Computação Gráfica

LEI - 3º ano - 2º semestre
Universidade do Minho

Trabalho Prático

Fase 3

Grupo 10

Trabalho realizado por:	*Número*
Duarte Moreira	A93321
Lucas Carvalho	A93176
Manuel Novais	A89512

Braga, 30 de abril de 2022

Índice

1. Introdução

O presente relatório foi realizado no âmbito da Unidade Curricular de Computação Gráfica e tem como objetivo expor detalhadamente as decisões e abordagens tomadas pelo grupo durante a terceira fase de desenvolvimento do projeto.

Este trabalho, dividido em 4 fases, tem como objetivo final a criação de um mini cenário baseado em gráficos 3D. Nesta terceira etapa é-nos pedido a alteração de ambas as aplicações, tanto do **generator** como do **engine**. Aqui, a nova aplicação **generator** deverá ser capaz de criar um novo tipo de modelo baseado em *Bezier patches*. Já o **engine** deverá ser atualizado de forma a conseguir suportar e lidar com as curvas de *Catmull-Rom* com o objetivo de criar animações.

2. Gerador

Assim sendo, o **generator** foi ajustado de maneira a suportar comandos do tipo:

```
./generator bezier teapot.patch 10 teapot.3d
```

Onde o primeiro argumento - bezier - nos indica que será necessário recorrer aos *Bezier patches*, o segundo - teapot.patch - será o nome do ficheiro .patch onde se encontra uma descrição de um conjunto de *Bezier patches*, o terceiro - 10 - o nível de *tesselation* e o último - teapot.3d - que será o nome do ficheiro .3d a criar com os pontos calculados.

2.1 Bezier Patches

O primeiro passo a ser tomado foi analisar o formato dos ficheiros .patch para assim decidirmos como ler o respetivo ficheiro e que estrutura de dados utilizar. Ao terminar a leitura do mesmo possuímos em mão os seguintes dados:

```
int patches;        // numero de patches
int** indices;      // indices dos control points
int numcp;          // numero de control points
float** cp;         // control points
```

Temos agora condições para calcular os pontos que irão dar origem às superfícies de *Bezier* fornecidas e posteriormente as escrever no ficheiro .3d. Para exercer este cálculo foi realizado para cada *patch*, o aninhamento de dois ciclos controlados pelas variáveis u e v respetivamente, que variam entre 0 e 1 e têm um incremento calculado a partir do nível de *tesselation* fornecido. Para cada iteração destes ciclos são calculados 4 pontos a partir da fórmula de *Bezier*.

```
float inc = 1.0 / tesselation;
float p1[3], p2[3], p3[3], p4[3];

for (int i = 0; i < patches; i++) {
    for (float u = 0; u < 1; u += inc) {
        for (float v = 0; v < 1; v += inc) {

            bezierPatches(     u,       v, indices[i], cp, p1);
            bezierPatches(     u, v + inc, indices[i], cp, p2);
            bezierPatches(u + inc,       v, indices[i], cp, p3);
            bezierPatches(u + inc, v + inc, indices[i], cp, p4);

            out << p4[0] << " " << p4[1] << " " << p4[2] << endl;
            out << p3[0] << " " << p3[1] << " " << p3[2] << endl;
            out << p2[0] << " " << p2[1] << " " << p2[2] << endl;

            out << p3[0] << " " << p3[1] << " " << p3[2] << endl;
```

```
                out << p1[0] << " " << p1[1] << " " << p1[2] << endl;
                out << p2[0] << " " << p2[1] << " " << p2[2] << endl;
            }
        }
    }
```

Para finalizar, falta analisar o algoritmo utilizado para o cálculo destes pontos. O algoritmo teve por base a fórmula de *Bezier* que é a seguinte:

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

Figura 2.1: Fórmula de *Bezier*.

Com isto, comecamos por obter os vetores **px**, **py** e **pz** através dos pontos de controlo e dos ìndices lidos do ficheiro .patch. Por sua vez, sabendo que a matriz **M** é igual à sua transposta M^T e seguindo a fórmula de *Bezier* começamos por multiplicar os parâmetros constantes (vetores **U** e **V** e matrizes **M** e M^T) e só no fim multiplicar pela matriz **P**.

```
float U[4] = {powf(u,3), powf(u,2), u, 1.0};
float V[4] = {powf(v,3), powf(v,2), v, 1.0};

float M[4][4] = {{-1.0f,  3.0f, -3.0f,  1.0f},
                 { 3.0f, -6.0f,  3.0f,  0.0f},
                 {-3.0f,  3.0f,  0.0f,  0.0f},
                 { 1.0f,  0.0f,  0.0f,  0.0f}};

float px[4][4], py[4][4], pz[4][4];
for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 4; j++) {
        int ind = indices[i * 4 + j];
        px[i][j] = cp[ind][0];
        py[i][j] = cp[ind][1];
        pz[i][j] = cp[ind][2];
    }
}

float  UM[4], MV[4], UMV[4], x[4], y[4], z[4];
multMatrixVector(  U,  *M,  UM);
multMatrixVector( *M,   V,  MV);
multMatrixVector( UM,  MV, UMV);
multMatrixVector(UMV, *px,   x);
multMatrixVector(UMV, *py,   y);
multMatrixVector(UMV, *pz,   z);

coord[0] = x[0];
coord[1] = y[0];
coord[2] = z[0];
```

3. Motor

Neste fase, o **engine** sofreu algumas alterações. A primeira consistiu em construir os modelos com o uso de VBO´s ao invés de utilizar *immediate mode*. Como forma de comparação de performance ambos os modos estão disponíveis. A próxima alteração realizada foi a utilização das curvas de *Catmull-Rom* tanto na translação como na rotação, com o objetivo de criar animações.

3.1 VBO's

Para a implementação dos VBO´s a *class* **Model** foi alterada de modo a conter uma instância da *class* **VBO** e uma função **drawVBO()**. Ou seja, para cada modelo lido do ficheiro **xml** é reservado um VBO. Depois disso, utiliza-se um *bool*, passado à função responsável por desenhar os modelos de um *group*, de forma a nos indicar se os modelos deveram ser construídos usando o VBO´s ou não.

```
VBO::VBO(vector<Ponto> pontos) {
        this->vertices = 1;
        this->verticeCount = pontos.size() * 3;
        this->prepareData(pontos);
}
void VBO::prepareData(vector<Ponto> pontos) {
        vector<float> coords;
        for (Ponto p : pontos) {
                coords.push_back(p.x);
                coords.push_back(p.y);
                coords.push_back(p.z);
        }
        // criar o VBO e copiar o vector para a memória gráfica
        ...
}
void VBO::draw() {
        glBindBuffer(GL_ARRAY_BUFFER, this->vertices);
        glVertexPointer(3, GL_FLOAT, 0, 0);
        glDrawArrays(GL_TRIANGLES, 0, this->verticeCount);
}
void Group::draw(bool vbo, bool eixos, bool curves, bool stop) {
        glPushMatrix();...
        for (Model* model : this->models) {
                if (vbo) model->drawVBO(eixos);
                else model->draw(eixos);
        }
        ...glPopMatrix();
}
```

3.2 Catmull-Rom

Para a criação das curvas de *Catmull-Rom* o primeiro ponto que temos de ter em consideração é a seguinte equação que nos dá a posição de um objeto que "caminha"ao longo de uma curva:

$$p(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

Figura 3.1: Equação da posição de um corpo numa curva de *Bezier*.

O segundo e último ponto são as coordenadas dos pontos que irão "formar"a pretendida curva. Estes pontos serão fornecidos através do ficheiro *xml* junto com as transformações geométricas **translate**. Com isto foi criada a seguinte *class* **Catmullrom**.

```
class Catmullrom {
private:
        int pointCount;     // nº de pontos do vetor p
        vector<Ponto> p;    // pontos que formam a curva
        float time;         // tempo para percorrer a curva em ms
        bool align;         // alinhamento
        float t, timePast;  // variaveis para controlo de tempo
        float pos[3];       // posicao atual na curva
        float deriv[3];     // derivadas - tangentes
public:
    void buildRotMatrix(float* x, float* y, float* z, float* m);
        void cross(float* a, float* b, float* res);
        void normalize(float* a);
        void multMatrixVector(float* m, float* v, float* res);
        void getCatmullRomPoint(float t, Ponto p0, Ponto p1, Ponto p2, Ponto p3,
                                float* pos, float* deriv);
        void getGlobalCatmullRomPoint(float gt, float* pos, float* deriv);
        Catmullrom(vector<Ponto> p, float time, bool align);
        void renderCatmullRomCurve();
        void runCatmullRomCurve(bool curves, bool stop);
};
```

Esta *class* é utilizada aquando da criação de uma transformação geométrica **translate** dependendo do formato em que esta se apresenta no ficheiro *xml*.

3.3 Translação

Como dito anteriormente, a partir de agora o formato de uma transformação geométrica **translate** poderá ser diferente apresentando um conjunto de pontos que irão auxiliar na construção das curvas mencionadas a cima. Para além disso é nos dado também um valor de **tempo**, que indicará o tempo que o objeto terá para efetuar uma volta na respetiva curva, permitindo assim trabalhar a sua velocidade, e também uma variável **align** que representa a orientação do corpo, ou seja, desenhar o corpo de forma a que este esteja sempre alinhado com a curva ou não.

Esta mudança, implicou a que fosse usada a nova *class* criada **Catmullrom** e também a um pequeno ajuste na leitura do ficheiro *xml*.

```
void parseTranslate(XMLElement* t, Transform* transform) {
        string value = t->FirstAttribute()->Name();
        if (value.compare("time") == 0) {
                float time = t->FloatAttribute("time");
                bool align = t->BoolAttribute("align");
                vector<Ponto> cp;
                t = t->FirstChildElement("point");
                for (; t != NULL; t = t->NextSiblingElement("point")) {
                        Ponto p;
                        p.x = t->FloatAttribute("x");
                        p.y = t->FloatAttribute("y");
                        p.z = t->FloatAttribute("z");
                        cp.push_back(p);
                }
                transform->setTranslate(new Translate(cp, time, align));
        }
        else {
                float x = t->FloatAttribute("x");
                float y = t->FloatAttribute("y");
                float z = t->FloatAttribute("z");
                transform->setTranslate(new Translate(x, y, z));
        }
}
```

Toda a implentação e utilização da *class* **Catmullrom** é baseado nos ensinamentos das aulas, tanto teóricas como práticas. O grupo decidiu apenas apresentar aqui o algoritmo que utilizou para calcular o valor global de **t** que é a variável que permite controlar a posição do objeto na curva de *Bezier*.

```
int time = glutGet(GLUT_ELAPSED_TIME);
int delta = time - this->timePast;
this->timePast = time;

this->t += (delta / this->time) * !stop;
```

Nota: a variável **stop** é um *bool* com o objetivo de parar toda a simulação caso o utilizador pretenda.

3.4 Rotação

De igual forma as transformações **rotate** também sofreram alterações no seu formato, podendo agora apresentar um tempo para o objeto respetivo realizar rotações sobre um eixo especifico. Para tal ajustou-se o *parser* da seguinte maneira:

```
void parseRotate(XMLElement* t, Transform* transform) {
        string value = t->FirstAttribute()->Name();
        float angulo;
        bool rotate;
        if (value.compare("angle") == 0) {
                angulo = t->FloatAttribute("angle");
                rotate = false;
        }
        else {
                angulo = t->FloatAttribute("time");
                rotate = true;
        }
        float x = t->FloatAttribute("x");
        float y = t->FloatAttribute("y");
        float z = t->FloatAttribute("z");
        transform->setRotate(new Rotate(rotate, angulo, x, y, z));
}
```

Sendo que, para além disso, a *class* **Rotate** apresenta agora um *bool rotate* para nos indicar se o objeto vai ter um "periodo de rotação"ou não e um *float timePast* como sendo uma variável de controlo de tempo. A implementação deste objetivo consistiu no seguinte:

```
void Rotate::run(bool stop) {
        if (this->rotate) {
                int time = glutGet(GLUT_ELAPSED_TIME);
                float delta = time - this->timePast;
                this->timePast = time;
                this->angulo += (360.0 / this->angulo) * delta * !stop;
        }
        glRotatef(this->angulo, this->x, this->y, this->z);
}
```

4. Resultados Obtidos

4.1 Testes

Apresentam-se a seguir os resultados obtidos com as nossas aplicações de acordo com os ficheiros de testes fornecidos pela equipa docente. De notar que para obter os resultados semelhantes aos fornecidos pela equipa docente foi necessário comentar a seguinte linha de código:

```
glEnable(GL_CULL_FACE);
```

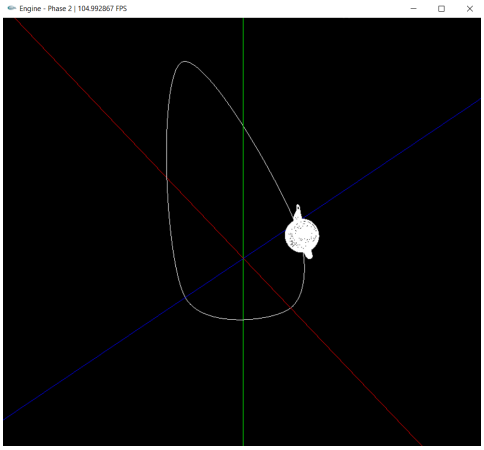

Figura 4.1: Resultado de *test_3_1.xml*.

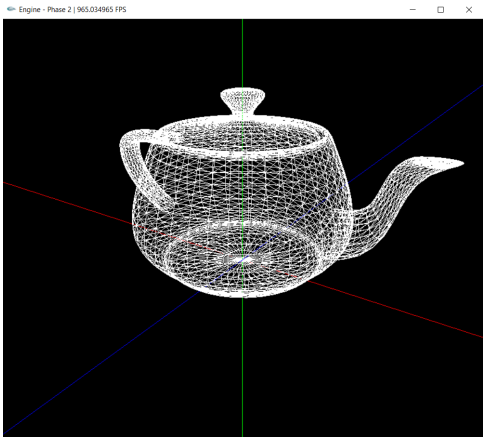

Figura 4.2: Resultado de *test_3_2.xml*.

4.2 Cenários de demonstração

De seguida expõem-se o cenário de demonstração pretendido para esta fase: um sistema solar dinâmico e um cometa com uma órbita definida através das curvas de *Catmull-Rom*.

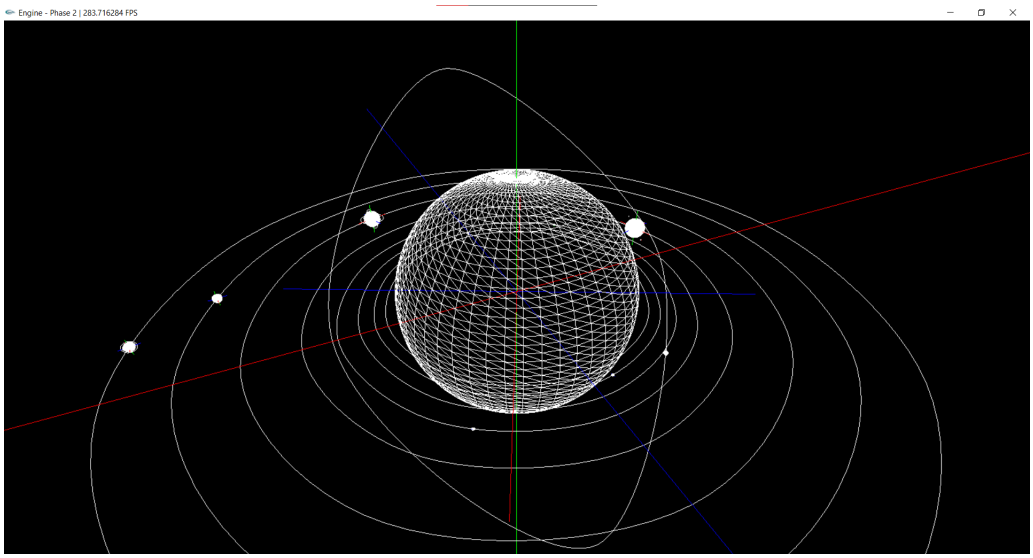

Figura 4.3: Sistema solar.

```
C:\Users\duart\Desktop\CG\trabpratico\fase3\proje
Vendor: Intel
Renderer: Intel(R) UHD Graphics 630
Version: 4.6.0 - Build 27.20.100.8681

Use mouse left button to move arround
Use mouse right button to zoom in/out
Controls:
'z' - Show/Hide central axis.
'x' - Show/Hide secundary axis.
'c' - Change polygon mode.
'v' - Enable/Disable VBO mode.
'p' - Pause/Unpause the solar system.

Current state:
        -Showing central axis.
        -Hiding secundary axis.
        -Polygon mode set to GL_LINE.
        -Using VBO mode.
        -Solar System runnning.

Current state:
        -Hiding central axis.
        -Hiding secundary axis.
        -Polygon mode set to GL_LINE.
        -Using VBO mode.
        -Solar System runnning.

Current state:
        -Hiding central axis.
        -Hiding secundary axis.
        -Polygon mode set to GL_LINE.
        -Using immediate mode.
        -Solar System runnning.

Current state:
        -Hiding central axis.
        -Hiding secundary axis.
        -Polygon mode set to GL_FILL.
        -Using immediate mode.
        -Solar System runnning.

Current state:
        -Hiding central axis.
        -Hiding secundary axis.
        -Polygon mode set to GL_FILL.
        -Using immediate mode.
        -Solar System paused.
```

Figura 4.4: Registo de atividade.

5. Extras

Neste capítulo o grupo conseguiu implementar algumas funcionalidades extra enumeradas em baixo.

1. Possibilidade de colocar em "pausa"o sistema solar, parando assim a movimentação e rotação de todos os modelos incluídos no sistema.
2. Possibilidade de "esconder"as órbitas dos planetas permitindo assim uma menor poluição visual.
3. Capacidade de "esconder"os eixos das coordenadas associados a cada modelo do sistema, assim como o eixo central.
4. Capacidade de intercalar o desenho dos modelos entre *immediate mode* e VBO's.

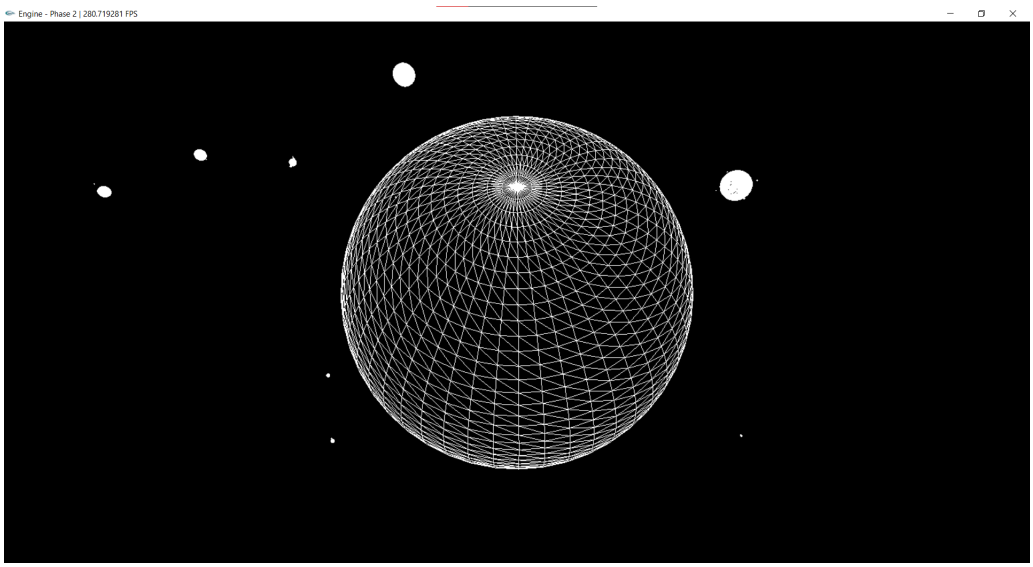

Figura 5.1: Sistema solar sem órbitas e eixos.

6. Conclusão

Concluindo, o grupo ficou satisfeito com o resultado obtido uma vez que foi implementado tudo que nos foi pedido no enunciado.

Salienta-se a importância das curvas de *Catmull-Rom*, dado que o seu uso permite a criação de animações que, de certa forma, são essenciais na criação de um projecto que tenha por base o sistema solar, como é o caso.

Com um pouco mais de tempo a equipa gostaria de ter explorado mais funcionalidades extras, como por exemplo a possibilidade de acelerar/desacelerar os tempos de animação e também um controlo de câmara em *first person* que traria ao utilizador uma melhor experiência.

Por último, concluída esta parte, o grupo mostra-se ansioso por começar a trabalhar na quarta e última fase do projeto de forma a conseguir implementar todas as funcionalidades que ficaram por fazer.