Universidade do Minho
Licenciatura em Ciências da Computação

Unidade Curricular de Computação Gráfica

Ano Lectivo de 2021/2022

Phase 3 – Curves, Cubic Surfaces and VBOs

Filipe Azevedo A87969

João Nogueira A87973

Miguel Gonçalves A90416

Rui Baptista A87989

CG

Resumo

Neste documento, iremos abordar a terceira fase da avaliação prática da Unidade Curricular de Computação Gráfica.

O objetivo desta fase é implementar um sistema capaz de desenhar um certo conjunto de figuras já idealizadas, mas que desta vez, estas figuras e modelos sejam baseadas em fragmentos de Bezier. Para isto, tivemos de atualizar mais uma vez o Generator para puder receber mais um parâmetro, sendo este um ficheiro em que nele é definido os pontos de controlo e também o nível de tesselação. Tivemos ainda que, alterar o Engine, pois este foi estendido para puder receber os elementos de rotação e translação, onde nestes são providenciados um conjunto de pontos de forma a definir uma curva de Catmull-Rom, assim como o tempo para ir percorrendo estas curvas.

O objetivo desta fase é também, criar um sistema solar dinâmico, incluindo um cometa, contruído usando os ditos fragmentos de Bezier, que percorre uma trajetória definida por uma curva de Catmull-Rom.

Palavras-Chave: Computação gráfica, Generator, Engine, Bezier, Catmull-Rom, curvas.

Índice

Índice de Figuras

1. Introdução

Nesta fase do trabalho prático vamos acrescentar a possibilidade de animarmos a cena à nossa aplicação Engine, assim como gerarmos modelos a partir de patches de Bézier. Começamos por motivar o uso destas técnicas e descrevemos o processo de implementação das mesmas.

2. Estrutura do Projeto

A estrutura do projeto nesta terceira fase divide-se no generator e no engine. Como já foi visto, no generator geram-se os pontos, guardando-os num ficheiro .3d, que vão ser usados para desenhar as figuras no engine. Esses ficheiros estão num XML que o engine irá ler e produzir o modelo especificado.

No generator foi modificado para que fosse possível a produção de patches de Bezier.

Houve algumas modificações no engine, visto que foi necessário produzir figuras usando VBO's e introduzir movimento, criando órbitas segundo curvas de Catmull-Rom.

3. Alterações no Engine

Através do engine as figuras são desenhadas e apresentadas numa janela. Fizeram-se várias alterações no ficheiro XML e alteraram-se as seguintes classes.

3.1 Modelo

Para passarmos a desenhar figuras com VBO's aumentando o desempenho do programa, uma vez que o array de vértices ficará guardado na placa gráfica e utilizando esse array alocado para desenhar os triângulos.

Para implementarmos os tais VBO's podemos ou não utilizar índices. Como fizemos um VBO para cada modelo não faz diferença utilizar índices, porque a quantidade de triângulos não é grande, mas no caso de o número ser significativamente maior, a utilização de índices é melhor.

Na classe Modelo cada classe terá um vetor de float's que vai representar os pontos, ao invés da lista de float's que tínhamos anteriormente, vamos ter dois métodos prepareData e draw.

```cpp
using namespace std;

Modelo::Modelo() {
    r = 0.0f;
    g = 0.0f;
    b = 1.0f;
}

Modelo::Modelo(vector<float> pontos1, float r1, float g1, float b1) {
    pontos = pontos1;
    r = r1;
    g = g1;
    b = b1;
}

vector<float> Modelo::getPontos() {
    return pontos;
}

float Modelo::getR() {
    return r;
}

float Modelo::getG() {
    return g;
}

float Modelo::getB() {
    return b;
}

void Modelo::setPontos(vector<float> pontos1) {

    pontos = pontos1;
    pontos1.clear();
}

void Modelo::setR(float r1) {
    r = r1;
}

void Modelo::setG(float g1) {
    g = g1;
}

void Modelo::setB(float b1) {
    b = b1;
}


void Modelo::prepareData() {

    glGenBuffers(1, &vertices);
    glBindBuffer(GL_ARRAY_BUFFER, vertices);
    glBufferData(GL_ARRAY_BUFFER, sizeof(float) * pontos.size(), pontos.data(), GL_STATIC_DRAW);

}


void Modelo::draw() {

    glBindBuffer(GL_ARRAY_BUFFER, vertices);
    glVertexPointer(3, GL_FLOAT, 0, 0);
    glDrawArrays(GL_TRIANGLES, 0, pontos.size() / 3);

}
```

Figura 1 - Classe Modelo

No método prepareData criámos o VBO que se faz com glGenBuffers, e depois com a glBindBuffer definimos o VBO ativo e no final com glBufferData guardámos o vetor com os pontos no modelo na memória gráfica.

No método draw indicámos qual vértices é o VBO ativo, definimos os dados do VBO, neste caso cada vértice tem 3 float's, e no final desenhamos os triângulos tendo o primeiro argumento GL_TRIANGLES, 0 sendo a posição inicial e o tamanho de pontos a serem desenhados que será o números de pontos a dividir por 3, porque cada triângulo é constituído por 3 pontos

3.2 Rotação

Tivemos de implementar rotações "timed", que possuem um atributo de tempo em segundos, sendo que é o tempo que demorar a fazer uma rotação total de 360º.

Foi então introduzido um novo atributo "tempo" juntamente com os getters e setters, assim como a mudança no construtor.

```
Rotacao::Rotacao() {
    x = 0;
    y = 0;
    z = 0;
    tempo = 0;
    angulo = 0;
}

Rotacao::Rotacao(float x1, float y1, float z1, float t, float angulo1) {
    x = x1;
    y = y1;
    z = z1;
    tempo = t;
    angulo = angulo1;
}


float Rotacao::getAngulo() {
    return angulo;
}

float Rotacao::getX() {
    return x;
}

float Rotacao::getY() {
    return y;
}

float Rotacao::getZ() {
    return z;
}

float Rotacao::getTempo() {
    return tempo;
}

void Rotacao::setAngulo(float angulo1) {
    angulo = angulo1;
}

void Rotacao::setTempo(float t) {
    tempo = t;
}

void Rotacao::setX(float a) {
    x = a;
}

void Rotacao::setY(float b) {
    y = b;
}

void Rotacao::setZ(float c) {
    z = c;
```

Figura 2 - Classe Rotação

Tivemos de alterar a leitura no XML para guardar o novo atributo:

```
if (rotacaoElemento->Attribute("time") != nullptr) {
    tempo = stof(rotacaoElemento->Attribute("time"));
```

Figura 3 - Atributo time

E foi criado uma função para aplicar a rotação:

```
void aplicaRotacao(Rotacao rot) {

    float aux, anguloRot;
    int tempoPrograma;

    if (rot.getTempo() != 0) {
        tempoPrograma = glutGet(GLUT_ELAPSED_TIME);
        aux = tempoPrograma % (int)(rot.getTempo() * 1000);
        anguloRot = (aux * 360) / (rot.getTempo() * 1000);
        glRotatef(anguloRot, rot.getX(), rot.getY(), rot.getZ());
    }
}
```

Figura 4 - Aplica Rotação

Sendo que vai usar “GLUT_ELAPSED_TIME” para calcular o tempo de execução, em milissegundos, já passou no programa desde o “glutInit” como o tempo cresce linearmente vamos fazer o módulo pelo tempo que demora a fazer uma rotação completa. Em seguida, guardámo-la numa variável para a conta seguinte, onde calculamos o ângulo de rotacão, usando a “regra dos três simples”.

Para aplicar a rotação é chamada a cada “renderScene” na função drawGrupo.

```
case 'R':
    if (r.getAngulo() != 0) {
        glRotatef(r.getAngulo(), r.getX(), r.getY(), r.getZ());
    }
    else {
        aplicaRotacao(r);
    }
```

Figura 5 - Aplica Rotação II

3.3 Translação

Na classe Translação, não terá apenas coordenadas x, y e z. Terá também um tempo e dois vetores de Pontos, um alocará os pontos lidos no xml e o outro alocará os pontos que constituem a Catmull Rom Curve.

```
Translacao::Translacao() {
    x = 0;
    y = 0;
    z = 0;
}

Translacao::Translacao(float x1, float y1, float z1) {
    x = x1;
    y = y1;
    z = z1;
}

Translacao::Translacao(float t, vector<Ponto> v) {
    tempo = t;
    pontos = v;
}
```

Figura 6 - Classe Translação

Uma translação com tempo vai occrer no método draw da class Translação:

```
void Translacao::draw() {
    float pos[3];
    float deriv[3];
    float te, gt;
    if (tempo != 0 && pontos.size() > 3) {
        te = glutGet(GLUT_ELAPSED_TIME) % (int)(tempo * 1000);
        gt = te / (tempo * 1000);
        buildCurve(pos, deriv);
        drawCatmullRomCurve();
        getGlobalCatmullRomPoint(gt, pos, deriv, pontos);
        pontosCurva.clear();
        glTranslatef(pos[0], pos[1], pos[2]);
        alinhamentoCurva(deriv);
    }
}
```

Figura 7 - Classe Translação II

Definimos 4 variáveis:

pos[3] = vai alocar as coordenadas x,y e z para a próxima translação dentro da curva.

deriv[3] = array para ajudar na criação da matriz de rotação de rotação de forma a alinhar com a curva.

te = tempo desde última chamada do glutInit em milissegundos, módulo pelo tempo * 1000, pois estamos a trabalhar em milissegundos e não queremos que o te aumente infinitamente.

gt = tempo em segundos

Depois no “if statement” temos a restrição que o tempo deve ser maior que 0, e devemos ter mais que 3 pontos no mínimo pois para uma **Catmull Rom Cubic Curve** necessitamos de 4 pontos.

Caso o “if statement” seja superado teremos então:

1. O cálculo do **gt**, de forma a futuramente obter-mos a posição na curva associada a esse tempo.

2. **buildCurve** -> função que irá preencher o vetor pontosCurva da translação, dependendo do t usado podemos escolher quantos pontos constituirão a curva. Quanto menor o incremento do t, mais pontos constituirão a curva, logo mais segmentos de reta e por isso mais curvatura.

```
void Translacao::buildCurve(float* pos, float* deriv) {

    for (float t = 0; t < 1; t += 0.001) {

        getGlobalCatmullRomPoint(t, pos, deriv, pontos);
        pontosCurva.push_back(*new Ponto(pos[0], pos[1], pos[2]));
    }

}
```

Figura 8 - Classe Translação III

10 pontos (t+=0.1) 1000 pontos (t+=0.001)

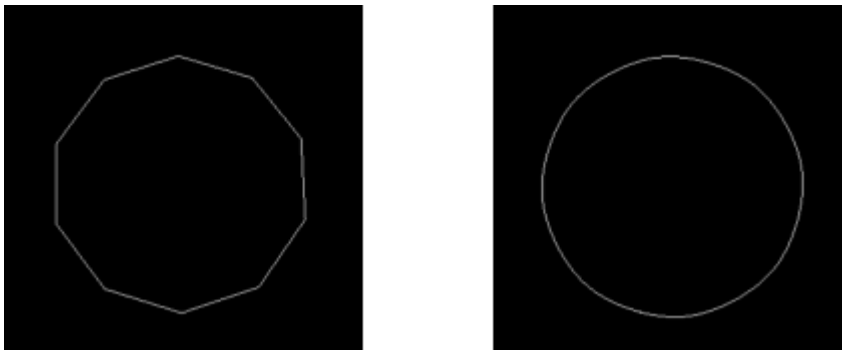

Figura 9 - Pontos

Para cada t ∈ [0,1[, foi calculado o ponto na curva relativa a esse t pela função **getGlobalCatmullRomPoint**, que usando o t descobre 4 pontos da curva de forma a descobrir-mos o **P(t).**

```
void Translacao::getGlobalCatmullRomPoint(float gt, float* pos, float* deriv, vector<Ponto> pontos) {

    int tamLoop = pontos.size(); // Points that make up the loop for catmull-rom interpolation
    float t = gt * tamLoop; // 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 + tamLoop - 1) % tamLoop;
    indices[1] = (indices[0] + 1) % tamLoop;
    indices[2] = (indices[1] + 1) % tamLoop;
    indices[3] = (indices[2] + 1) % tamLoop;

    getCatmullRomPoint(t, pontos[indices[0]], pontos[indices[1]], pontos[indices[2]], pontos[indices[3]], pos, deriv);
}
```

Figura 10 - Classe Translação IV

O processo de cálculo do **P(t)** é feito na função **getCatmullRomPoint**, tal como a derivada P’(t).

Temos que:

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

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

Onde:

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

```
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} };
```

$$T = [t^3 \quad t^2 \quad t \quad 1]$$

```
float T[4] = { t * t * t, t * t, t, 1 };
```

$$T' = [3t^2 \quad 2t \quad 1 \quad 0]$$

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

$$P = \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

```
float P[4][3] = { {p0.getX(),p0.getY(), p0.getZ()},
                  {p1.getX(), p1.getY(), p1.getZ()},
                  {p2.getX(), p2.getY(), p2.getZ()},
                  {p3.getX(), p3.getY(), p3.getZ()} };
```

Calculando assim:

$$A = M \times P$$

```
float A[4][3] = { 0 };

for (int i = 0; i < 4; i++)
    for (int j = 0; j < 3; j++)
        for (int k = 0; k < 4; k++) A[i][j] += M[i][k] * P[k][j];
```

$$P(t) = T \times A$$

```
for (int j = 0; j < 3; j++)
    for (int k = 0; k < 4; k++) pos[j] += T[k] * A[k][j];
```

$$P'(t) = T' \times A$$

```
for (int j = 0; j < 3; j++)
    for (int k = 0; k < 4; k++) deriv[j] += Tderiv[k] * A[k][j];
```

Figura 11 - Calculo de P(t)

Depois de termos a curva desenhada vamos descobrir a sua posição, que ficará guardada no array **pos,** e descobrir a sua derivada(tangente), que ficará guardada no array **deriv**, para o **gt**(global time), usando a função descrita em cima **getGlobalCatmullRomPoint.**

```
void Translacao::drawCatmullRomCurve() {

    int tam = pontosCurva.size();

    glBegin(GL_LINE_LOOP);

    for (int i = 0; i < tam; i++) {
        Ponto p = pontosCurva[i];
        glVertex3f(p.getX(), p.getY(), p.getZ());
    }

    glEnd();
```

Figura 12 - Classe Translação V

Depois de descobrimos a posição na curva basta fazer um **glTranslatef** com os valores no array **pos.**

De forma a alinharmo-nos pela curva precisamos de calcular a **matriz de rotação,** e para isso usamos a **derivada** calculada anteriormente.

```
void Translacao::buildRotMatrix(float* x, float* y, float* z, float* m) {

    m[0] = x[0]; m[1] = x[1]; m[2] = x[2]; m[3] = 0;
    m[4] = y[0]; m[5] = y[1]; m[6] = y[2]; m[7] = 0;
    m[8] = z[0]; m[9] = z[1]; m[10] = z[2]; m[11] = 0;
    m[12] = 0; m[13] = 0; m[14] = 0; m[15] = 1;
}


void Translacao::cross(float* a, float* b, float* res) {

    res[0] = a[1] * b[2] - a[2] * b[1];
    res[1] = a[2] * b[0] - a[0] * b[2];
    res[2] = a[0] * b[1] - a[1] * b[0];
}


void Translacao::normalize(float* a) {

    float l = sqrt(a[0] * a[0] + a[1] * a[1] + a[2] * a[2]);
    a[0] = a[0] / l;
    a[1] = a[1] / l;
    a[2] = a[2] / l;
}
```

Figura 13 - Classe Translação VI

É preciso funções auxiliares, uma para normalizar vetores de forma a serem unitários, e outra para criar a matriz de rotação a partir de 3 vetores. Por fim uma que calcula o produto vetorial entre dois vetores de forma a obtermos um perpendicular aos dois.

Para alinharmos o objeto pela curva temos de calcular a matriz de rotação M e utiliza-la na glMultMatrixF que executará a rotação:

$$\overrightarrow{X_i} = P'(t)$$
$$\overrightarrow{Z_i} = X_i \times \vec{Y}_{i-1}$$
$$\overrightarrow{Y_i} = \overrightarrow{Z_i} \times \overrightarrow{X_i}$$

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

```
void Translacao::alinhamentoCurva(float* deriv) {

    float Z[3];
    float Y[3] = { 0,1,0 };
    float X[3] = { deriv[0],deriv[1],deriv[2] };
    float m[16];

    cross(X, Y, Z);
    cross(Z, X, Y);

    normalize(X);
    normalize(Y);
    normalize(Z);

    buildRotMatrix(X, Y, Z, m);

    glMultMatrixf((float*)m);
}
```

Figura 14 - Classe Translação VII

4. Alterações no generator

Aqui, todas as funcionalidades anteriormente desenvolvidas, obviamente, se mantêm e houve a criação então da funcionalidade que permite construir modelos com base em curvas de Bezier.

De forma a aumentar a qualidade de desenho do projeto, vai se usar uma nova técnica que recorre a patches de bezier. Um patch de Bezier utiliza 16 pontos de controlo relativamente a uma superfície. Isto torna possível desenhar figuras complexas com um aspeto realista, pois são postas de parte as figuras desenhadas com faces retas e passam a ser, de facto, desenhadas com superfícies arredondadas.

Para explicar de uma forma ainda mais direta, foi fornecido um ficheiro .patch, o qual servirá como modelo para estrutura de ficheiros do mesmo tipo, a qual passo a mostrar:

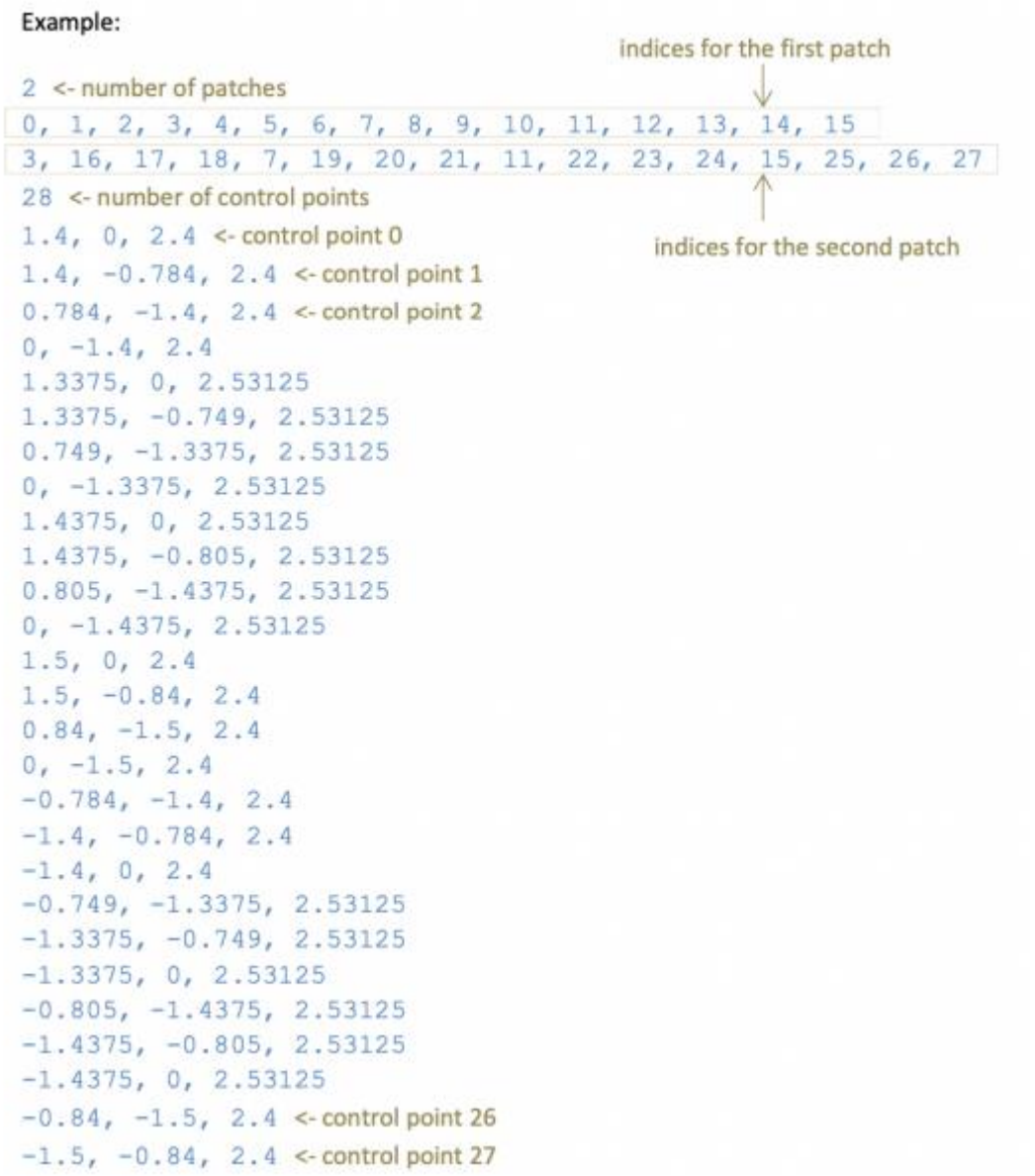

Figura 15 - Patch file format

Como temos de processar este patch e transformá-lo num dos ficheiros .3d usados para guardar pontos de modelos foi criada a seguinte função que permite a leitura do ficheiro e chama ainda outras funções que vao ser referidas:

```
void drawBezierPatches(int nivel, string origem) {
```

Figura 16 - Função DrawBezierPatches

Para criar uma curva de Bezier, são necessários 4 pontos, que se denominam de pontos de controlo e são constituídos pelas coordenadas x, y e z. Mas, aqui só temos um conjunto de pontos, para termos uma curva temos de a calcular, usando estes pontos.
É calculada segundo esta fórmula:

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

Figura 17 - Fórmula Bezier

Aqui são necessárias 2 variáveis para efetuar a especificação do nível de tesselação (u,v), que calculam o grau de curvatura, ou seja, representam o mesmo que a variável t na fórmula:

$$B(t) = (1-t)^3 B_0 + 3t(1-t)^2 B_1 + 3t^2(1-t)B_2 + t^3 B_3, t \in [0,1]$$

Figura 18 - Fórmula Bezier II

Voltando então à função “drawBezierPatches”, primeiramente recorremos à leitura do ficheiro .patches:

```
//abrir o ficheiro
ifstream file1("C:\\Users\\Pestana\\Desktop\\Trabalho_Fase_3_Grupo10\\Patches\\" + origem + ".patch");
if (!file1.is_open()) { cout << "Erro ao ler o ficheiro paches" << endl; return; }
//pega na primeira linha que é o número de patches ou seja numero de linhas a ler
getline(file1, linha);
int nPatches = atoi(linha.c_str());

int t;
for (int i = 0; i < nPatches; i++) {
    getline(file1, linha);     //pegar na linha atual
    stringstream strstream(linha);
    patches.push_back(std::vector<int>());
    while (strstream >> t) {    // retirar as ,
        patches[i].push_back(t);
        if (strstream.peek() == ',') {
            strstream.ignore();
        }
    }
}
```

Figura 19 - Função DrawBezierPatches II

Em seguida, a função aborda a primeira linha que contém o número de vértices, ou seja, o número de linhas a ler (1 vértice por linha).

```
getline(file1, linha);
int nLinhas = atoi(linha.c_str());
float x, y, z;
for (int i = 0; i < nLinhas; i++) {
    getline(file1, linha);      //pegar na linha atual
    stringstream strstream(linha);
    strstream >> x;

    if (strstream.peek() == ',') {
        strstream.ignore();
    }

    strstream >> y;
    if (strstream.peek() == ',') {
        strstream.ignore();
    }

    strstream >> z;
    if (strstream.peek() == ',') {
        strstream.ignore();
```

Figura 20 - Função DrawBezierPatches III

Aqui já vemos que estamos a preencher o vetor dos pontos de Controlo para que depois sejam utilizados.
Mas, só agora chegamos à parte crucial da função:

```
file1.close();

for (int i = 0; i < nPatches; i++) {
    float u = 0.0;
    float v = 0.0;
    for (int j = 0; j < nivel; j++) {
        for (int m = 0; m < nivel; m++) {
            curvaBezier(&resultado, patches[i], pontosControlo, u, v, intervalo);
            v += intervalo;
        }
        u += intervalo;
        v = 0.0;
    }
}
```

Figura 21 - Função DrawBezierPatches IV

É aqui que percorremos as chamadas à próxima função definida, que se denomina de “curvaBezier”.

Como podemos ver, aqui quanto maior for o nível de tesselação, maior vai ser o número de pontos calculados o que, consequentemente, resulta numa superfície mais perfeita e realista.

Agora, passo a mostrar a função “curvaBezier”, a qual apartir do vetor patch, do vetor pontos de controlo e ainda das variáveis u e v, num determinado intervalo, calculam os vértices:

```
void curvaBezier(std::vector<Ponto>* vertices, std::vector<int> patch, std::vector<Ponto*> pontosControlo, float u, float v, float intervalo) {

    Ponto p1 = calcula(patch, pontosControlo, u, v);
    Ponto p2 = calcula(patch, pontosControlo, u, v + intervalo);
    Ponto p3 = calcula(patch, pontosControlo, u + intervalo, v);
    Ponto p4 = calcula(patch, pontosControlo, u + intervalo, v + intervalo);

    vertices->push_back(p1);
    vertices->push_back(p4);
    vertices->push_back(p2);

    vertices->push_back(p4);
    vertices->push_back(p1);
    vertices->push_back(p3);
}
```

Figura 22 - Função curvaBezier

De forma a explicar e a entender de forma mais simples o sucedido “na prática”, vejamos a seguinte imagem

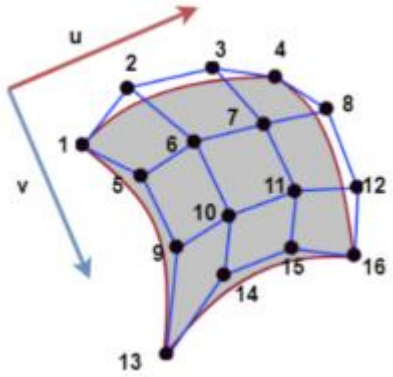

Figura 23 - Curva Bezier

Aqui, podemos entender que, como avançamos a variável v até que esta chegue ao valor 1 e
depois, colocamos de novo v com valor 0 e incrementámos a variável u com o valor do intervalo (float intervalo = (float)1.0 / nivel;). Temos que, no caso da figura apresentada, calcularíamos os triângulos compostos pelos vértices (1,6,5) e (6,1,2) primeiramente, depois seria (5,10,9) e (10,5,6) e assim sucessivamente.

Finalmente, relativamente a estas funções para calcular curvas de Bezier, a função “calcula”, esta é um pouco extensa e portanto, não vamos colocar print da mesma, para não se tornar exausta a leitura do relatório.

Mas, de realçar que nesta, era colocado em prática a fórmula referida em cima. Onde, primeiramente se calcula U*M, em seguida calculamos resultado obtido pela matriz dos pontos de controlo, daqui obtêm-se: (1) U*M * M[pontos controlo].

Em seguida, calcula-se (2) M^T (transposta) * V , onde no final da função acabamos por obter o resultado final esperado: (1) * (2) , retornando o Ponto (x,y,z).

5. Extras

Nas fases anteriores já conseguiamos mudar a cor dos objetos, mas ficavam todos com a mesma. Nesta fase o grupo decidiu mudar e aplicar uma cor diferente a cada objeto, de forma a que o sistema solar fique mais “bonito”.
Para tar, adicionamos 3 atributos ao XML referentes ao R G B.

```
<group>
    <!-- Mercurio -->
    <transform>
        <rotate time="4.4" x = "0.0" y ="1.0" z ="0.0" />
        <translate x="2.316" y="0" z="0" />
        <scale x="0.05" y="0.05" z="0.05"/>
    </transform>
    <models>
        <model file="sphere.3d" R="0.552" G="0.549" B="0.486" />
    </models>
</group>
```

Figura 24 - XML RGB

E
actualizamos a nossa classe Modelo para receber estes novos valores.

```
float Modelo::getR() {
    return r;
}

float Modelo::getG() {
    return g;
}

float Modelo::getB() {
    return b;
}

void Modelo::setPontos(vector<float> pontos1) {

    pontos = pontos1;
    pontos1.clear();
}

void Modelo::setR(float r1) {
    r = r1;
}

void Modelo::setG(float g1) {
    g = g1;
}

void Modelo::setB(float b1) {
    b = b1;
}
```

Figura 25 - Classe Modelo RGB

6. Resultados

Como na fase anterior, o sistema solar foi feito o mais parecido possível com a realidade.

Nesta fase na animação do mesmo tentamos assim aplicar tempos de translação em torno do sol e de rotação em torno de si mesmo com proporções semelhantes á realidade.

Também foi adicionado ao sistema solar um cometa na forma de um teapot, gerado a partir de curvas de Bezier, adicionando também uma rota e animação ao mesmo.

Sistema solar:

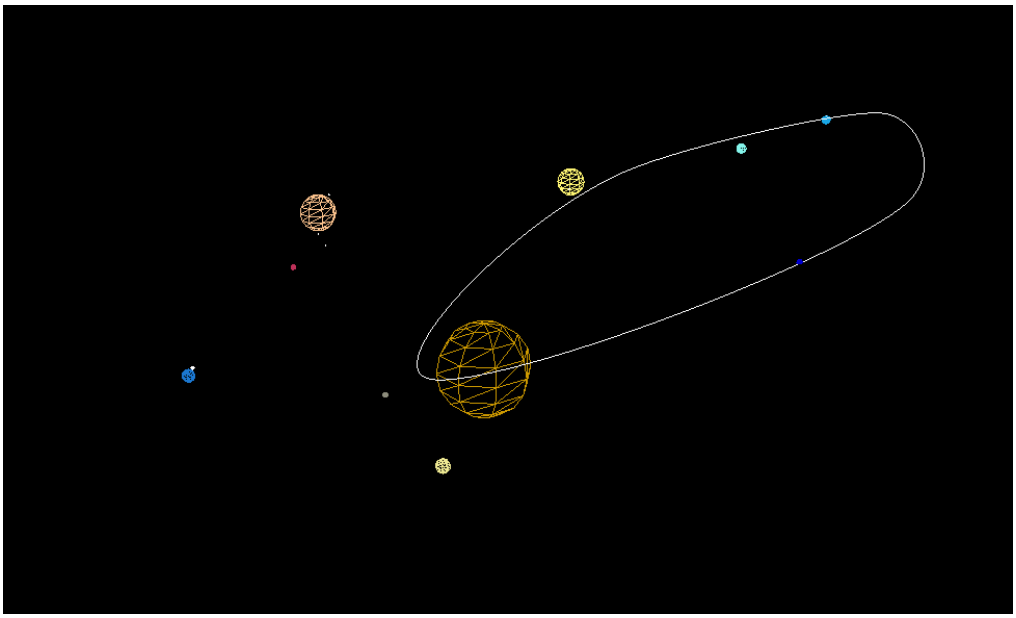

Figura 26 - Sistema Solar

Cometa aproximado:

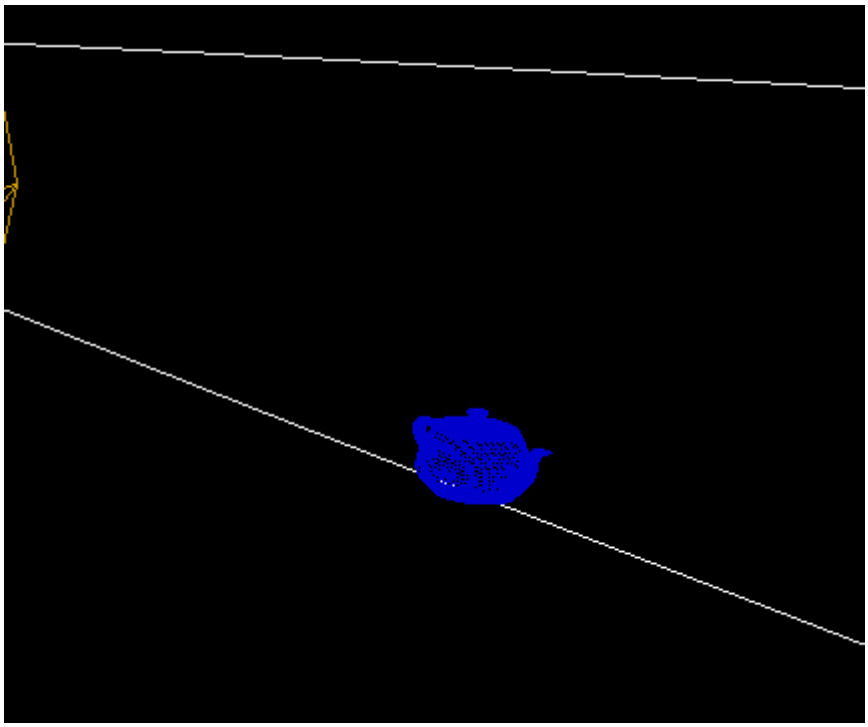

Figura 27 - Cometa

7. Conclusão

Concluindo, o grupo concorda em dizer que até agora foi de facto a fase do trabalho mais complexa. Isto sendo que, não é preciso ser necessário criar tantas classes, como foi o caso da segunda fase, para que de facto tenhamos mais dificuldades. Esta foi uma fase em que podemos dizer que a dificuldade dos temas abordados é mais elevada e que contém conceitos já mais complexos como as curvas de Catmull-Rom e Patches de Bezier.

Em ambos os casos houve a situação de tentativa erro que por mais que parecesse um ciclo infinito, acabou por se tornar resolvido com uma solução que o grupo achou adequada.

Posto isto, achamos o trabalho foi bem desenvolvido e que se obteve o resultado pretendido.