Computação Gráfica - 3.ª Fase

Dinis Lourenço Gomes
a87993@alunos.uminho.pt

Rui Miguel Gonçalves Dias
a89480@alunos.uminho.pt

Universidade do Minho - Departamento de Informática

1 Introdução

Nesta fase do trabalho a aplicação **Generator** tem que ter a capacidade de criar um novo tipo de modelo utilizando as *patches de Bezier*. Sendo que este recebe o nome do ficheiro, onde a informação dos pontos de *Bezier* se encontram, e o respectivo nível de detalhe. Após ser gerado, o ficheiro irá conter a lista de triângulos necessários para desenhar a superfície do modelo.

Por outro lado será agora necessário estender as translações e rotações na aplicação **Engine**. A extensão da translação não só terá um conjunto de pontos para definir uma curva cúbica de *Catmull-Rom*, como também o número de segundos que a animação irá durar. O objectivo é então fazer animações baseadas nestas curvas.

A extensão da rotação o atributo *angle* será agora substituído pelo atributo *time*, o qual corresponde ao número de segundos necessários para uma rotação de 360º no seu respectivo *axis*.

Há que notar que as extensões não eliminam a possibilidade de fazer uma transformação geométrica estática.

2 Implementação

Os tópicos seguintes vão descrever o raciocínio utilizado durante cada secção do trabalho.

2.1 Generator

2.1.1 Estruturas utilizadas

Esta estrutura contém a informação relativa a uma *patch de Bezier*, ou seja, é composto pelo conjunto de pontos que a representa, que são obtidos a partir dos índices associados aos pontos que estão representados no ficheiro *.patch*.

```
class Patch {
    public:
        vector<Point> points;    //Pontos da patch
        vector<int> indices;    //Indices dos respetivos pontos
};
```

A seguinte estrutura contém a informação de um ponto individual (x, y, z) neste contexto utilizado para guardar a informação relativa a um ponto das *patches de Bezier*.

```
class Point {
  public:
      float p[3]; //Coordenadas dos pontos

    // Default constructor
    Point(){
        p[0] = 0;
        p[1] = 0;
        p[2] = 0;
    }

    Point(float xx, float yy, float zz){
        p[0] = xx;
        p[1] = yy;
        p[2] = zz;
    }
};
```

2.1.2 Parsing dos ficheiros *.patch*

O ficheiro *.patch* segue a seguinte estrutura:

- Número de *patches* que o ficheiro contém, ou seja o número de linhas que se seguem com informação para cada *patch*.
- Cada linha relativa a uma *patch* contém os índices dos pontos que vão estar associados a *patch*.
- Número de pontos que o ficheiro contém, ou seja o número linhas que se seguem com a informação para cada ponto específico.
- Cada linha irá conter três componentes, referentes a um ponto (*x, y, z*).

Tendo em conta esta estrutura o *parser* desenvolvido limita-se a ler o número de *patches* que o modelo terá, de seguida, itera sobre cada linha dentro do limite pré-estabelecido de *patches* e para cada uma guarda na estrutura referida anteriormente, **Patch**, os índices dos pontos correspondentes. A *patch* é então guardada num vector.

Após os índices terem sido associados, o *parser* lê a linha seguinte que contém o número de pontos que o ficheiro contém e itera sobre as próximas linhas guardando cada ponto dentro da estrutura referida anteriormente, **Point**, sendo posteriormente guardado num vector.

O vector de *Patches* é então percorrido e para cada *patch* são copiados os pontos presentes no vector temporário de pontos, que correspondem aos índices especificados na variável indices da *Patch*.

2.1.3 Input Para Gerar um File *.3d*

O comando genérico seguinte corresponde ao input necessário para gerar um ficheiro *.3d* a partir de um ficheiro *.patch*.

./generator.exe patch "n.ºTesselações" "file.patch" "nome_do_ficheiro_final"

2.1.4 Geração de um Modelo Baseado nas *Patches de Bezier*

A geração do modelo é feita a partir da seguinte fórmula. É necessário referir que para cada *patch* o par (u,v), que representa um ponto da superfície a criar, varia sendo o número de combinações do par (u,v) igual ao quadrado do número de tesselações.

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

$$M = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{2}$$

Estes pontos são então colocados no ficheiro *.3d* de forma a criar triângulos que representem as superfícies e consequentemente o modelo.

2.2 Engine

2.2.1 Estruturas Utilizadas

As estruturas referidas nos itens seguintes são estruturas adicionadas e/ou remodeladas para abarcar as necessidades desta fase.

Point

A seguinte estrutura contém a informação de um ponto individual (x, y, z) e é posteriormente guardada num vector na estrutura **TranslateT**.

```
class Point {
  public:
      float p[3]; //Coordenadas dos pontos

    // Default constructor
    Point(){
        p[0] = 0;
        p[1] = 0;
        p[2] = 0;
    }

    Point(float xx, float yy, float zz){
        p[0] = xx;
        p[1] = yy;
        p[2] = zz;
    }
};
```

TranslateT

Esta classe é a extensão da classe de **Translate**, adaptada para trabalhar com os pontos das *patches de Bezier* e o respectivo tempo de duração da animação.

```
#include "point.cpp"
class TranslateT {
  public:
    float time; //Time
    vector<Point> points;
    Point* pointmatrix;
    float y[3];

    // Default constructor
    TranslateT(){
        time = 0;
        y[0] = 0;
        y[1] = 1;
        y[2] = 0;
    }

    void createMatrix() {
        pointmatrix = (Point*)malloc(points.size() * sizeof(Point));
        int j = 0;
        for (vector<Point>::const_iterator i = points.begin(); i != points.end();
     i++)
        {
            pointmatrix[j] = *i;
            j++;
        }
    }
};
```

RotateT

Esta classe é a extensão da classe de **Rotate**, adaptada para trabalhar com tempo, que corresponde ao número de segundos que uma rotação de 360º irá demorar de acordo ao respectivo *axis*.

```
class RotateT {
    public:
        float time,             // Time to complete one rotation
        axisX, axisY, axisZ;    // Axis

        RotateT(){
            time = 0;
            axisX = 0;
```

```
            axisY = 0;
            axisZ = 0;
        }
};
```

Group

O **Group** mantém a sua função da fase anterior, no entanto, agora contém as extensões necessárias **TranslateT** e **RotateT**. Tal como na fase anterior, em cada grupo são apenas aplicadas as transformações geométricas presentes no vector **order** pela ordem em que aparecem.

```
using namespace std;

class Group{
    public:
        // Componente de  Translao do grupo
        Translate trans;

        // Componente de  Translao tempo do grupo
        TranslateT transT;

        // Componente de  Rotao do grupo
        Rotate rot;

        // Componente de  rotao tempo do grupo
        RotateT rotT;

        // Componente de Escala do grupo
        Scale scale;

        // Vector de modelos dentro do grupo
        vector<Model> models;

        // SubGrupos dentro do grupo
        vector<Group> subGroups;

        // Order Vector - pode conter T | TT | R | RT | S
        vector<string> order;

        // Construtor
        Group(Translate t, TranslateT tt, Rotate r, RotateT rt, Scale s, vector<
    Model> ms, vector<Group> sgs, vector<string> odr) {
            trans = t;
            transT = tt;
            rot = r;
            rotT = rt;
            scale = s;
            models = ms;
            subGroups = sgs;
            order = odr;
        }
};
```

2.2.2 Parsing do XML

O grupo optou novamente por utilizar a biblioteca *tinyxml2* para fazer o parsing dos ficheiros de XML. Havendo agora duas estruturas novas, extensões das estruturas **Translate** e **Rotate** foram feitas funções de parsing para estas novas estruturas sendo então necessário alterar a função que trata do parse no grupo para fazer a diferenciação entre translação/rotação, dependente do tempo ou estática.

```
TranslateT parseTransT(XMLElement* node) {
    // Temp string holder
    const char* output;

    // Vector de pontos
    vector<Point> points;

    // Point temporario
    Point p;

    // Temp Translate
    TranslateT tt = TranslateT();

```

```
    // Se exisitr
    if (node) {
        // Time
        output = node->Attribute("time");
        if (output) {
            tt.time = stof(output);
        }

        // Points
        // Primeiro model
        XMLElement* tempPoint = node->FirstChildElement("point");

        while (tempPoint) {
            output = tempPoint->Attribute("X");
            if (output) {
                p.p[0] = stof(output);
            }

            output = tempPoint->Attribute("Y");
            if (output) {
                p.p[1] = stof(output);
            }

            output = tempPoint->Attribute("Z");
            if (output) {
                p.p[2] = stof(output);
            }

            // Criar um model e associar coordenadas e filename
            points.push_back(p);

            // Avança para o model seguinte
            tempPoint = tempPoint->NextSiblingElement();
        }

        tt.points = points;
    }

    tt.createMatrix();

    return tt;
}
```

Listing 1: Função de parse de uma Translação dependente de tempo

Podemos observar no código acima que a informação dos pontos é obtida a partir de um ciclo que itera sobre cada componente <*ponto*> que é guardada numa estrutura temporária **Point** p e posteriormente guardado na estrutura **TranslateT** tt.

```
RotateT parseRott(XMLElement* node) {
    // Temp string holder
    const char* output;

    // Temp Rotate
    RotateT rt = RotateT();

    //Se existir
    if (node) {
        output = node->Attribute("time");
        if (output) {
            rt.time = stof(output);
        }

        output = node->Attribute("axisX");
        if (output) {
            rt.axisX = stof(output);
        }

        output = node->Attribute("axisY");
        if (output) {
            rt.axisY = stof(output);
        }

        output = node->Attribute("axisZ");
        if (output) {
            rt.axisZ = stof(output);
        }
    }
```

```
    return rt;
}
```

Listing 2: Função de parse de uma Rotação dependente de tempo

O código acima relativo ao parsing da **RotaçãoT** é analogo à **Rotação** sendo a componente *angle* substituída pela componente *time*.

```

        // Ciclo que percorre os child elements ate chegar aos models
        while (fchild && flag != 0) {
            // Passagem do nome do node para string para comparação
            nodename = fchild->Name();
            string stemp = nodename;

            // verifica se não e um model ou um grupo
            if (stemp != "models" && stemp != "grupo") {
                // Se for um translate
                if (stemp == "translate") {
                    if (fchild->Attribute("time") == NULL) {
                        // Executa o parsing do translate
                        t = parseTrans(fchild);

                        // Coloca Translate no vector de ordem
                        order.push_back("T");
                    }

                    else {
                        // Executa o parsing do translateT
                        tt = parseTransT(fchild);

                        // Coloca TranslateT no vector de ordem
                        order.push_back("TT");
                    }

                }

                // Se for um rotate
                else if (stemp == "rotate") {
                    if (fchild->Attribute("time") == NULL) {
                        // Executa o parsing do rotate
                        r = parseRot(fchild);

                        // Coloca Rotate no vector de ordem
                        order.push_back("R");
                    }

                    else {
                        // Executa o parsing do rotateT
                        rt = parseRott(fchild);

                        // Coloca RotateT no vector de ordem
                        order.push_back("RT");
                    }
                }
```

Listing 3: Mudança feita na *parseGroup* para permitir distinção entre TGs dependentes de tempo e estáticas

A distinção é feita procurando pelo atributo *time* do nodo de XML de translação/rotação, caso exista executam-se as funções de *parse* respectivas.

2.2.3 Curvas de *Catmull-Rom*

Utilizando como base o *exercício 8* focado nas curvas de *Catmull-Rom* juntamente com a seguinte fórmula o grupo implementou a **TranslaçãoT**, que se resume a uma translação com parâmetros correspondentes a *P(t)*.

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

O grupo derivando a fórmula anterior implementou a **RotateT**, feita através da função **glMultMatrix** utilizando a matriz de rotação **M**.

$$P'(t) = \begin{bmatrix} 3t^2 & 2t & 1 & 0 \end{bmatrix} \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} \tag{4}$$

Para construir a matriz de rotação, M, é inicialmente criado um vector $\vec{Y_0} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$ associado à translação, que ao fim de cada iteração é substituído com o novo valor de Y. Depois da criação deste vector são aplicadas as seguintes fórmulas, posteriormente todos os vectores consequentes são normalizados antes de serem usados para a criação da matriz de rotação

$$\vec{X_i} = P'(t) \tag{5}$$

$$\vec{Z_i} = X_i \times \vec{Y}_{i-1} \tag{6}$$

$$\vec{Y_i} = \vec{Z_i} \times \vec{X_i} \tag{7}$$

$$M = \begin{bmatrix} X_x & Y_x & Z_x & 0 \\ X_y & Y_y & Z_y & 0 \\ X_z & Y_z & Z_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{8}$$

3 Funcionalidades Extra

Uma das quatro funcionalidades extra implementadas foi a possibilidade de parar o movimento da *scene*. Para esta foram criadas duas variáveis globais, uma que guarda o tempo em pausa e outra que guarda o tempo em movimento, a soma destas será, num único instante de cada iteração, igual ao tempo total decorrido desde o início da aplicação.

A segunda funcionalidade foi a opção de apresentar ou não as trajectórias dos movimentos feitos pelos *groups*, para a qual foi criada uma variável global que habilita o desenho destas trajectórias, desenhadas por uma função criada para esse efeito.

Outra funcionalidade implementada foi a possibilidade de se poder mudar o formato de apresentação dos modelos, podendo este ser *FILL*, *LINE* ou *POINT*.

A última funcionalidade é relacionada à câmara. Foi implementado o controlo desta através de *mouse movements* e também a possibilidade da câmara focar noutro ponto.

4 Sistema Solar com as Novas Implementações

Na secção seguinte o grupo vai apresentar o Sistema Solar com as novas implementações. De modo a demonstrar a rotação dos planetas o grupo optou por tirar vários *prints*, no entanto em anexo com os restantes ficheiros do projecto irá ser enviado um GIF.

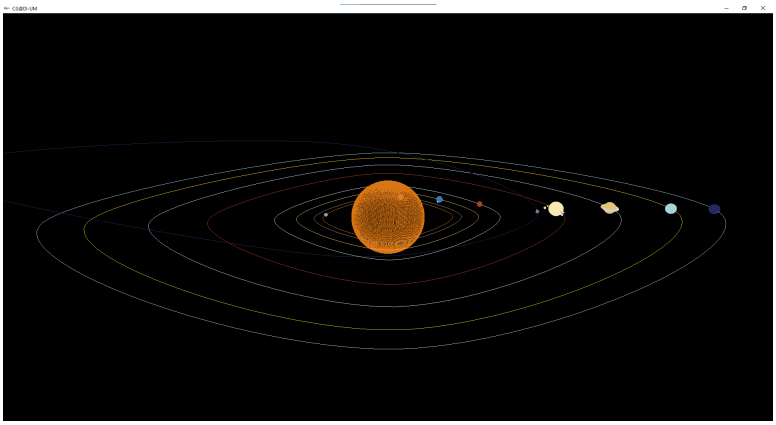

Figure 1: Frame 1

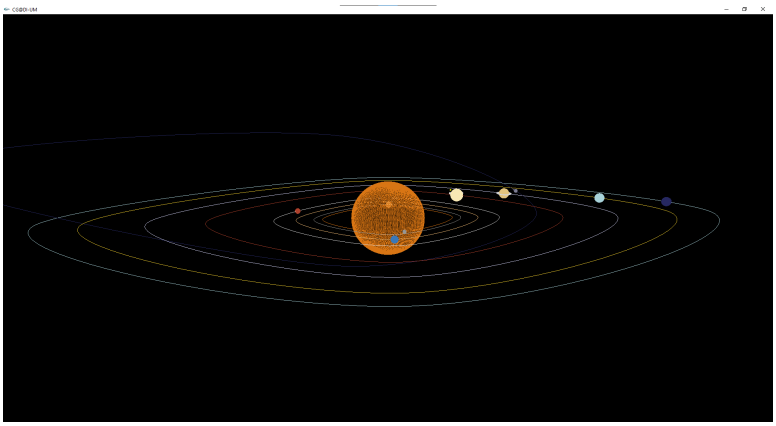

Figure 2: Frame 2

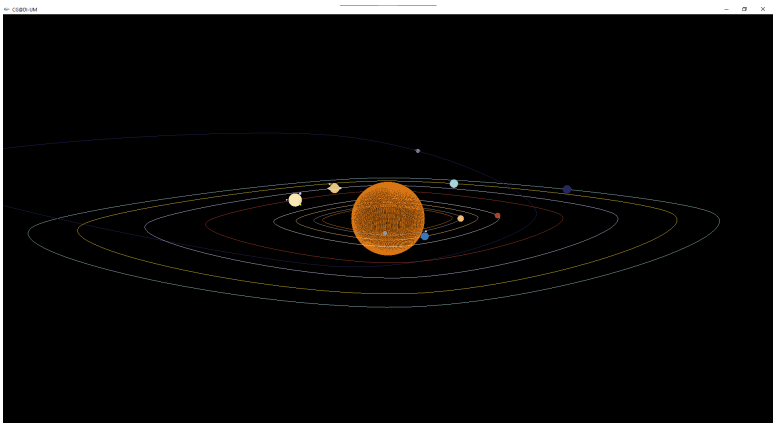

Figure 3: Frame 3

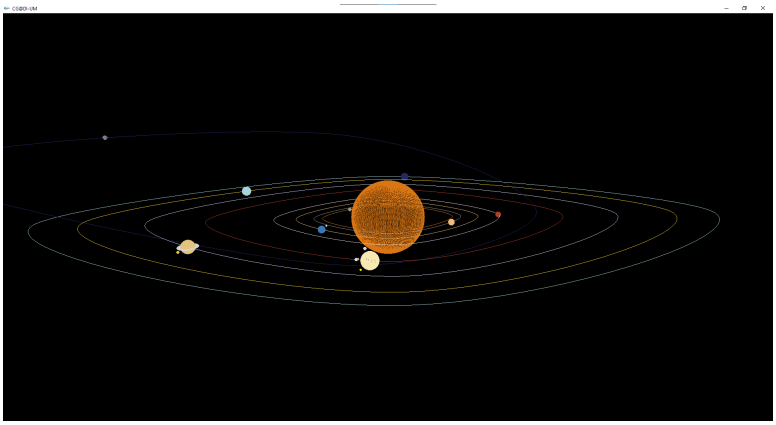

Figure 4: Frame 4

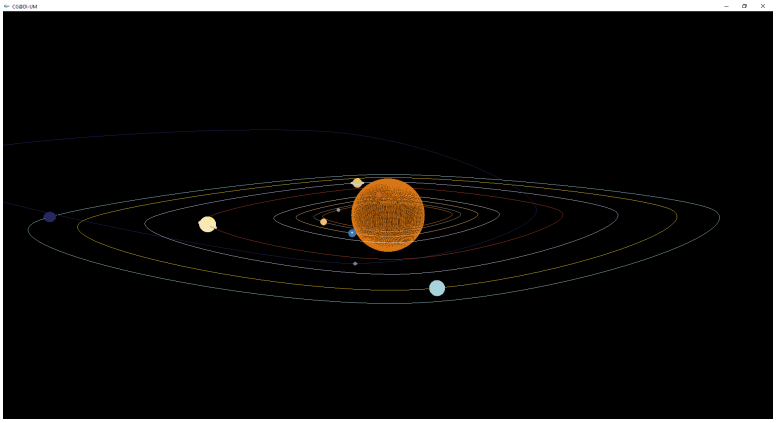

Figure 5: Frame 5