Computação Gráfica

Trabalho Prático

Fase 3 - Catmull-Rom Curves and Bézier Patches

Luís Almeida	João Pedro Antunes	Fernando Lobo	Diogo Monteiro
A84180	A86813	A87988	A71452

2 de maio de 2021

Resumo

O presente documento descreve a resolução do enunciado adotada pelo grupo. Nesta fase do trabalho prático foi nos proposta a introdução da possibilidade da aplicação *Generator* gerar um novo modelo baseado em patches de Bézier. Foi nos também pedido que adicionássemos uma funcionalidade à aplicação *Engine* para permitir a definição de curvas Catmull-Rom para animação da cena.

Conteúdo

1 Introdução **2**

2 Análise e Especificação **3**

2.1 Curvas de Catmull-Rom 3

2.2 Curvas de Bézier 3

2.2.1 *Geração da superfície* 3
2.2.2 *Bézier patches* 3

3 Concepção/desenho da Resolução **4**

3.1 Curvas de Catmull-Rom 4

3.2 *Bézier patches* 6

4 Extras **9**

4.1 *Height Maps* 9

4.2 Controles de *Speed* 10

5 Conclusão **11**

Capítulo 1

Introdução

Nesta fase do trabalho prático vamos acrescentar a possibilidade de animarmos a cena à nossa aplicação *Engine*, assim como gerarmos novos modelos a partir de patches de Bézier. Começamos por motivar o uso destas técnicas e descrevemos o processo de implementação das mesmas. Apresentamos também os extras que decidimos incluir nesta fase do trabalho prático.

Capítulo 2

Análise e Especificação

2.1 Curvas de Catmull-Rom

As curvas de Catmull-Rom são bastante úteis para modelarmos a animação de modelos na nossa cena. Permitem-nos calcular a posição de um modelo que está seguir uma determinada curva num dado instante, assim como a sua rotação. Para implementarmos esta técnica no nosso trabalho, podemos olhar para a animação sobre uma curva de Catmull-Rom como uma transformação geométrica. Assim, podemos estender o nosso "leque"de transformações geométricas com as translações (e rotações) especificadas pelas curvas de Catmull-Rom. Basta então encapsularmos todos os cálculos necessários dentro da nossa classe que implementa esta "transformação geométrica".

2.2 Curvas de Bézier

De maneira a conseguir-se obter a superfície curva pretendida em determinados objetos é necessário recorrer a uma outra abordagem que não a já definida. Para isso recorre-se à implementação de curvas de Bézier.

2.2.1 *Geração da superfície*

Para se gerar uma *patch* são necessários dezasseis pontos de controlo, os quais estão separados em grupos de quatro pontos, que é a quantidade necessária para definir uma curva. Sabendo isto e tendo uma quantia, denominada tesselação, é possível calcular os pontos pertencentes a essa curva. Definidos os pontos para as quatro curvas é então possível traçar uma outra, perpendicular, que intersete as curvas já definidas, gerando assim uma superfície arredondada.

2.2.2 *Bézier patches*

Para entender melhor a implementação das curvas descritas é melhor analizar a organização do ficheiro fornecido, como exemplo, com patches de Bézier.

O ficheiro está separado em duas partes, sendo que a primeira contém, para além do número de *patches*, os índices dos pontos de controlo para cada *patch* onde, cada uma destas, contendo dezasseis índices, está definida por linha e na segunda parte encontram-se as coordenadas dos pontos de controlo, ordenados por índices.

Capítulo 3

Concepção/desenho da Resolução

3.1 Curvas de Catmull-Rom

Conforme referido no capítulo anterior, vamos então estender a nossa classe de transformações geométricas para lidar também com curvas Catmull-Rom, encapsulando os cálculos na mesma. Ora, o nome da nossa sub-classe da classe Transform será CatmullTranslate, e iremos criar uma instância dela após termos feito parsing de um ficheiro XML que nos indique que tal será necessário. Assim, definimos o construtor da classe da seguinte forma:

```
CatmullTranslate(float time,std::vector<float> controlPts):Transform(controlPts){
                        timeToCycle = time*1000;
                        if (timeToCycle <= 0) timeToCycle = 1000;

                }
```

A variável timeToCycle vai nos indicar o tempo total que temos disponível para o modelo dar uma volta inteira à curva.

Podemos já pensar que o nosso método applyTransform irá ser da seguinte forma

```
    void applyTransform(int timeDelta, int toDraw){
                        if (toDraw)     {
                                this->draw();
                        }

                        currPoint(timeDelta);
                        getRotationMat();
                        glTranslatef(point[0],point[1],point[2]);
                        glMultMatrixf(&rot[0][0]);
                }
```

A variável timeDelta será comunicada pela renderScene e corresponde ao intervalo de tempo que passou desde o desenho da última frame. A variável toDraw serve para indicar se o utilizador pediu que a curva seja desenhada.

O método currPoint faz então o cálculo da posição do modelo na curva, assim como a sua rotação, seguindo o procedimento de Catmull-Rom.

```
  void currPoint(int timeDelta){
                        float offset = (float) timeDelta/timeToCycle;
                        lastPoint += offset;

                        float t = lastPoint * (float)(axis.size()/3);
                        std::vector<float> vec = getPointCurve(t,1);
                        point.clear();


                        for (int i=0; i<vec.size(); i++)
                        point.push_back(vec[i]);
                }
```

Começamos por calcular o quanto o avançamos na curva, depois usamos a função getPointCurve que faz todos os cálculos necessários para realizarmos a translação e a rotação do modelo que segue a curva.

```
std::vector<float> getPointCurve(float t, int rot=0){
                    int index = floor(t);
                    t = t - index;

                    int indices[4];
                    indices[0] = (index + (axis.size()/3)-1)%(axis.size()/3);
                    indices[1] = (indices[0]+1)%(axis.size()/3);
                    indices[2] = (indices[1]+1)%(axis.size()/3);
                    indices[3] = (indices[2]+1)%(axis.size()/3);


                    std::vector<float> p0 = { axis[(indices[0]*3)],axis[(indices[0]*3)
                        +1],axis[(indices[0]*3)+2],1.0f };
                    std::vector<float> p1 = { axis[(indices[1]*3)],axis[(indices[1]*3)
                        +1],axis[(indices[1]*3)+2],1.0f };
                    std::vector<float> p2 = { axis[(indices[2]*3)],axis[(indices[2]*3)
                        +1],axis[(indices[2]*3)+2],1.0f };
                    std::vector<float> p3 = { axis[(indices[3]*3)],axis[(indices[3]*3)
                        +1],axis[(indices[3]*3)+2],1.0f };


                    std::vector<float> res;

                    std::vector<std::vector<float>>catmullM = {
                            {-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}
                            };

                    std::vector<std::vector<float>> a;
                    std::vector<float> T = { (float)pow(t,3),(float)pow(t,2),t,1 };
                    std::vector<float> Tlinha = { 3*(float)(pow(t,2)), 2*t, 1, 0 };
                    for (int i = 0;  i < 4; i++){
                            a.push_back({0});
                            std::vector<float> p = { p0[i],p1[i],p2[i],p3[i] };

                            matDotVec(catmullM, p, &a[i]);
                    }

                    res.clear();
                    matDotVec(a,T,&res); //

                    if (rot){
                            matDotVec(a,Tlinha,&deriv);
                    }
                    return res;
            }
```

O método matDotVec é um método auxiliar que simplesmente multiplica uma matriz por um vetor.

```
void matDotVec(std::vector<std::vector<float>> mat, std::vector<float> vec, std::
    vector<float>*res){
                    (*res).clear();
                    for (int i = 0; i < mat.size(); i++){
                            (*res).push_back(0);
                            for (int j = 0; j < mat[0].size(); j++){
                                    (*res)[i] = (*res)[i] + (vec[j] * (mat[i][j]));
                            }
                    }
            }
```

Precisamos agora de definir o método getRotationMat() que irá calcular a matriz de rotação do modelo na curva.

```
void getRotationMat(){
                    normalize(deriv);
                    normalize(lastY);
                    lastZ = crossProd(deriv,lastY);
                    lastY = crossProd(lastZ,deriv);
                    rot[0][0] = deriv[0]; rot[0][1] = lastY[0]; rot[0][2] = lastZ[0];
                        rot[0][3] =        0;
                    rot[1][0] = deriv[1]; rot[1][1] = lastY[1]; rot[1][2] = lastZ[1];
                        rot[1][3] =        0;
                    rot[2][0] = deriv[2]; rot[2][1] = lastY[2]; rot[2][2] = lastZ[2];
                        rot[2][3] =        0;
                    rot[3][0] =        0; rot[3][1] =        0; rot[3][2] =        0;
                        rot[3][3] =        1;
            }
```

Para obtermos a matriz de rotação seguimos o processo descrito pelo professor nas aulas práticas. Após este método, simplesmente aplicamos a translação que representa o deslocamento do modelo na curva e multiplicamos a matriz de rotação pela matriz do sistema de coordenadas.

3.2 *Bézier patches*

Uma abordagem para as *patches* de *Bézier* e o cálculo dos vértices da superfície, consistiu na criação de uma nova classe no *generator*. Nesta classe faz-se a leitura dos respetivos ficheiros e a geração dos pontos, que serão depois tratados recorrendo ao método (*trisPolyLine()*) já definido anteriormente.

Para obtenção dos dados durante a leitura do ficheiro das patches foi definido um método que, de acordo com o número de *patches* e o número de pontos de controlo, armazena os índices, por *patch*, num vetor e os pontos relativos a cada curva noutro.

Com esta informação e com a tesselação que é calculada através dos argumentos fornecidos é possível calcular os pontos presentes em cada curva, armazenando-os para a geração do ficheiro *.3d*.

```
void shape(){

        float u = 0;
        float v = 0;
        int vtxPts = 1;
        std::vector<int> row;

        for (int kk = 0; kk < patcheNum; kk++){
                u = 0;
                std::vector<std::vector<float>> interCurve;
                for (int i = 0; i <= tesselationLv; i++){
                        interCurve.clear();
                        row.clear();

                        interCurve.push_back(getCurrPoint(curves[(kk*4)],u));
                        interCurve.push_back(getCurrPoint(curves[(kk*4)+1],u));
                        interCurve.push_back(getCurrPoint(curves[(kk*4)+2],u));
                        interCurve.push_back(getCurrPoint(curves[(kk*4)+3],u));

                        v = 0;
                        for (int j = 0; j <= tesselationLv; j++){

                                std::vector<float> newPoint = getCurrPoint(interCurve,v);
                                newPoint.erase(newPoint.begin() + 3);
                                vertex.push_back(newPoint);

                                row.push_back(vtxPts++);
                                v += tessFact;
                        }
```

```
                        row.push_back((-1));

                        (this->planar).insert(this->planar.begin(), row);
                        u += tessFact;
                }

                row.clear();
                for (int i = 0; i < tesselationLv+2; i++){
                        row.push_back(-1);

                }
                (this->planar).insert(this->planar.begin(), row);

        }
}
```

Para calcular os pontos recorre-se ao seguinte método, que efetua o cálculo das coordenadas recorrendo à matriz definida e de acordo com a tesselação fornecida.

```
std::vector<float> getCurrPoint(std::vector<std::vector<float>> pts,float t){
        std::vector<float> point;
        std::vector<std::vector<float>>bezierM = {
                        {-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}
                        };

        std::vector<std::vector<float>> a;
        std::vector<float> T = { (float)pow(t,3),(float)pow(t,2),t,1 };

        for (int i = 0;  i < 4; i++){
                a.push_back({0});
                std::vector<float> p = { pts[0][i],pts[1][i],pts[2][i],pts[3][i] };
                matDotVec(bezierM, p, &a[i]);
        }

        point.clear();
        matDotVec(a,T,&point);

        return point;
}
```

Apresentamos de seguida um exemplo em que um bule de chá gerado por um patch de Bézier percorre uma curva de Catmull-Rom, bem como a nossa cena do sistema solar:

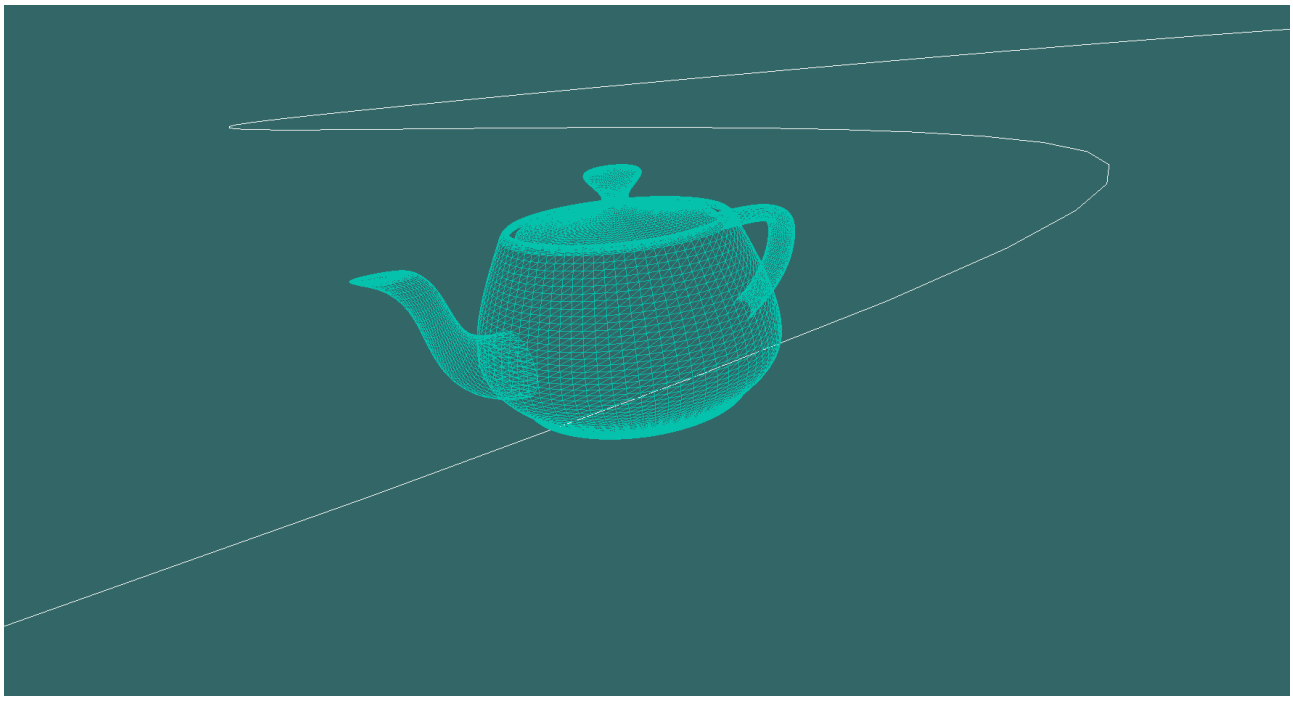

Figura 3.1: Exemplo

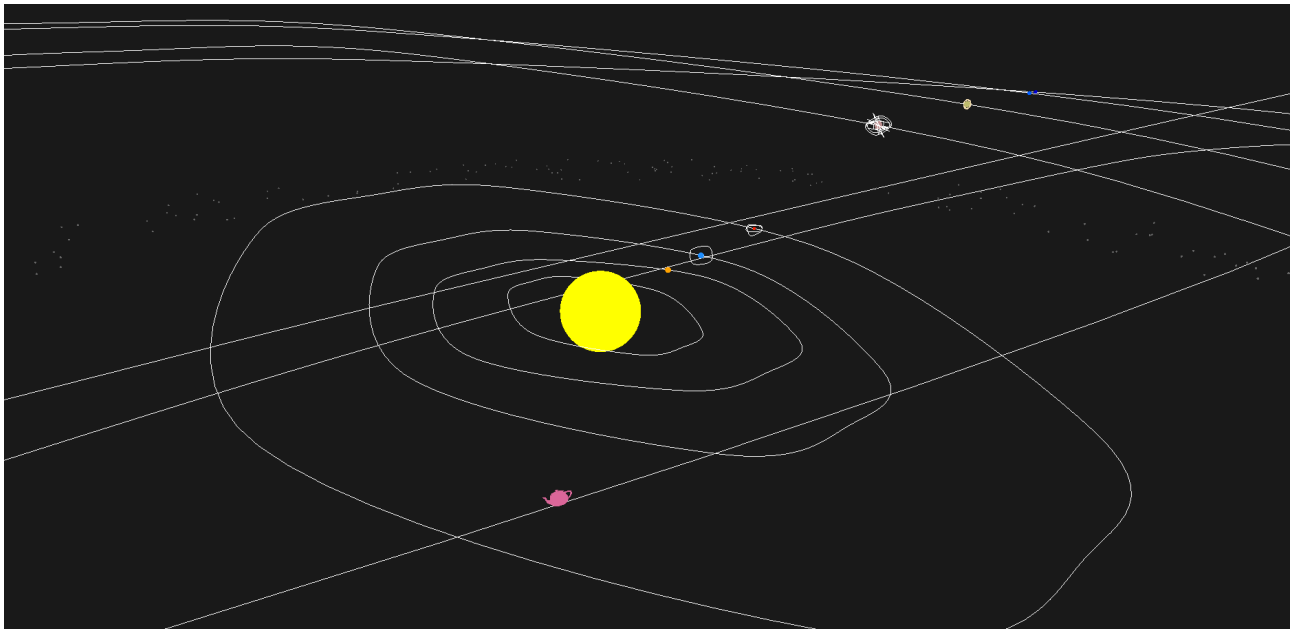

Figura 3.2: Exemplo

Capítulo 4

Extras

4.1 *Height Maps*

Nesta secção abordaremos o extra realizado pelo grupo relativo a aplicação de mapas de relevo, ao contrário dos mapas apresentados em aula, foi realizada uma adaptação de maneira a que os mapas de relevo fossem aplicados a qualquer plano sem restrição quanto ao número de vértices.

Para a realização, foram adicionadas novos atributos a *tag* de *XML* relativa ao modelo. Nestes atributos, nomeadamente (*hmap* e I)definem-se respetivamente o arquivo de imagem a ser utilizado como *height map* e sua devida intensidade na *mesh*.

Para a aplicação genérica do mapa, os véritces do plano que possuiam identificadores por índices compatíveis com uma grelha tiveram tais índices normalizados pelo número de *slices* do plano. O valor normalizado, então, é multiplicado pelo número de pixeis da imagem, tirando então o "floor"da expressão. O resultado desta aplicação será o índice do pixel correspondente àquele vértice na imagem.

```
void setHeightMap(float nLins=0,float nCols=0){
        unsigned int lastVx = vertexList.size()/3;

        if (!(nLins)) nLins = sqrt((float)lastVx);
        if (!(nCols)) nCols = sqrt((float)lastVx);

        //printf("%f\n",nLins);

        for(int h = 0; h < (vertexList.size()/3); h++){
                float pixelH = (float)h/ (float)lastVx;


                float lin = floor(((float)h)/nCols);
                float col = floor((float)h - lin*nCols);

                lin = (lin/nLins)*hmh;
                col = (col/nCols)*hmw;


                int pix = ((int)lin)*hmw;
                pix += ((int)col);
                vertexList[(h*3)+1] += ((float)imageData[pix]/255)*intensity;
        }
}
```

A seguir, como apresentado no código acima, tomamos o valor do pixel na posição especificada, normalizamos o valor para 255, obtendo a intensidade do mapa no ponto específicado. O valor normalizado é então multiplicado pela intensidade passada no atributo de *XML*, obtendo então a nova elevação do ponto no plano.

Neste momento, a aplicação do height map possui bom funcionamento apenas para planos, a implementação, porém, pretende ser extendida para outras figuras. De seguida apresentamos um exemplo do funcionamento desta feature:

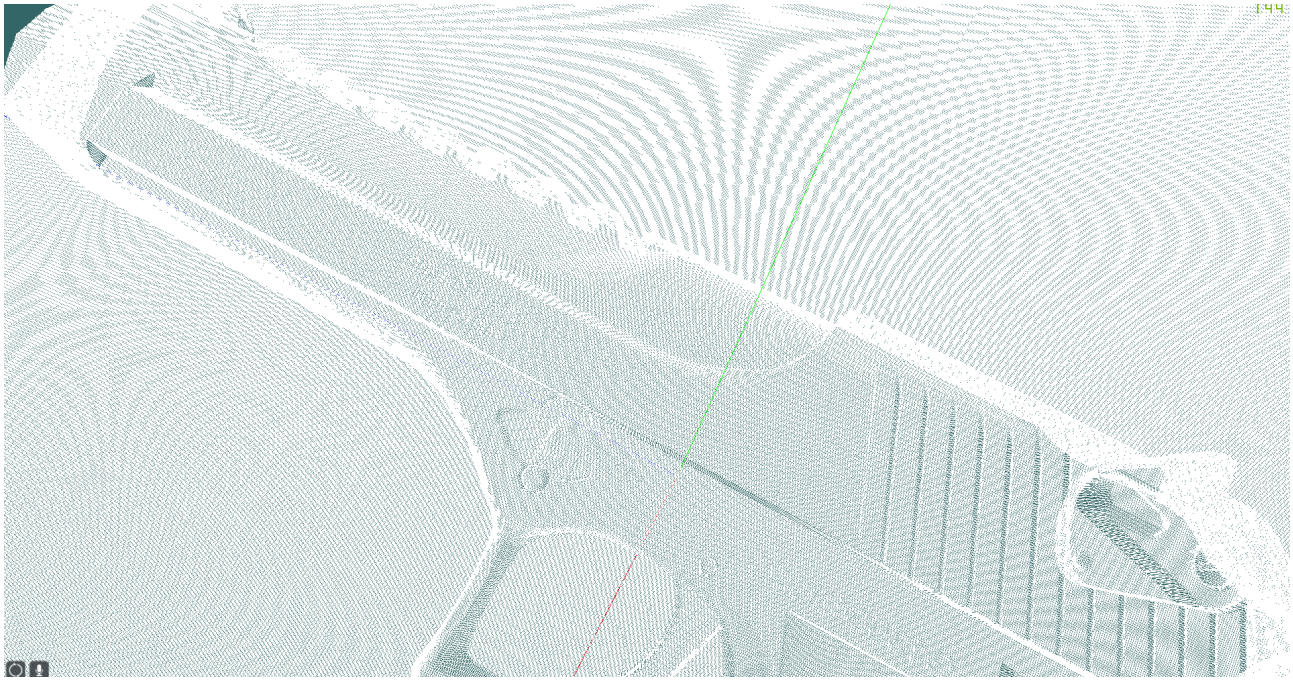

Figura 4.1: Exemplo

4.2 Controles de *Speed*

Dando extensão às animações baseadas em curvas propostas nesta fase, foi desenvolvido um sistema de controle de velocidade para as transformações dependentes de tempo.

Na realização das animações, como referido nos tópicos anteriores, a diferença de tempo entre instantes é medida e as devidas transformações aplicadas consoante o valor de tal diferença. O sistema de velocidade, entretanto trata de manipular o valor de diferença temporal passada às funções de transformação, garantindo controle sobre a passagem de tempo entre as *frames* e permitindo o efeito desejado.

O sistema permite ainda pausar a cena, "congelando"a passagem de tempo entre animações. Para evitar efeitos bruscos no "descongelar"da cena, a passagem de tempo ainda é registrada, mas não contabilizada nas animações, garantindo fluidez ao "descongelar".

Capítulo 5

Conclusão

Foram cumpridos os objetivos desta fase, implementamos as curvas de Catmull-Rom para definir a órbita de um cometa na cena, assim como a órbita dos planetas. Também temos a possibilidade de gerar um novo modelo através de patches de Bézier, especificados por um ficheiro. Em relação aos VBOs, como já fazíamos uso deles para desenharmos os modelos da cena na 1 fase, não precisamos de alterar mais nada. De notar também que a adoção de uma solução orientada a objetos permitiu-nos facilmente implementar os requisitos pedidos devido à modularidade que as nossas aplicações apresentam, pelo que acreditamos que estas serão estendidas de uma forma bastante simples para incorporar a possibilidade de usarmos texturas e iluminação nas cenas.