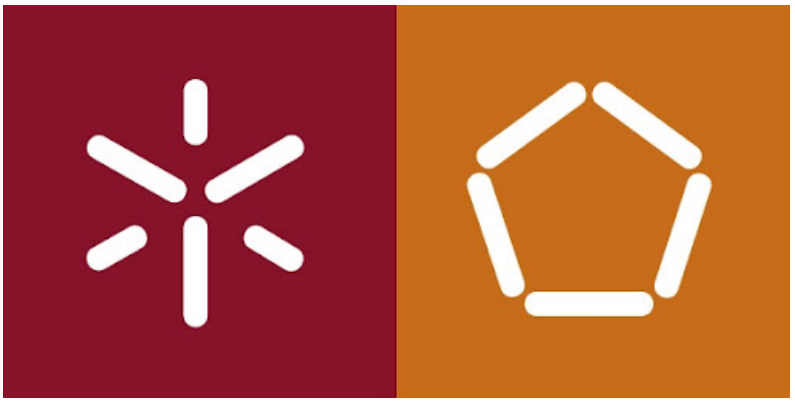
Universidade do Minho
Escola de Engenharia



Curves, Cubic Surfaces and VBOs

Computação Gráfica

Relatório do Trabalho Prático

Grupo nº44
Alexandre Eduardo Vieira Martins A93242
José Eduardo Silva Monteiro Santos Oliveira A100547
Pedro Afonso Moreira Lopes A100759
Pedro Silva Ferreira A97646

26 de abril de 2024

Índice

Introdução

Nesta fase do trabalho, procedemos então à implementação dos VBOs, o que tornou o processo de desenho das figuras mais eficiente e logicamente simples, o que nos vai ajudar para a fase 4, no futuro. Adicionalmente, implementamos também as curvas Catmull-Rom, o que nos permite agora criar um sistema solar com cometas a atravessá-lo, por exemplo, e muito mais.

Finalmente, tendo sido o mais custoso matematicamente, adicionamos também o tratamento de patches de *Bezier*, podendo assim desenhar *teapots,* por exemplo, através da leitura e conversão dos pontos de controlo dos patches em pontos de vértices, que vão ser utilizados para desenhar a figura em si.

Generator

Leitura e conversão dos Patches

De forma a criar os ficheiros .3d a partir de um ficheiro Patch, tivemos primeiro que perceber a informação contida nestes ficheiros, chegando à conclusão abaixo representada:

Example:

indices for the first patch

```
2 <- number of patches
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
3, 16, 17, 18, 7, 19, 20, 21, 11, 22, 23, 24, 15, 25, 26, 27
28 <- number of control points
1.4, 0, 2.4 <- control point 0
1.4, -0.784, 2.4 <- control point 1
0.784, -1.4, 2.4 <- control point 2
0, -1.4, 2.4
1.3375, 0, 2.53125
1.3375, -0.749, 2.53125
0.749, -1.3375, 2.53125
0, -1.3375, 2.53125
1.4375, 0, 2.53125
1.4375, -0.805, 2.53125
0.805, -1.4375, 2.53125
0, -1.4375, 2.53125
1.5, 0, 2.4
1.5, -0.84, 2.4
0.84, -1.5, 2.4
0, -1.5, 2.4
-0.784, -1.4, 2.4
-1.4, -0.784, 2.4
-1.4, 0, 2.4
-0.749, -1.3375, 2.53125
-1.3375, -0.749, 2.53125
-1.3375, 0, 2.53125
-0.805, -1.4375, 2.53125
-1.4375, -0.805, 2.53125
-1.4375, 0, 2.53125
-0.84, -1.5, 2.4 <- control point 26
-1.5, -0.84, 2.4 <- control point 27
```

indices for the second patch

Sabendo isso, procedemos então à leitura do ficheiro .patch, guardando os índices e os pontos de Controlo em estruturas diferentes.

Com isto feito, tendo agora toda a informação necessária para o cálculo dos vértices de desenho da figura, procedemos então aos cálculos necessários para a derivação dos vértices, seguindo as seguintes fórmulas:

$$\text{Let } U = [u^3 \quad u^2 \quad u \quad 1] \text{ and } M = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Figura 1. Representação de variáveis

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

Figura 2. Fórmula do cálculo de um ponto B, para determinados u e v

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

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

Figura 3. Fórmulas do cálculo das tangentes u e v, respetivamente

Com estas fórmulas, conseguimos calcular os vértices necessários para o desenho da figura, aplicando esses cálculos para todos os patches no ficheiro patch. Não só conseguimos calcular isso, mas também as coordenadas normais, necessárias para a fase 4, através do produto cruzado entre ambos os vetores tangentes.

Nos cálculos em si, como nós já pré-calculamos a matriz transposta e a matriz de pontos de Controlo, basta então, na prática, apenas multiplicar estes valores pelos vetores u e v.
Porém, com os pontos calculados, precisamos de arranjar maneira de criar uma ordem de desenho dos vértices, de forma a *engine* poder desenhar de forma correta. Para isto, decidimos seguir uma lógica alternativa de desenho de um plano, iterando ao longo dos pontos em cada patch e, assim, criando os seus índices de desenho.

Com estes cálculos feitos, fica assim então concluída a criação do ficheiro 3d, com as informações necessárias para o seu desenho.

Engine

As mudanças feitas na engine foram todas para acomodar as curvas de *Bézier* e as funcionalidades que vêm com elas.

Para começar, tivemos que alterar o parsing do xml e, para isso, tivemos que mudar a *struct Transform*, adicionando os novos parâmetros e um novo construtor.

Os parâmetros adicionados são um vetor de coordenadas, este guarda os *points* presentes num *translate*, o *align* (diz se o objecto deve ser orientado na curva) e o *time* (tempo de translação do objeto).

O novo construtor é aplicado no caso de um *translate* de movimento, e o construtor que já estava criado será posteriormente aplicado a mais casos.

```
struct Transform {
    std::string type;
    float angle, x, y, z, time;
    bool align;
    std::vector<Coordenadas> pontos;

    Transform(std::string type, float angle, float x, float y, float z) :type(type), angle(angle), x(x), y(y), z(z) {}
    Transform(std::string type, float time, bool align, std::vector<Coordenadas> pontos) :type(type), time(time),
align(align), pontos(pontos) {}

};
```

Figura 4. Estrutura Transform atualizada

Nas funções de parsing, as mudanças foram todas feitas na *processTransform Element.* Na imagem seguinte é possível ver que agora um translate vai ser dividido em dois casos, o *translate* que muda a posição inicial do modelo 3D(type = translate), e o *translate* de movimento (type = translateP).

```
void processTransformElement(tinyxml2::XMLElement* transformElement, Group& og_group) {

    ...

    for (tinyxml2::XMLElement* child = transformElement->FirstChildElement(); child;
         child = child->NextSiblingElement()) {
        const char* childName = child->Name();

        if (strcmp(childName, "translate") == 0) {
            if (child->QueryFloatAttribute("time", &time) == tinyxml2::XML_SUCCESS &&
                child->QueryBoolAttribute("align", &align) == tinyxml2::XML_SUCCESS) {
                // translate with time and align
                std::vector<Coordenadas> points;
                for (tinyxml2::XMLElement* pointElement = child->FirstChildElement("point"); pointElement; pointElement
= pointElement->NextSiblingElement("point")) {
                    float px, py, pz;
                    pointElement->QueryFloatAttribute("x", &px);
                    pointElement->QueryFloatAttribute("y", &py);
                    pointElement->QueryFloatAttribute("z", &pz);
                    points.push_back(Coordenadas(px, py, pz));
                }
                Transform transform = Transform("translateP", time, align, points);
                og_group.transforms.push_back(transform);
            }
            else {
                // Simple translation
                child->QueryFloatAttribute("x", &tx);
                child->QueryFloatAttribute("y", &ty);
                child->QueryFloatAttribute("z", &tz);
                Transform transform = Transform("translate", 0, tx, ty, tz);
                og_group.transforms.push_back(transform);
            }
        }
        else if ...
    }

}
```

Figura 5. Tipos do translate na processTransformElement

Translação por curva Catmull-Rom

Estes dois tipos de *translate* são posteriormente tratados na *handle_group()* e, nesta fase, foi adicionado o tratamento do *translate* de movimento. Para fazer isso, recorre-se a *processCatmullRomTranslation(),* que inclui invocações para *renderCatmullRomCurve()*, responsável por desenhar cada curva, e a getGlobalCatmullRomPoint(), que calcula as coordenadas que servem imediatamente a seguir para a glTranslatef() e toma em conta o fator do tempo corrente para a velocidade de percurso.

```
void processCatmullRomTranslation(Transform transform){

    float pos[3], deriv[3];

    renderCatmullRomCurve(pos, deriv, transform);
    getGlobalCatmullRomPoint(transform.time, pos, deriv, transform);
    glTranslatef(pos[0], pos[1], pos[2]);

    float xv[3],
          yv[3] = {0.0f,1.0f,0.0f},
          zv[3];

    normalize(xv);
    cross(xv,yv,zv);
    normalize(zv);
    cross(zv,xv,yv);
    normalize(yv);

    if (transform.align){
        float rot[16];
        buildRotMatrix(xv,yv,zv,rot);
        glMultMatrixf(rot);
    }
}
```

Figura 6. Função de entrada no cálculo das curvas Catmull-Rom

```
void getGlobalCatmullRomPoint(float gt, float* pos, float* deriv,
Transform transform) {

    int pointCount = transform.pontos.size();
    float t = gt * pointCount;
    int index = floor(t);
    t = t - index;

    // indices store the points
    int indices[4];
    indices[0] = (index + pointCount - 1) % pointCount;
    indices[1] = (indices[0] + 1) % pointCount;
    indices[2] = (indices[1] + 1) % pointCount;
    indices[3] = (indices[2] + 1) % pointCount;

    getCatmullRomPoint(
        t,
        indices,
        transform,
        pos,
        deriv
    );
}
```

Figura 7. Função de seleção dos pontos para posterior multiplicação matricial.

Outra das mudanças nesta função foi como o *rotate* é tratado. Tal como fizemos para o translate, dividimos o rotate em dois tipos: o *rotate* que define a posição inicial e o *rotate* que roda o modelo no decorrer da execução da *scene*. Tal como podemos ver na figura abaixo, dê-mos os nomes *rotateT* e *rotateA* aos dois diferentes tipos. Ambos utilizam o mesmo construtor mas são diferenciados pela string *type*.

```
void processTransformElement(tinyxml2::XMLElement* transformElement, Group& og_group) {

    ...

        else if (strcmp(childName, "rotate") == 0) {

            if (child->QueryFloatAttribute("time", &time) == tinyxml2::XML_SUCCESS) {
                child->QueryFloatAttribute("time", &time);
                child->QueryFloatAttribute("x", &rx);
                child->QueryFloatAttribute("y", &ry);
                child->QueryFloatAttribute("z", &rz);
                Transform transform = Transform("rotateT", time, rx, ry, rz);
                og_group.transforms.push_back(transform);
            }
            else {
                child->QueryFloatAttribute("angle", &angle);
                child->QueryFloatAttribute("x", &rx);
                child->QueryFloatAttribute("y", &ry);
                child->QueryFloatAttribute("z", &rz);
                Transform transform = Transform("rotateA", angle, rx, ry, rz);
                og_group.transforms.push_back(transform);
            }

        }
        else if ...
    }

}
```

Figura 8. Tipos do rotate na processTransformElement

O handle do rotate é feito também na handle_groups da seguinte forma:

```
void handle_groups(const Group& group) {

    pushMatrix();

    for (const auto& transform : group.transforms) {

            ...
        } else if (transform.type == "rotateA") {
            glRotatef(transform.angle, transform.x, transform.y, transform.z);
        } else if (transform.type == "rotateT") {
            float angle = (elapsedTime / transform.angle) * 360.0f;
            glRotatef(angle, transform.x, transform.y, transform.z);
        ...

    popMatrix();
}
```

Figura 9. Handle dos rotates no handle_group

O *rotateT*, que trata do rotate do *model* na execução do programa, de acordo com um certo tempo, calcula o ângulo atual e atualiza a posição do modelo com esse valor.

VBOs

Para implementar as VBOs, voltamos a criar ficheiros .3d com 6 pontos para cada 2 triângulos, em vez dos 4 que usávamos antes, visto que utilizamos VBOs sem índices que, apesar de serem menos eficientes e ocuparem mais espaço, são mais simples de se implementar e a sua manutenção é mais simples.

Cada Group, a fim de se implementarem as VBOs, foi adaptado. Antes, ele guardava o caminho de cada ficheiro 3d com os pontos necessários para construir as figuras. Atualmente, ele armazena o conjunto de modelos que ele possui. Estes armazenam a sua própria VBO, que é criada durante o parse do xml, as coordenadas dos pontos e o número de pontos da figura.

Para o desenho deste modelo, foi criada uma função draw genérica. Esta, em vez de dar parse aos pontos sempre que necessita desenhar, recebe a VBO e o número de pontos que irá desenhar, e desenha-os, assim aumentando a performance da engine.

Demo Scene

Nesta fase, a demo scene criada, depois de todas as implementações feitas, ficou da seguinte maneira:

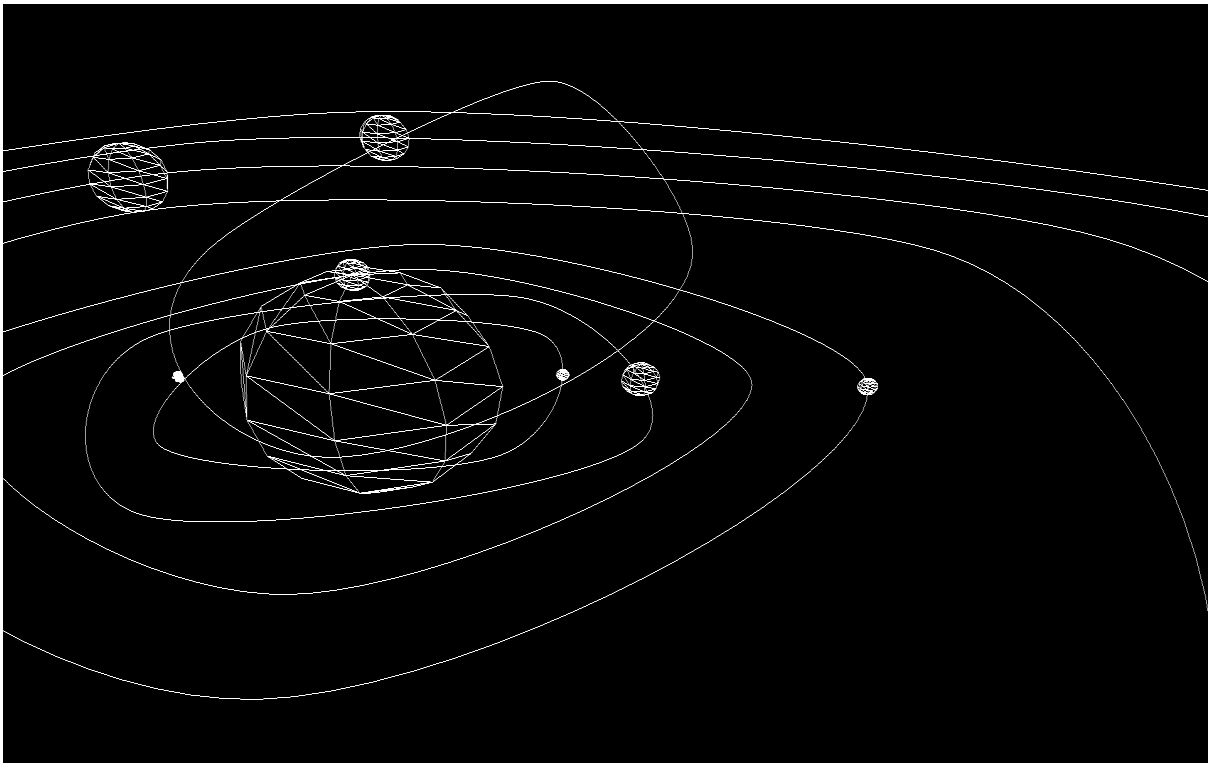


Figura 10. Representação da Demo Scene

As órbitas foram definidas com 4 pontos, o que explica o formato ligeiramente quadrado. No entanto, a implementação está perfeitamente capaz de receber definições de curvas com mais pontos.

Além disto, para a melhor representação dos objetos com a câmara e melhor auxílio em estados de debug do código, decidimos implementar também botões para desligar/ligar os eixos e as curvas Catmull-Rom, sendo estes botões o X e o A, respetivamente.

Conclusão

Para a próxima fase, a separação das coordenadas de cada modelo permite que sejam também armazenadas as normais e as texturas juntamente com as coordenadas, que permitirá armazená-las e desenhá-las ao mesmo tempo que o resto sem ser necessária grande adaptação à estrutura do projeto. Além disto, acreditamos que todos os objetivos foram correspondidos e ficamos também a perceber a necessidade e importância desta fase em relação à próxima.