz.Get(0)};
41 }
```

Listing 3.3: Cálculo dos pontos num Bezier Patch

Capítulo 4

Curvas

Para realizar a animação, os modelos devem mover-se seguindo uma curva de Catmull-Rom. No ficheiro de configuração xml é passada uma lista de pontos de controlo, o tempo que o modelo deve demorar a percorrer a curva e se este deve mudar a sua orientação face ao sentido da curva.

Usando a equação $\begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} M \begin{bmatrix} P0 \\ P1 \\ P2 \\ P3 \end{bmatrix}$, onde M é a matriz de Catmull-Rom, conseguimos obter a posição do ponto no segmento.

```
1   cg_math::Vec3f TranslateCurve::GetCatmullRomPoint(float gt) {
2       float t = ComputeIndices(gt);
3
4       Mat mt(1,4, {t*t*t, t*t, t, 1});
5
6       Mat mp(4,4,{mPoints[mCurrentIndices[0]].GetX(), mPoints[mCurrentIndices[0]].
            GetY(), mPoints[mCurrentIndices[0]].GetZ(), 1,
7                  mPoints[mCurrentIndices[1]].GetX(), mPoints[mCurrentIndices[1]].
                       GetY(), mPoints[mCurrentIndices[1]].GetZ(), 1,
8                  mPoints[mCurrentIndices[2]].GetX(), mPoints[mCurrentIndices[2]].
                       GetY(), mPoints[mCurrentIndices[2]].GetZ(), 1,
9                  mPoints[mCurrentIndices[3]].GetX(), mPoints[mCurrentIndices[3]].
                       GetY(), mPoints[mCurrentIndices[3]].GetZ(), 1});
10
11      Mat mpc = mp * C_MAT;
12      Mat res = mt * CATMULL_ROM_MAT * mpc;
13
14      return {res.Get(0,0), res.Get(0,1), res.Get(0,2)};
15  }
```

Listing 4.1: Cálculo de um ponto na curva Catmull-Rom

Com este ponto, efetuamos uma translação para a nova posição. Caso seja preciso mudar a orientação, usamos a derivada para calcular a matriz de rotação, de modo a rodar o modelo face à curva. Para que este percorra a curva num determinado tempo predefinido, calculamos a velocidade ($1/tempo$) e multiplicamo-la pelo tempo decorrido, de forma a obter o ponto seguinte na curva.

```
        float delta = glutGet(GLUT_ELAPSED_TIME) / 1000.0f;
        // Compute the speed at which the object should move
        float speed = 1 / mTime; // mTime seconds to complete the curve
        // Compute the total distance traveled by the object
        float distance = speed * delta;

        Vec3f p = GetCatmullRomPoint(distance);
        glTranslatef(p.GetX(), p.GetY(), p.GetZ());

        if(mAlign) { Align(distance); }
    }

    void TranslateCurve::Align(float t) {
        Vec3f d = GetCatmullRomDerivative(t);
        Vec3f x = Vec3f::Normalize(d);
        Vec3f z = Vec3f::Normalize(Vec3f::Cross(x, mPrevY));
        Vec3f y = Vec3f::Normalize(Vec3f::Cross(z, x));
        mPrevY = y;

        Mat rot = Mat::RotationMatrix(x, y, z);

        glMultMatrixf(rot.GetData());
    }
```

Listing 4.2: Cálculo do movimento

Capítulo 5

Conclusão

Com o desenvolvimento da terceira fase do projeto, foi-nos possível consolidar os conhecimentos adquiridos nas aulas, nomeadamente, a importância do impacto que VBOs têm na performance e o cálculo de pontos numa curva, que permite a animação e a criação de superfícies cúbicas.

Ao longo desta fase, enfrentamos alguns obstáculos, sobretudo nas multiplicações de matrizes e na ordem em como deviam ser feitas. Sendo este um dos problema que ainda persiste, pois a mudança de orientação dos modelos não parece correta.

Podemos considerar que foi elaborado um trabalho bastante coeso e que, apesar das dificuldades encontradas, foram superados muitos desafios. Os planetas no modelo do sistema solar seguem uma curva com 10 pontos equidistantes encontrados no perímetro de uma circunferência de raio igual á distância do sol. Já no caso da lua, utiliza-se a distância ao planeta. Abaixo, encontra-se representado graficamente o modelo do sistema solar, de acordo com o que nos foi solicitado no enunciado.