Universidade do Minho
Mestrado Integrado em Engenharia Informática

Computação Gráfica

Practical Assignment CG - 2019/20
Phase 3: Curves, Cubic Surfaces and VBOs

Maio 2020


Angélica Freitas
(A83761)


Joana Afonso Gomes
(A84912)


Rodrigo Pimentel
(A83765)

Conteúdo

1 Introdução

No âmbito da UC de Computação Gráfica foi-nos proposto o desenvolvimento de um cenário com gráficos 3D de um Sistema Solar. O objetivo da terceira fase deste trabalho prático é, em jeito de continuidade das fases anteriores, na elaboração de animações relativas a translações e rotações.

As translações dos diferentes elementos são definidas tendo por base curvas de *Catmull-Rom* e são efetuadas num determinado período de tempo dado. As rotações passam agora também a ser realizadas em função de um determinado *timing*.

Em paralelo a isto, foi também adicionada uma nova primitiva baseada em *patches de Bezier* (um *Teapot*, que será um cometa neste contexto) .

Com vista a melhorar o desempenho, usamos VBOs com indíces para apoiar o desenho de todos os outros elementos.

No presente relatório incluímos a apresentação pormenorizada das várias vertentes acima apresentadas, com o apoio de gráficos, esquemas, figuras e equações para auxiliar a compreensão do processo de elaboração.

2 Generator

O *Generator* continua a ter a mesma função que nas fases anterior (gerar os pontos dos triângulos que possibilitam a construção dos variados elementos do modelo), passando agora a ter métodos que possibilitam ler e interpretar um ficheiro de *input*, de forma a serem guardados os pontos de controlo, que permitirão depois obter os triângulos criar a superfície.

2.1 Bézier

Para podermos criar métodos que interpretassem o ficheiro de *input* e assim criarmos o *teapot*, foi necessário primeiro entendermos bem o formato do ficheiro *patch*. Este tem uma sintaxe impermutável, de modo que a leitura do mesmo se facilita, sendo imprescindível seguir os seguintes passos.

- Guardar o número de patches correspondente à primeira linha do ficheiro (`nPatches`)

```
    getline(file, line);
    nPatches = atoi(line.c_str());
```

- As próximas "`nPatches`"linhas têm cada uma 16 números inteiros que correspondem aos índices dos pontos de controlo serão guardadas num int**

```
int** index = (int** )malloc(sizeof(int*) * nPatches);

for (int efe = 0; efe < nPatches; efe++) {
    index[efe] = (int *) malloc(sizeof(int) * 16);
}

for (x = 0; x < nPatches;x++)
{
    getline(file, line);
    removeChar(line, ',');

    std::istringstream data(line.c_str());
    int a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p;
    data >> a >> b >> c >> d >> e >> f >> g >> h >> i >> j >> k >> l >> m >> n >> o >> p;

    vector<int> *aux = new vector<int>();
    *aux = { a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p };

    index[x] = aux->data();

}
```

- A linha posterior aos patches contém o número de *control points* precisos (`nPoints`)

```
    getline(file, line);
    int nPoints = atoi(line.c_str());
```

- De seguida são guardados todos "`nPoints`"pontos de controlo.

```
    for (x = 0; x < nPoints; x++)
    {
        getline(file, line);
        removeChar(line, ',');

        std::istringstream data(line.c_str());
        float a, b, c;
        data >> a >> b >> c;
        vector<float>* aux = new vector<float>();
        *aux = { a,b,c };

        allPoints[x] = aux->data();
    }
```

Para obtermos um ponto num determinado parâmetro t através da fórmula de Bezier, criamos a função `bezier` que, através de 4 pontos, nos retorna esse ponto.

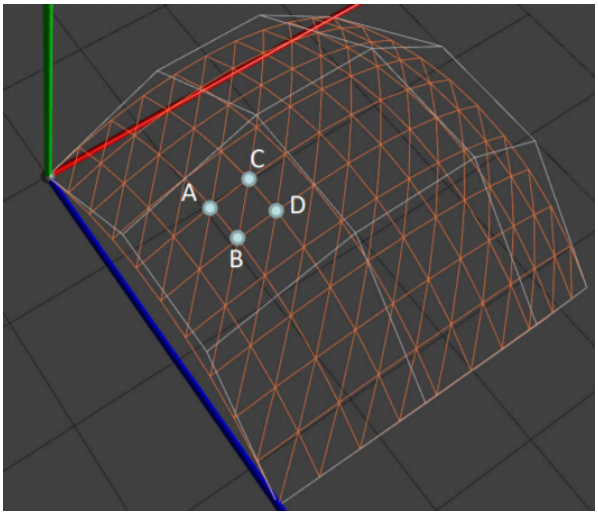
```
float* bezier(float t, float* p0, float* p1, float* p2, float* p3) {
    float* points[4] = { p0,p1,p2,p3 };
    float vectorT[4] = { pow((1 - t),3), 3 * t * pow((1 - t),2), 3 * (1 - t) *
   pow(t,2), pow(t,3) };

    float* res = (float*)malloc(sizeof(float*) * 10);
    multVectorMatrix(points, vectorT, res);
    return res;
}
```

De seguida, iremos a cada Patch e por cada 4 pontos, calcular o respetivo ponto resultante.

```
float* bezierPatch(float u, float v, float** allPoints, int* index) {
    float* controlPoints[4];
    for(int i=0;i<4;i++){
        int idd = index[4 * i];

        float* p0 = allPoints[index[4 * i]];
        float* p1 = allPoints[index[4 * i + 1]];
        float* p2 = allPoints[index[4 * i + 2]];
        float* p3 = allPoints[index[4 * i + 3]];

        float* point = bezier(u, p0, p1, p2, p3);

        controlPoints[i] = point;
    }
    return bezier(v, controlPoints[0], controlPoints[1], controlPoints[2],
  controlPoints[3]);
}
```

Após a leitura do ficheiro *patch*, iremos chamar a função `writeResultPoints` que trata de calcular os pontos Bezier e invoca a `trianglesToFile` para escrever os triângulos.

Building a triangulation for the patch, foi necessário um *float* `step` equivalente a 1/tessellation(dado como argumento). Para cada patch, iremos percorrer uma matriz do género:



Em cada iteração deparámos-nos com 4 pontos essenciais, identificados na imagem acima: A,B,C,D. De modo a invocarmos a função BezierPatch recorremos à criação de criar 4 parâmetros: u,u1,v e v1, tendo valores step*j, step*(j+1), step*k e step*(k+1), respetivamente, sendo que j e k representam números de iteração.

De modo a obtermos os pontos referidos anteriormente, temos de invocar o bezierPatch com os devidos parâmetros u e v. Como é possível determinar através da imagem demonstrada acima, os pontos A,B,C e D tem parâmetros, u-v,u-v1,u1-v e u1-v1, respetivamente. Assim obtemos a função writeResultPoints que nos retornará os pontos necessários:

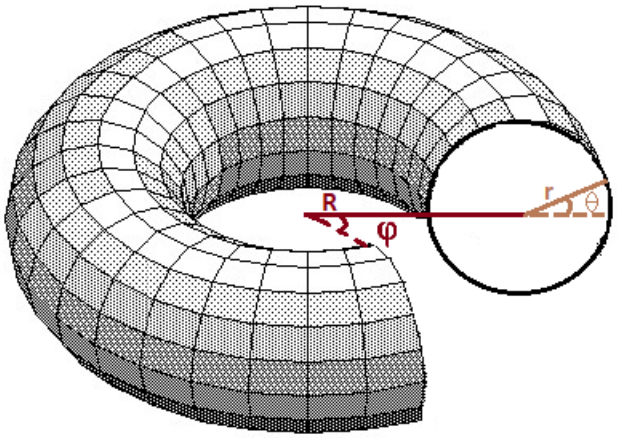
```
1   void writeResultPoints(int tecellationLevel, float** allPoints, int** index, string f) {
2       vector<Triangle> triangles;
3       float step = 1.0 / tecellationLevel;
4       float u, u1, v, v1;
5
6       for (int i = 0; i < nPatches; i++) {
7
8           for (int j = 0; j < tecellationLevel; j++) {
9               for (int k = 0; k < tecellationLevel; k++) {
10                  u = step * j;
11                  u1 = step * (j + 1);
12                  v = step * k;
13                  v1 = step * (k + 1);
14
15                  float* aux = bezierPatch(u, v, allPoints, index[i]);
16                  float* aux1 = bezierPatch(u, v1, allPoints, index[i]);
17                  float* aux2 = bezierPatch(u1, v, allPoints, index[i]);
18                  float* aux3 = bezierPatch(u1, v1, allPoints, index[i]);
19
20                  Point pA(aux[0], aux[1], aux[2]);
21                  Point pB(aux1[0], aux1[1], aux1[2]);
22                  Point pC(aux2[0], aux2[1], aux2[2]);
23                  Point pD(aux3[0], aux3[1], aux3[2]);
24
25                  Triangle t1(pC, pA, pB);
26                  Triangle t2(pB, pD, pC);
27                  triangles.push_back(t1);
28                  triangles.push_back(t2);
29              }
30          }
31      }
32      trianglesToFile(triangles, f);
33  }
```

2.2 Resultados do *Teapot*

Dado o ficheiro de *input* que nos foi dado e um nível de *tesselation* adequado, foi possível obtermos o seguinte *teapot*/cometa.

2.3 Torus

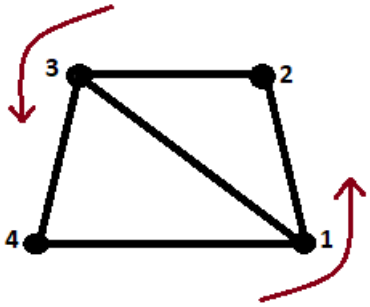
Para o desenvolvimento do anel de Saturno, recorremos a uma forma geométrica designada por torus. Seja `phi` o ângulo que dá a volta ao mesmo, `teta` o ângulo dentro do tubo, **R** a distância entre o centro do *torus* ao centro de cada círculo do seu tubo e **r** o raio desse círculo:



De modo a auxiliar a construção do anel, queremos que o *torus* se encontre "deitado". Para tal, é de fácil compreensão que as coordenadas num ponto arbitrário seguem as seguintes definições.

$$x(\theta,\varphi) = (R + rcos\theta)cos\varphi$$
$$y(\theta,\varphi) = rsin\theta$$
$$z(\theta,\varphi) = (R + rcos\theta)sin\varphi$$

Sendo que o *torus* vai ser dividido em *slices* e *stacks*, necessitamos de um `sliceSkew` e `stackSew` de modo a determinar os ângulos `teta` e `phi` atuais. Em cada iteração temos os seguintes pontos:



Pela imagem acima, sabemos que teremos 2 triângulos cuja ordem dos seus pontos é influenciada pela regra da mão direita. O ponto 1 tem como ângulos *teta* e *phi*. Por sua vez, o ponto 2 passa a ter como seu *teta*, o *teta* do ponto 1 mais o `sliceSkew`, mantendo o *phi*. O ângulo teta no ponto 4 mantém-se igual ao ponto 1, diferindo o ângulo *phi* que passa a ser *phi* (ponto 1) + o `stacSkew`. Intuitivamente, os ângulos do ponto 3 vão ambos diferir e assim obtemos os seguintes pontos:

$$Ponto\ 1 = ((R + rcos(\varphi)) * cos(\theta), (R + r * cos(\varphi)) * sin(\theta), r * sin(\varphi))$$

$$Ponto\ 2 = ((R + rcos(\varphi)) * cos(\theta + sliceSkew), (R + r * cos(\varphi)) * sin(\theta + sliceSkew), r * sin(\varphi))$$

$$Ponto\ 3 = ((R+rcos(\varphi+stackSkew))*cos(\theta+sliceSkew), (R+r*cos(\varphi)+stackSkew)*sin(\theta+sliceSkew), r*sin(\varphi+stackSkew))$$

$$Ponto\ 4 = ((R + rcos(\varphi + stackSkew)) * cos(\theta), (R + r * cos(\varphi) + stackSkew) * sin(\theta), r * sin(\varphi + stackSkew))$$

E assim, com dois simples *for's* obtemos um *torus*.

3 Engine

Assim como na fase prévia, a partir da interpretação de um ficheiro XML, o *Engine* possibilita a apresentação dos modelos numa janela. De modo a cumprir os novos requisitos do projeto, o *parsing* e o *render* sofreram alterações.

3.1 Ficheiro XML

Uma alteração notória nesta fase na constituição dos ficheiros XML foi a introdução de um atributo `time` no ficheiro `SolarSystem.xml`, nas *tags* `rotation` e `translation`, indicando o tempo que demora uma translação / uma rotação inteira. Apresentamos um exemplo no código que se segue, referente ao modelo do planeta Terra.

```
<!--Terra-->
    <group>
        <colour R="33" G="154" B="143" />
        <translate time="1000">
            <point X="29.4" Y="0.0" Z="0.0"/>
            <point X="27.30919" Y="0.0" Z="-10"/>
            <point X="20" Y="0.0" Z="-19.78991"/>
            <point X="10" Y="0.0" Z="-25.39016"/>
            <point X="0.0" Y="0.0" Z="-27"/>
            <point X="-10" Y="0.0" Z="-25.39016"/>
            <point X="-20" Y="0.0" Z="-19.78991"/>
            <point X="-27.30919" Y="0.0" Z="-10"/>
            <point X="-29.4" Y="0.0" Z="0.000000"/>
            <point X="-27.30919" Y="0.0" Z="10"/>
            <point X="-20" Y="0.0" Z="19.78991"/>
            <point X="-10" Y="0.0" Z="25.39016"/>
            <point X="0" Y="0.0" Z="27"/>
            <point X="10" Y="0.0" Z="25.39016"/>
            <point X="20" Y="0.0" Z="19.78991"/>
            <point X="27.30919" Y="0.0" Z="10"/>
        </translate>
        <rotate time="2.74" X="0" Y="1" Z="0" />
        <scale X="0.93" Y="0.93" Z="0.93" />
        <models>
            <model file="sol.3d" />
        </models>
```

Este atributo, assim como todos outros argumentos no ficheiro *XML*, são interpretados no `Engine` no método `auxReadFile`, que recebe de *input* o ficheiro *XML* e um elemento da classe `TransformationState`

```
  void auxReadFile(XMLElement * elem, TransformationState ts)
```

A classe `TransformationState` contém as variáveis e metódos necessários para as diversas transformações no nosso modelo poderem ser concretizadas, contendo, entre outras coisas, os métodos `translate`, `rotate` e `setScale`.

3.2 Curvas Catmull-Rom

3.2.1 Transformações

A inserção dos valores nos *arrays* foi feita na função `getGlobalCatmullRomPoint`, que recorre à função `getCatmullRomPoint`. Esta última recorrer a operações entre uma matriz, dois vetores e os pontos.

$$\mathrm{m} = \begin{pmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1.0 & -2.4 & 2.0 & -0.5 \\ -0.5 & 0.0 & 0.5 & 0.0 \\ 0.0 & 1.0 & 0.0 & -0.0 \end{pmatrix}$$

$$\mathrm{vectorT} = \begin{pmatrix} t^3 & t^2 & t & 1 \end{pmatrix}$$

$$\mathrm{vectordT} = \begin{pmatrix} 3 * t^2 & 2 * t & 1 & 0 \end{pmatrix}$$

A função `getCatmullRomPoint` preenche os *arrays* obtendo os valores através da multiplicação da matriz `m` pelo vetor que contém os **pontos recolhidos do ficheiro XML** (`vectorP`). Com o vetor resultante desta multiplicação e o vetor `vectorT` indicado em cima, obtemos os valores para preencher o vector `pos` (pontos para a próxima translação na curva), e se o multiplicarmos pelo `vectordT`) podemos ocupar o vetor `deriv` com a derivada no ponto.

Assim, com estes dois *arrays* e a necessária variável do tempo, a função `getGlobalCatmullRomPoint` permite-nos obter as coordenadas do próximo ponto da curva para esse dado valor `t` de tempo.

3.2.2 Desenho das curvas

Com as funções listadas acima, utilizando um ciclo *for*, é possível na função `renderCatmullRomCurve` desenhar a curva que pretendiamos.

Definimos o ***tesselation level*** como 500.

```
1   void renderCatmullRomCurve(std::vector< std::vector< float >> basePoints) {
2
3     float res[3];
4     float deriv[3];
5
6     glBegin(GL_LINE_LOOP);
7
8     for (int i = 0; i < tecellation; i++) {
9       getGlobalCatmullRomPoint(i / float(tecellation), res, deriv,basePoints);
10      glVertex3f(res[0], res[1], res[2]);
11    }
12
13    glEnd();
14  }
```

3.3 VBOs

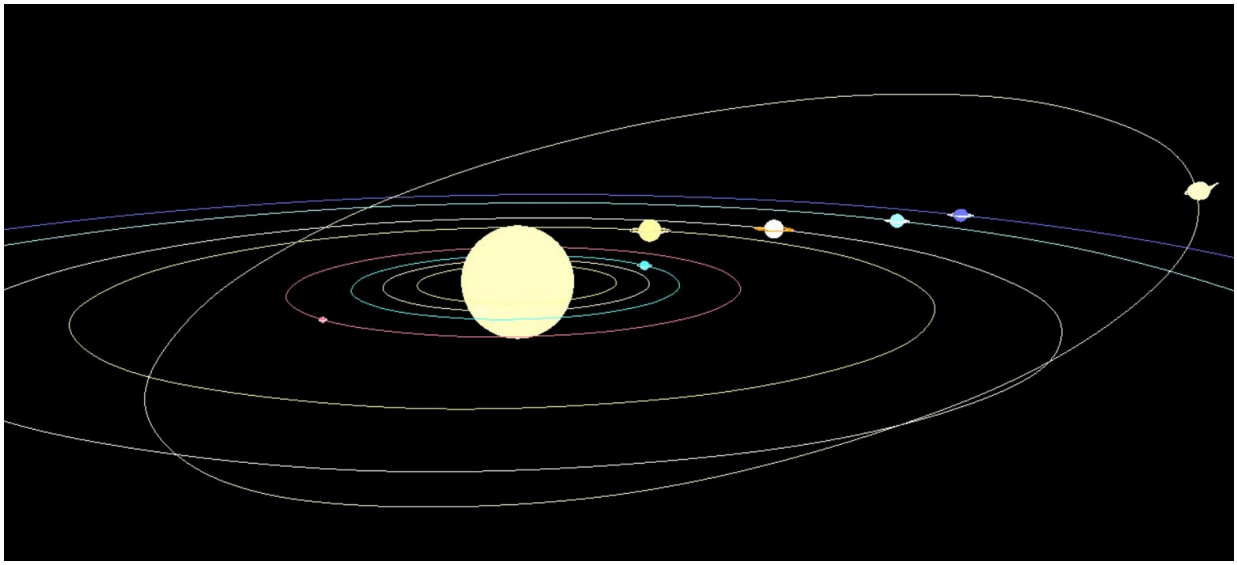
Como já referimos, um dos requerimentos desta nova fase era a transição para VBOs (*Virtual Buffer Objects*) no desenho dos modelos, com vista a uma melhor eficiência, já que guardam os pontos dos triângulos num *buffer* diretamente na placa gráfica, levando a um desenho e a um *parsing* mais rápidos. Deste modo, passaremos a assinalar as mudanças no nosso código para albergar a correta implementação dos VBOs.

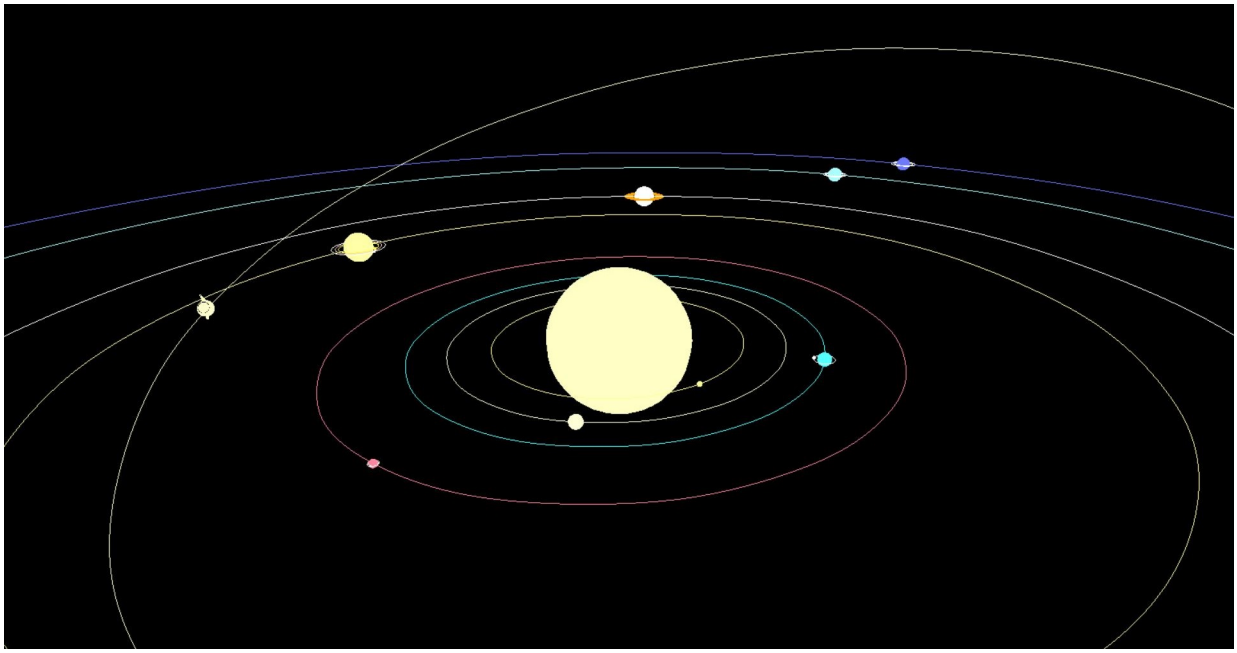
Temos portanto um método `addFile` que preenche um vetor com os pontos para o desenho dos triângulos (a partir de um ficheiro), e que cria um *hashmap* com os pontos desse vetor na ordem correta.

Criamos também o método `drawVBO` que desenha os triângulos com os pontos guardados nesse mesmo *hashmap*.

4 Novo Modelo do Sistema Solar

Nesta secção apresentamos o resultado final do Sistema Solar nesta fase do projeto.





5 Conclusão

Nesta fase foi possível aplicarmos e aprofundarmos conceitos lecionados na UC.

Tornámo-nos familiarizados com o funcionamento das curvas de *Catmull-Rom*, incrivelmente úteis para o cálculo dos pontos de das animação de translação.

A utilização de *patches* de Bezier revelou-se um desafio dado ser o nosso primeiro contacto com a sua aplicação. Aprofundando esta vertente do trabalho possibilitou-nos a perceção das suas de propriedades de construção de primitivas complexas.

De destaque ainda a aplicação de VBOs, que permite uma enorme melhoria em termos de eficiência.

Sentimos que os objetivos propostos foram alcançados, sendo que conseguimos aplicar todas as vertentes apontadas para a criação de um Sistema Solar dinâmico.