Universidade do Minho
Escola de Engenharia

Universidade do Minho
Mestrado Integrado em Engenharia Informática

Computação Gráfica

Projeto Computação Gráfica
3ª Fase

4 Maio 2020

Ana Margarida Campos
(A85166)

Ana Catarina Gil
(A85266)

Tânia Rocha
(A85176)

Contents

1 Introdução **4**

1.1 Contextualização 4

1.2 Resumo do trabalho desenvolvido 4

2 Trabalho desenvolvido nesta fase **5**

2.1 Patches de Bezier 5

2.1.1 Desenvolvimento do Teapot 5

2.2 Vbos 7

2.3 Catmull Rom Curves 8

2.3.1 Implementação Catmull-Rom 8

2.3.2 Translação 10

2.3.3 Órbitas 11

2.4 Resultado final do Sistema Solar 12

3 Conclusão **13**

List of Figures

2.1 Polinómio de Bernstein 5
2.2 Cálculo de cada ponto 6
2.3 Fórmula para calcular cada ponto 6
2.4 Função drawVBOs 7
2.5 Curva derivada de 4 pontos 8
2.6 Matrizes de Calculo P 8
2.7 Matriz M 8
2.8 Pontos indexados 9
2.9 Matriz M 9
2.10 Construção da matriz A 9
2.11 Polinomios T e T' 10
2.12 Classe Translate 10
2.13 Calculo das Curvas CatMull 10
2.14 Calculo dos pontos para o Translate 11
2.15 Resultado gráfico do Sistema Solar 12

Introdução

1.1 Contextualização

O presente relatório foi elaborado no âmbito da unidade curricular de Computação Gráfica de maneira a explicar o trabalho desenvolvido na terceira fase do projeto prático. Com o desenvolvimento do trabalho temos vindo a aproximar-nos cada vez mais de um Sistema Solar semelhante ao real.

Os principais objetivos nesta 3ª fase do trabalho foram o desenvolvimento de modelos baseados em *patches* de *Bezier*, nomeadamente a trajetória de um cometa utilizando os pontos de controlo fornecidos do *teapot*, a extensão das transformações geométricas translação e rotação a partir das curvas de *Catmull-Rum* e, por fim, a passagem do modo de desenho das figuras para VBOs.

1.2 Resumo do trabalho desenvolvido

De maneira a incorporar todos os objetivos desta fase no projeto foi necessária a alteração de vários elementos relativas ao trabalho da segunda fase, nomeadamente o *generator*, onde foram acrescentadas as funções relativas à criação dos *patches* de *Bezier*, e o *engine*, onde foram elaboradas as funções relatvias às curvvas de *Catmull-Rom*, bem como alterado o modo de renderização das figuras para a opção de uso de VBOs ou o uso do método anteriormente utilizado nas outras fases.

Trabalho desenvolvido nesta fase

2.1 Patches de Bezier

O programa *Generator* passou a suportar novas primitivas com recurso a Bezier Patches. Os argumentos necessários para o seu desenvolvimento vão ser do tipo *"InputFile Tesselation OutputFile"*. O ficheiro de input é o nome do ficheiro onde se encontram os patches necessários para o desenho de uma determinada primitiva. A tesselagem representa o valor da precisão com que a primitiva irá ser desenhada e o ficheiro output diz respeito ao nome do ficheiro onde serão guardados os vétices calculados. Relativamente ao ficheiro input este terá de possuir um formato específico, onde a primeira linha contém o número de patches seguido dos índices de cada patches e o número de pontos de controlo, também ele seguido por todos os pontos(um por cada linha) do tipo *x,y,z*.

2.1.1 Desenvolvimento do Teapot

Para o desenvolvimento do Teapot foram criadas três funções distintas e complementares.
A função *parsing* é responsável pela leitura dos dados presentes no ficheiro *teapot.patch* (disponibilizados pelos docentes da cadeira) e armazená-los em dois arrays distintos, um referente aos patches e o outro referentes ao pontos de controlo.
Como função principal temos a *bezier*, que após chamar a função parsing vai iterar pelo número de patches e pela tesselation bidimensionalmente, calculando os pontos que geram o teapot. Para fazer o cálculo de cada ponto utilizamos uma nova função extra, *pontoBezier*, que recebe o respetivos patch e níveis de tesselation .
Nesta função são criadas inicialmenente dois *polinómios de Bernstein* para ambas as dimenões u e v.

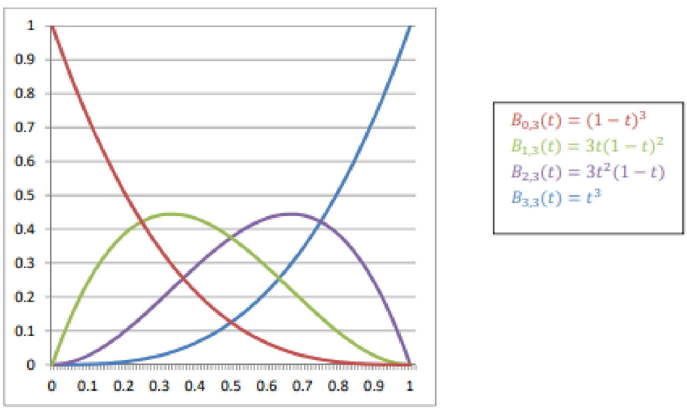

Figure 2.1: Polinómio de Bernstein

Posteriormente dentro de dois ciclos aninhados, calculamos os seus índices e o respetivo ponto de controlo e criamos um ponto que irá ser calculado através da multiplicação entre cada uma das coordenadas dos pontos de controlo pelos polinómios u e v.
Após calculados todos os pontos, são escritos num ficheiro ".3d" do mesmo modo que as primitivas anteriormente desenvolvidas.
Na figura 2.3 está expresso a fórmula utilizada para calcular as superfícies de Bezier, onde (u,v)

```
Ponto pontoBezier(int p, float u, float v) {
    Ponto ponto = Ponto( a: 0.0, b: 0.0, c: 0.0);

    // Polinomio de Bernstein
    float bernsteinU[4] = { powf( x: 1-u, y: 3), 3 * u * powf( x: 1-u, y: 2), 3 * powf(u, y: 2) * (1-u), powf(u, y: 3) };
    float bernsteinV[4] = { powf( x: 1-v, y: 3), 3 * v * powf( x: 1-v, y: 2), 3 * powf(v, y: 2) * (1-v), powf(v, y: 3) };


    for (int j=0; j<4; j++)
        for (int i=0; i<4; i++) {

            //Indice dentro de um patch p
            int indexPatch = j*4+i;
            //Respetivo indice do ponto de controlo
            int indexCP = patches[p*16 + indexPatch];
            ponto = Ponto( a: ponto.getX() + controlPoints[indexCP * 3 + 0] * bernsteinU[j] * bernsteinV[i],
                           b: ponto.getY() + controlPoints[indexCP * 3 + 1] * bernsteinU[j] * bernsteinV[i],
                           c: ponto.getZ() + controlPoints[indexCP * 3 + 2] * bernsteinU[j] * bernsteinV[i]);


        }

    return ponto;
}
```

Figure 2.2: Cálculo de cada ponto

[0,1] ,*B(u)* e *B(v)* correspondem aos polinómios de Bernstein e *Kij* aos pontos de controlo.

$$p(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_i^n(u)\, B_j^m(v)\, K_{ij}$$

Figure 2.3: Fórmula para calcular cada ponto

2.2 Vbos

A utilização de VBOs (*Vertex Buffer Objects*) apresenta grandes benfícionos a nível de desempenho devido especialmente a dois fatores: deixa de ser necessária a chamada da função *glVertex* para cada vértice e explora, tirando partido do mesmo, o paralelismo disponiblizado pelo GPU uma vez que estamos a desenhar cada triângulo imeadiatamente ao receber cada conjunto de três vértices.

Como no trabalho prático apenas algumas das figuras construídas no *generator* é que iriam ser utilizadas no Sistema Solar (esfera,cintura de asteróides e anéis), criamos a possibilidade de apenas as utilizadas serem renderizadas com o uso de VBOs. Para tal, o *engine* reconhece se o ficheiro *.3d* tem de ser desenhado recorrendo aos VBOs ou a *GL TRIANGLES* através da primeira linha do mesmo. Se esta contiver o número de pontos da figura, então será gerada com recurso a VBOs. Caso se inicie com as coordenadas dos pontos desenha com triângulos. Isto é assegurado pela função **getFigure** criado no *engine* e também pelo *print* do número de pontos da figura que se encontra *generator*.
Como é necessário armazenar todos os vértices de uma figura num array, foi desenvolvida a função **drawVBOs**. Esta função recebe uma figura e o número de pontos da mesma, itera as coordenadas e preenche o array com as mesmas. Por fim a figura é desenhada utilizando esse mesmo array preenchido. Em baixo é mostrado o código da mesma.

```
void drawVBOs(vector<vector<Coordinate> > figures, int numPontos) {

    float *array = new float[numPontos];

    vector<vector<Coordinate> >::iterator fig;
    glEnableClientState(GL_VERTEX_ARRAY);
    unsigned int buffers;
    glGenBuffers(1, &buffers);
    glBindBuffer(GL_ARRAY_BUFFER, buffers);



    for (fig = figures.begin(); fig != figures.end(); fig++) {
        vector<Coordinate>::iterator it_coords;
        int it = 0;
        for (it_coords = fig->begin(); it_coords != fig->end(); it_coords++) {
            array[it++] = it_coords->x;
            array[it++] = it_coords->y;
            array[it++] = it_coords->z;
        }
        glBufferData(GL_ARRAY_BUFFER, it*sizeof(float), array, GL_STATIC_DRAW);
        glVertexPointer( size: 3,GL_FLOAT, stride: 0, pointer: 0);
        glDrawArrays(GL_TRIANGLE_STRIP, first: 0, count: numPontos/3);
    }
    delete[] array;
    glDeleteBuffers(1, &buffers);
}
```

Figure 2.4: Função drawVBOs

2.3 Catmull Rom Curves

Um dos objetivos principais nesta fase era definir as órbitas dos planetas através da definição das curvas de *CatMull-Rom*. Neste sentido, começando por uma pesquisa sobre a informção necessária obtivemos um série de equações necessárias para futura implementação no nosso programa.

As curvas de *Catmull-Rom* são uma implementação de cálculos de pontos através de matrizes. Para ter uma melhor compreenção do seu funcionamento, temos então a seguinde curva definida por 4 pontos:

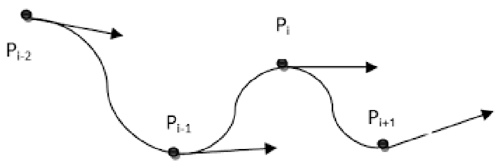

Figure 2.5: Curva derivada de 4 pontos

A partir destes mesmo pontos é possível então seguir a construção de duas matrizes essenciais para o cálculo destas mesmas curvas. Com eles contruímos então duas matrizes relacionas com o mesmo polinómio, no entanto uma é construída com este mesmo e a outra com a respetiva derivada.

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

- $P'(t) = [3t^2 \quad 2t \quad 1 \quad 0]\begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}\begin{bmatrix} P_o \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$

Figure 2.6: Matrizes de Calculo P

Também observado na figura anterior é a matriz que denominaremos M:

$$\begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Figure 2.7: Matriz M

Esta matriz necessária para o calculo das curvas e característica das mesmas, faz parte de uma série de equações correspondentes às curvas *Catmull-Rom*.

2.3.1 Implementação Catmull-Rom

Para a implementação destas curvas no nosso programa, começamos estão por calcular os pontos necessários para a formulação de uma curva. Estes pontos calculados através de equações envolvento pontos já lidos de ficheiros e valores caracteristicos, são armazenados com indices para posteriormente serem utilizador no calculo da curva:

```
int pointCount = trans.catPontos.size();
float t = gt * pointCount; // 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 + pointCount-1)%pointCount;
indices[1] = (indices[0]+1)%pointCount;
indices[2] = (indices[1]+1)%pointCount;
indices[3] = (indices[2]+1)%pointCount;
```

Figure 2.8: Pontos indexados

Após o calculo destes pontos seguimos então ao calculo dos polinomios que geram as curvas. Numa primeira fase, definimos a nossa matriz M necessária par os cálculos:

```
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} };
```

Figure 2.9: Matriz M

Esta é posteriormente utilizada para calcular a matriz A com os pontos que haviam sido guardados em indices através de multiplicação de matrizes, neste caso levada a cabo por uma função que denominamos por *multMatrixVector*:

```
// Compute A = M * P
float a[3][4];

float px[4] = { p0[0], p1[0], p2[0], p3[0] };
float py[4] = { p0[1], p1[1], p2[1], p3[1] };
float pz[4] = { p0[2], p1[2], p2[2], p3[2] };

multMatrixVector(*m, px,  res: a[0]);
multMatrixVector(*m, py,  res: a[1]);
multMatrixVector(*m, pz,  res: a[2]);
```

Figure 2.10: Construção da matriz A

Podemos então calcular as duas matrizes para a renderização da curva. Ambas são calculas através de um polinómio e a sua derivada definidos de seguida:

```
float tv[4] =  { t*t*t, t*t, t, 1 };
float tvd[4] = { 3*t*t, 2*t, 1, 0 };
```

Figure 2.11: Polinomios T e T’

Por fim temos então o calculo da matriz do molinomio T, denominado por *pos* e da matriz valculada por T’ denominado por *deriv*:

```
pos[0] = tv[0]*a[0][0] + tv[1]*a[0][1] + tv[2]*a[0][2] + tv[3]*a[0][3];
pos[1] = tv[0]*a[1][0] + tv[1]*a[1][1] + tv[2]*a[1][2] + tv[3]*a[1][3];
pos[2] = tv[0]*a[2][0] + tv[1]*a[2][1] + tv[2]*a[2][2] + tv[3]*a[2][3];
```

Matriz pos

```
// compute deriv = T' * A
deriv[0] = tvd[0]*a[0][0] + tvd[1]*a[0][1] + tvd[2]*a[0][2] + tvd[3]*a[0][3];
deriv[1] = tvd[0]*a[1][0] + tvd[1]*a[1][1] + tvd[2]*a[1][2] + tvd[3]*a[1][3];
deriv[2] = tvd[0]*a[2][0] + tvd[1]*a[2][1] + tvd[2]*a[2][2] + tvd[3]*a[2][3];
```

Matriz deriv

Estes valores vão então ser utilizados para a renderização das curvas.

2.3.2 Translação

No caso da translação, que é definida uma classe diferente representada em seguida,

```
class Translate {
public:
    bool empty = true;
    float time;
    vector<Coordinate> catPontos;
    float catmulls[50][3];
};
```

Figure 2.12: Classe Translate

armazena todos o pontos lidos do ficheiro XML no veto de coordenadas, e mais tarde, através das funções definidas para o cálculo das curvas *Catmull*, passará a guardar estes mesmos valores dentro da matriz *catmulls* para futura renderização.

```
if (!t.figure.translate.empty){
    renderCatmullRomCurve(t.figure.translate);
    renderCatmullTranslate(t.figure.translate);
```

Figure 2.13: Calculo das Curvas CatMull

2.3.3 Órbitas

Ainda no tema das tranlações, são necessários os pontos para o futuro cálculo das curvas. Estes pontos definidos no próprio ficheiro XML, foram calculados por nós através de uma pequena função em C que dado um x, y e z calcula em x, y e z uma curva circular nestas mesmas dimenções fornecidas.

```c
int main(){
    float um;
    float a = M_PI /12;
    float dois;


    float meio;
    scanf("%f",&um);
    scanf("%f",&meio);
    scanf("%f",&dois);
    int i;
    for(i=0;i<24;i++)
        printf("<point X=\"%f\"  Y=\"%f\" Z= \"%f\"  />\n",cos(a*i)*um,sin(a*i)*meio,-(sin(a*i)*dois));
    return 0;
}
```

Figure 2.14: Calculo dos pontos para o Translate

Como podemos observar nesta função, temos um ciclo *for* definido para 24 iterações, isto porque foi decidido por nós que seria um numero razoál de pontos a calcular para calcular uma órbita bem conseguida e suave.

Estes cálculos foram então utilizados no cálculo dos pontos para as órbitas dos planetas e respetivas luas, com algumas variações devido à órbita oval do nosso cometa.

2.4 Resultado final do Sistema Solar

Após todo o trabalho desenvolvido nesta fase e nas anteriores o resultado gráfico obtido pode ser visto nas figuras seguintes:

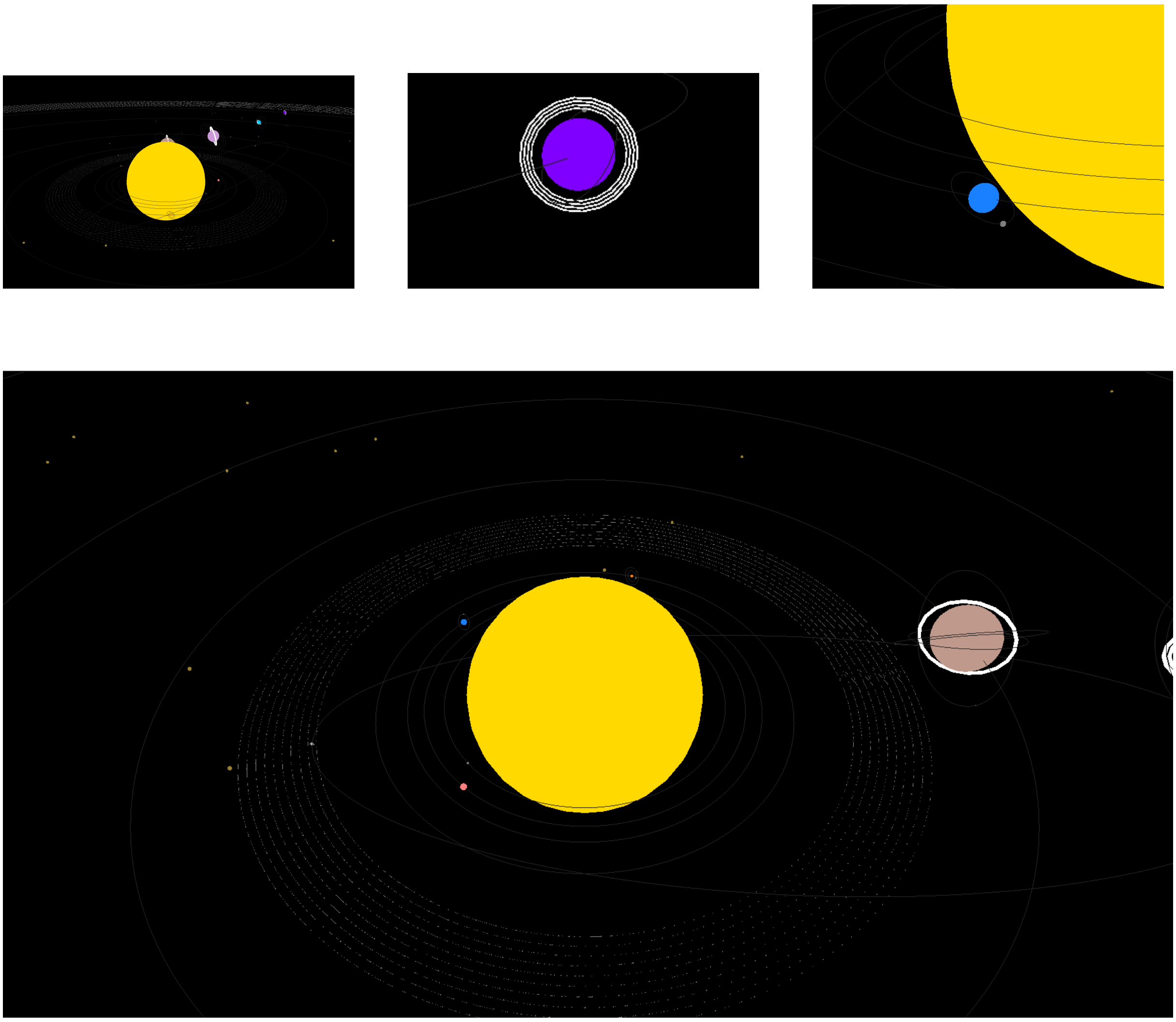

Figure 2.15: Resultado gráfico do Sistema Solar

Conclusão

Com a elaboração desta fase do projeto prático foi possível aplicar conhecimentos teóricos e práticos relativos tanto às curvas de *Bezier* como às de *Catmull-Rom* utilizadas para efetuar transformações sobre os objetos através de pontos de controlo e outros dados relativos. Foi também possível a implementação de um novo modo de gerar as figuras através de buffers.

A introdução das novas versões da rotação e translação a todos os outros componentes que já se encontravam presentes no projeto, fez com que o Sistema Solar esteja cada vez mais próximo da realidade.