Universidade do Minho

Licenciatura em Ciências da Computação

Computação Gráfica

Phase 3 - Curves, Cubic Surfaces and VBOs

Grupo Nº 1

Bruna Carvalho (a87982) | Carlos Beiramar (a84628) | Daniel Ferreira (a85670) | Ricardo Cruz (a86789)

2 de maio de 2021

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

O objetivo desta fase passa por fazer alterações tanto na aplicação *generator* assim como na aplicação *engine*.
Na aplicação *generator* é necessário incluir um novo tipo de modelo baseado no *Bezier Patches*. O *generator* vai receber como parâmetros o nome do ficheiro *Bezier*, onde contém os pontos de controlo que são definidos, assim como o *tesselation level*.
É, também, necessário criar uma função *torus* no *generator*, de modo a que o planeta Saturno, tenha um anel.
No que diz respeito ao *engine*, é preciso reestruturar os elementos de translação e de rotação. Para a translação, com o conjunto de pontos que o *engine* vai ler do ficheiro *XML* vai ser definido uma curva cubica de *Catmull-Rom* assim como o número de segundos que demora a percorrer a curva toda. Isto tudo, com a intenção de realizar animações baseadas nestas curvas. Os modelos podem ter um tempo dependente da translação. Nas rotações, o ângulo vai ser substiruído por um tempo, isto é, o tempo, em milissegundos, em que o modelo demora a percorrer $360^{\underline{o}}$ à volta de um eixo específico.
Nesta fase, será necessário recorrer à tecnologia dos **VBOs** para desenhar os modelos.
O resultado final desta fase, será apresentar uma "cena"de um sistema solar dinâmico, incluindo a trajetória de um cometa usando a curva de *Catmull-Rom*. O cometa tem de ser construído usando *Bezier Patches*, partindo do ficheiro que é dado para o *generator* conter os pontos de controlo para o *teapot*.

Capítulo 2

Alterações no *Generator*

2.1 Leitura de *Patches de Bezier*

Nesta nova fase, foi necessário acrescentar uma opção diferente ao *Generator*, que permitisse construir desenhos, a partir de ficheiros fornecidos, conhecidos como *Patches de Bezier*. Dessa forma, foram implementadas algumas alterações para que tais ficheiros pudessem ser reconhecidos.
A leitura de um *Patch de Bezier* é feita de forma a obter a seguinte informação, pela ordem referida:

- Número de *Patches*

```
// leitura de número de patches
getline(myfile, line);
nPatches = atoi(line.c_str());
```

Figura 2.1: Leitura do número de *Patches*

- Índices dos pontos de controlo de cada *Patch*

```
// leitura dos índices
int x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13, x14, x15, x16;

for (int i = 0; i < nPatches; i++) {
    getline(myfile, line);
    removeChar(line, ',');
    istringstream data(line.c_str());
    data >> x1 >> x2 >> x3 >> x4 >> x5 >> x6 >> x7 >> x8 >> x9 >> x10 >> x11 >> x12 >> x13 >> x14 >> x15 >> x16;
    vector<int> v = { x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13, x14, x15, x16 };
    indexVector.push_back(v);
}
```

Figura 2.2: Leitura dos *indices*

- Número de pontos de controlo

```
//leitura do número de pontos
getline(myfile, line);
nPoints = atoi(line.c_str());
```

Figura 2.3: Leitura do número de pontos de controlo

- Lista de todos os pontos de controlo

```
vector<vector<float>> coordinatesVector;

// leitura das coordenadas dos pontos
float p1, p2, p3;

for (int i = 0; i < nPoints; i++) {
    getline(myfile, line);
    removeChar(line, ',');
    istringstream data(line.c_str());
    data >> p1 >> p2 >> p3;
    vector<float> v = { p1,p2,p3 };
    coordinatesVector.push_back(v);
}
```

Figura 2.4: Leitura de todos os pontos de controlo

2.1.1 Curvas de *Bezier*

Depois de lido o ficheiro *Patch de Bezier*, foram desenvolvidas as funções para calcular os pontos tendo em conta o algoritmo das curvas de *Bezier*.
Para ser obtido um ponto, com um parâmetro t, utilizando a fórmula de *Bezier*, foi criada a seguinte função, que através de 4 dos pontos de controlo, nos retorna esse ponto.
A fórmula que utilizada foi a seguinte, fornecida nas aulas:

$$p(t) = t^3P_3 + 3t^2(1-t)P_2 + 3t(1-t)^2P_1 + (1-t)^3P_0$$

Figura 2.5: Expressão auxiliar para o cálculo

```
Vertex bezierCurve(float t, Vertex p1, Vertex p2, Vertex p3, Vertex p4) {
    vector<Vertex> points = { p1,p2,p3,p4 };
    float vectorT[4] = { powf((1 - t),3) , 3 * t * powf((1 - t),2) , 3 * (1 - t) * powf(t,2) , powf(t,3) };

    Vertex result = multMatrixVector(points, vectorT);
    return result;
}
```

Figura 2.6: Cálculo do ponto de *Bezier*

Este processo é repetido em todos os *Patches* , utilizando 4 pontos de controlo em cada vez.

```
Vertex bezier(float p, float q, vector<vector<float>> coordinatesVector, vector<int> indexes) {
    vector<Vertex> controlPoints;
    for (int i = 0; i < 4; i++) {

        Vertex p1(coordinatesVector[indexes[4 * i]][0], coordinatesVector[indexes[4 * i]][1], coordinatesVector[indexes[4 * i]][2]);
        Vertex p2(coordinatesVector[indexes[4 * i + 1]][0], coordinatesVector[indexes[4 * i + 1]][1], coordinatesVector[indexes[4 * i + 1]][2]);
        Vertex p3(coordinatesVector[indexes[4 * i + 2]][0], coordinatesVector[indexes[4 * i + 2]][1], coordinatesVector[indexes[4 * i + 2]][2]);
        Vertex p4(coordinatesVector[indexes[4 * i + 3]][0], coordinatesVector[indexes[4 * i + 3]][1], coordinatesVector[indexes[4 * i + 3]][2]);

        Vertex controlPoint = bezierCurve(p, p1, p2, p3, p4);

        controlPoints.push_back(controlPoint);

    }
    return bezierCurve(q, controlPoints[0], controlPoints[1], controlPoints[2], controlPoints[3]);
}
```

Figura 2.7: Ciclo de cálculo dos pontos de *Bezier*

Por fim, estes pontos, depois de calculados são adicionados a um vetor que depois vai ser escrito no ficheiro de destino, definido aquando da invocação do *generator*.

2.2 *Torus*

Para criar o anel de Saturno, desenhou-se uma forma geométrica chamada de *Torus*. Utilizando os seguintes parâmetros:

- Raio exterior
- Raio interior
- *Slices*
- *Stacks*

Com o número de *Slices* e de *Stacks*, cálculamos o valor de variação de 2 ângulos (Φ e Θ).

$$\Theta = 360/slices$$
$$\Phi = 360/stacks$$

De seguida, foram calculados os pontos utilizando um algoritmo semelhante ao algoritmo para as esferas. Utilizou-se 2 ciclos, para percorrer o número de *Slices* e de *Stacks*. Em cada iteração do ciclo interior (*Stacks*) era aumentado o Φ, pela amplitude da segunda expressão acima enquanto que o Θ era incrementando pela primeira expressão acima. Nota ainda para o factor de ser utilizada a expressão *outerRadius + radius*.

```
for (int i = 0; i < slices; i++) {
    for (int j = 0; j < stacks; j++) {
        Vertex v1( x: (cos(theta) * (distance + radius * cos(phi))), y: (sin(theta) * (distance + radius * cos(phi))), z: (radius * sin(phi)));

        Vertex v2( x: (cos( x: theta + theta_shift) * (distance + radius * cos(phi))),
                   y: (sin( x: theta + theta_shift) * (distance + radius * cos(phi))), z: (radius * sin(phi)));

        Vertex v3( x: (cos( x: theta + theta_shift) * (distance + radius * cos( x: phi + phi_shift))),
                   y: (sin( x: theta + theta_shift) * (distance + radius * cos( x: phi + phi_shift))), z: (radius * sin( x: phi + phi_shift)));

        Vertex v4( x: (cos( x: theta + theta_shift) * (distance + radius * cos( x: phi + phi_shift))),
             y: (sin( x: theta + theta_shift) * (distance + radius * cos( x: phi + phi_shift))), z: (radius * sin( x: phi + phi_shift)));


        Vertex v5( x: (cos(theta) * (distance + radius * cos( x: phi + phi_shift))),
                   y: (sin(theta) * (distance + radius * cos( x: phi + phi_shift))), z: (radius * sin( x: phi + phi_shift)));


        Vertex v6( x: (cos(theta) * (distance + radius * cos(phi))),
                   y: (sin(theta) * (distance + radius * cos(phi))), z: (radius * sin(phi)));

        Triangle t1(v1, v2, v3);
        Triangle t2(v4, v5, v6);
        triangles.push_back(t1);
        triangles.push_back(t2);
        phi = phi_shift * (j + 1);
    }
    theta = theta_shift * (i + 1);
}
```

Figura 2.8: Cálculo do *Torus*

Capítulo 3

Alterações no *Engine*

3.1 Classe *Group*

Nesta classe, foi necessário fazer uma reestruturação visto que as translações deixaram de ser só baseadas num **X,Y,Z** e passaram a ser um conjunto de pontos em que, cada ponto tem **X,Y,Z** que, posteriormente, vão ser utilizados para calcular a curva de *Catmull-Rom*.
Foi, ainda, acrescentado um vetor *time* e um vetor *elapsedTime*, de modo a que o o primeiro guarde o tempo definido em cada translação e o segundo, guarde o tempo que passou até então.

```
class Group {

public:
    std::vector<std::vector<std::vector<float>>> translation;
    std::vector<float> rotation;
    std::vector<float> scale;
    std::vector<float> vbo;
    std::vector<std::vector<float>> triangleCoordinates;
    std::vector<float> colors;
    std::vector<float> time = { 1,1,1,1,1 };
    std::vector<float> elapsedTime = { 0,0,0,0 };
    int pos;
    float rotationTime;
    float angle = 0.0f;
```

Figura 3.1: Classe *Group*

3.2 *VBO*

A introdução de *VBO* (Virtual Buffer Objects) permitiu melhorar a eficiência visto que, os pontos dos triângulos são guardados num *buffer* diretamente da placa gráfica, levando a um desenho e a um *parsing* mais rápidos. Deste modo, o *VBO* foi implementado da seguinte maneira:

```
glBindBuffer(GL_ARRAY_BUFFER, buffers);
glBufferData(GL_ARRAY_BUFFER,  size: sizeof(float) * aux->vbo.size(), aux->vbo.data(), GL_STATIC_DRAW);
glVertexPointer( size: 3, GL_FLOAT,  stride: 0,  pointer: 0);
glDrawArrays(GL_TRIANGLES,  first: 0,  count: ((GLuint)aux->vbo.size() / 3));
```

Figura 3.2: VBO

3.3 Curvas de *Catmull-Rom*

3.3.1 Transformações

Os valores inseridos nos *arrays* foi feita com o auxílio da função **getGlobalCatmullRom** que recorre à função **getCatmullRomPoint**. Esta última, vai recorrer a operações entre dois vetores, uma matriz e os pontos.

$$\mathrm{M} = \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1.0 & -2.4 & 2.0 & -0.5 \\ -0.5 & 0.0 & 0.5 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \end{bmatrix}$$

$$\mathrm{vectorT} = (t^3\ t^2\ t\ 1)$$

$$\mathrm{vectordT} = (3 * t^3\ 2 * t^2\ t\ 1)$$

A função **getCatmullRomPoint** preenche os *arrays* obtendo os valores através da multiplicação da matriz **M** pelo vetor que contém os **pontos recolhidos do ficheiro XML** (vectorA). Com o vetor resultante desta multiplicação e o vetor *vectorT* indicado em cima, obtemos os valores para preencher o vetor **pos** (pontos para a próxima translação na curva), e se o multiplicarmos pelo **vectordT**) podemos ocupar o vetor **deriv** com a derivada no ponto.
Assim, com estes dois *arrays* e a necessária variável do tempo, a funçãi **getGlobalCatmullRomPoint** permite-nos obter as coordenadas do p´roximo ponto da curva para esse dado valor **t** de tempo.

```
void getCatmullRomPoint(float t, float* p0, float* p1, float* p2, float* p3, float* pos, float* deriv) {
    int i;
    // catmull-rom matrix
    float m[4][4] = { {-0.5f,  1.5f, -1.5f,  0.5f},
                      { 1.0f, -2.5f,  2.0f, -0.5f},
                      {-0.5f,  0.0f,  0.5f,  0.0f},
                      { 0.0f,  1.0f,  0.0f,  0.0f} };

    for (i = 0; i < 3; i++) {
        float vectorA[4] = { 0,0,0,0 };
        float vectorP[4] = { p0[i],p1[i],p2[i],p3[i] };
        multMatrixVector(*m, vectorP, vectorA);

        float vectorT[4] = { powf(t,3),powf(t,2),powf(t,1),1 };
        float vectordT[4] = { 3 * powf(t,2), 2 * t, 1, 0 };

        pos[i] = (vectorT[0] * vectorA[0]) + (vectorT[1] * vectorA[1]) + (vectorT[2] * vectorA[2]) + (vectorT[3] * vectorA[3]);
        deriv[i] = (vectordT[0] * vectorA[0]) + (vectordT[1] * vectorA[1]) + (vectordT[2] * vectorA[2]) + (vectordT[3] * vectorA[3]);

    }
}
```

Figura 3.3: *getCatmullRompoint*

```
/// given  global t, returns the point in the curve
void getGlobalCatmullRomPoint(float gt, float* pos, float* deriv, std::vector<std::vector<float>> basePoints) {
    int POINT_COUNT = basePoints.size();
    float t = gt * POINT_COUNT; // this is the real global t
    int index = floor(t);  // which segment
    t = t - index; // where within  the segment

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

    getCatmullRomPoint(t,  p0: (float*)basePoints[indices[0]].data(),  p1: (float*)basePoints[indices[1]].data(),  p2: (float*)basePoints[indices[2]].data(),
                       p3: (float*)basePoints[indices[3]].data(), pos, deriv);
}
```

Figura 3.4: *getGlobalCatMullRomPoints*

3.3.2 Desenho das curvas

Com base nas funções apresentadas acima, foi implementada a função **renderCatmullRomCurve** para desenhar a curva pretendida. O ***tesselation level*** como 100.

```
void renderCatmullRomCurve(std::vector< std::vector< float >> basePoints) {

    // draw curve using line segments with GL_LINE_LOOP
    float pos[3], deriv[3];

    glBegin(GL_LINE_LOOP);
    for (int i = 0; i < tesselation; i++) {
        getGlobalCatmullRomPoint( gt: i / tesselation , pos, deriv, basePoints);
        glVertex3f( x: pos[0],  y: pos[1],  z: pos[2]);
    }
    glEnd();
}
```

Figura 3.5: *renderCatmullRomCurve*

3.4 Leitura do novo formato *XML*

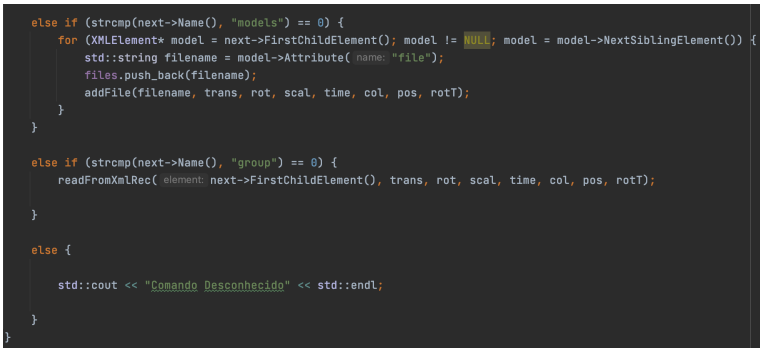
Dado que as transformações geométricas só podem existir dentro de elementos *group* e são aplicadas a todos os modelos e subgrupos desse elemento, a leitura ficheiro *XML* foi feita de forma recursiva (conceito de herança).

```
void readFromXmlRec(XMLElement* element , std::vector<std::vector<std::vector<float>>> trans, std::vector<float> rot,
                std::vector<float> scal, std::vector<float> time, std::vector<float> col ,int pos,float rotT) {
    for (XMLElement* next = element; next != NULL; next = next->NextSiblingElement()) {


        if (strcmp(next->Name(), "colour") == 0) {
            if (next->FindAttribute( name: "red")) {
                col[0] = atof(next->FindAttribute( name: "red")->Value());
            }
            if (next->FindAttribute( name: "green")) {
                col[1] = atof(next->FindAttribute( name: "green")->Value());
            }
            if (next->FindAttribute( name: "blue")) {
                col[2] = atof(next->FindAttribute( name: "blue")->Value());
            }
        }


        else if (strcmp(next->Name(), "rotate") == 0) {
            if (next->FindAttribute( name: "time")) {
                rotT = atof(next->FindAttribute( name: "time")->Value());
            }

            if (next->FindAttribute( name: "X")) {
                rot[0] += atof(next->FindAttribute( name: "X")->Value());
            }
            if (next->FindAttribute( name: "Y")) {
                rot[1] += atof(next->FindAttribute( name: "Y")->Value());
            }
            if (next->FindAttribute( name: "Z")) {
                rot[2] += atof(next->FindAttribute( name: "Z")->Value());
            }
```

Figura 3.6: *readXMLRec1*

```
else if (strcmp(next->Name(), "translate") == 0) {

    time[pos+1] = atof(next->FindAttribute( name: "time")->Value()) > 0 ? atof(next->FindAttribute( name: "time")->Value()) : 1000;

    std::vector<std::vector<float>> tstate;
    std::vector<float> points;

    for (auto tag = next->FirstChildElement(); tag != NULL; tag = tag->NextSiblingElement()) {

        std::vector<float> points = { tag->FindAttribute( name: "X") ? (float)atof(tag->FindAttribute( name: "X")->Value()) : 0,
                    tag->FindAttribute( name: "Y") ? (float)atof(tag->FindAttribute( name: "Y")->Value()) : 0,
                    tag->FindAttribute( name: "Z") ? (float)atof(tag->FindAttribute( name: "Z")->Value()) : 0 };

        tstate.push_back(points);
    }
    trans.push_back(tstate);
    pos++;
}

else if (strcmp(next->Name(), "scale") == 0) {

    if (next->FindAttribute( name: "X")) {
        scal[0] *= atof(next->FindAttribute( name: "X")->Value());
    }
    if (next->FindAttribute( name: "Y")) {
        scal[1] *= atof(next->FindAttribute( name: "Y")->Value());
    }
    if (next->FindAttribute( name: "Z")) {
        scal[2] *= atof(next->FindAttribute( name: "Z")->Value());
    }
}
```

Figura 3.7: *readXMLRec2*



```
    else if (strcmp(next->Name(), "models") == 0) {
        for (XMLElement* model = next->FirstChildElement(); model != NULL; model = model->NextSiblingElement()) {
            std::string filename = model->Attribute( name: "file");
            files.push_back(filename);
            addFile(filename, trans, rot, scal, time, col, pos, rotT);
        }
    }

    else if (strcmp(next->Name(), "group") == 0) {
        readFromXmlRec( element: next->FirstChildElement(), trans, rot, scal, time, col, pos, rotT);

    }

    else {

        std::cout << "Comando Desconhecido" << std::endl;

    }
}
```

Figura 3.8: *readXMLRec3*

Capítulo 4

Alterações *XML*

Em relação à fase anterior, houveram algumas alterações no ficheiro *XML* que necessitam de ser realçadas.
Foram adicionadas *translações* para todas os planetas e para todas as luas e também para o *teapot* onde, em cada uma dessas translações há um atributo **time** que representa o tempo, em milissegundos, que cada um demora a percorrer a sua órbita. Para além disso, nessas translações, estão presentes os pontos necessários para o desenho dessa mesma órbita.

```
<translate time="1000">
    <point X="29.4" Y="0.0" Z="0.0"/>
    <point X="27.30919" Y="0.0" Z="-10"/>
    <point X="20" Y="0.0" Z="-19.78991"/>
    <point X="10" Y="0.0" Z="-25.39016"/>
    <point X="0.0" Y="0.0" Z="-27"/>
    <point X="-10" Y="0.0" Z="-25.39016"/>
    <point X="-20" Y="0.0" Z="-19.78991"/>
    <point X="-27.30919" Y="0.0" Z="-10"/>
    <point X="-29.4" Y="0.0" Z="0.000000"/>
    <point X="-27.30919" Y="0.0" Z="10"/>
    <point X="-20" Y="0.0" Z="19.78991"/>
    <point X="-10" Y="0.0" Z="25.39016"/>
    <point X="0" Y="0.0" Z="27"/>
    <point X="10" Y="0.0" Z="25.39016"/>
    <point X="20" Y="0.0" Z="19.78991"/>
    <point X="27.30919" Y="0.0" Z="10"/>
</translate>
```

Figura 4.1: *Exemplo de um translate no XML*

Também foi acrescentado um atributo *colour* para cada planeta, lua e órbita que define a cor de cada um desses elementos.

```
<colour red="33" green="154" blue="143" />
```

Figura 4.2: *Exemplo de um colour no XML*

Capítulo 5

Resultados: Sistema Solar

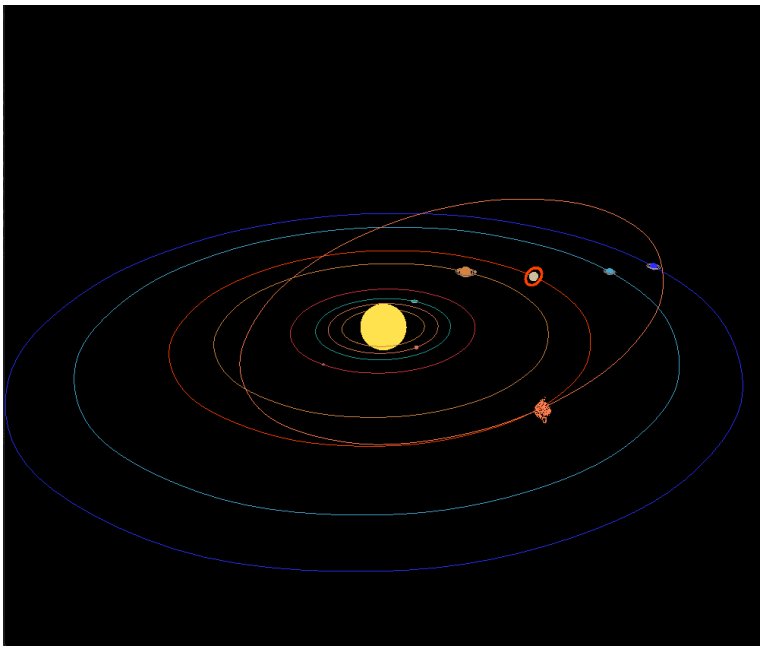


Figura 5.1: *Sistema Solar*

Capítulo 6

Conclusão

Nesta fase, foi possível introduzir o conceito das curvas de *Catmull-Rom* que, para além de facilitar no cálculo dos pontos das animações das translações, também contribuíram para desenhar as órbitras dos planetas e das suas respetivas luas.
Foi introduzido um VBO com o objetivo de melhorar a eficiência do projeto e também foram utilizados *patches de Bezier* para a implementação do *teapot*.
Assim, foi concluída mais uma fase para a criação do Sistema Solar.