



**Universidade do Minho**

**Curvas, Superfícies Cúbicas e VBOs**  
**Unidade Curricular de Computação Gráfica**  
Licenciatura em Ciências da Computação  
Universidade do Minho

Manuel Marques  
(A85328)

Pedro Oliveira  
(A86328)

João Rodrigues  
(A84505)

Eduardo Pereira  
(A84098)

2 de Maio de 2020

# Índice

|          |                                  |          |
|----------|----------------------------------|----------|
| <b>1</b> | <b>Contextualização</b>          | <b>2</b> |
| 1.1      | Enunciado . . . . .              | 2        |
| 1.2      | Resumo . . . . .                 | 3        |
| <b>2</b> | <b>Apresentação das Soluções</b> | <b>4</b> |
| 2.1      | Curvas . . . . .                 | 4        |
| 2.2      | Superfícies Cúbicas . . . . .    | 5        |
| 2.3      | VBOs . . . . .                   | 6        |
| <b>3</b> | <b>Conclusão</b>                 | <b>7</b> |
| <b>4</b> | <b>Anexos</b>                    | <b>8</b> |

# Índices de Figuras

|     |  |   |
|-----|--|---|
| 1.1 | Enunciado . . . . .                                      | 2 |
| 2.1 | Cálculo do ponto da superfície . . . . .                 | 5 |
| 4.1 | Função de cálculo das curvas para a superfície . . . . . | 8 |

# Capítulo 1

## Contextualização

### 1.1 Enunciado

#### Phase 3 – Curves, Cubic Surfaces and VBOs

In this phase the generator application must be able to create a new type of model based on Bezier patches. The generator will receive as parameters the name of a file where the Bezier control points are defined as well as the required tessellation level. The resulting file will contain a list of the triangles to draw the surface.

Regarding the engine, we want to extend the *translate* and *rotate* elements. Considering the translation, a set of points will be provided to define a Catmull-Rom cubic curve, as well as the number of seconds to run the whole curve. The goal is to perform animations based on these curves. The models may have either a time dependent transform, or a static one as in the previous phases. In the rotation node, the angle can be replaced with time, meaning the number of seconds to perform a full 360 degrees rotation around the specified axis.

To measure time the function `glutGet (GLUT_ELAPSED_TIME)` can be used.

```
...
<translate time=10 >
  <point X=1 Y=0 Z=1 />
  <point X=0.707 Y=0.707 Z=1 />
  <point X=0 Y=1 Z=1 />
  ...
  <point X=-1 Y=0 Z=1 />
</translate>
...
```

Note. Due to Catmull-Rom's curve definition it is always required an initial point before the initial curve segment and another point after the last segment. The minimum number of points is 4.

```
...
<rotate time=10 axisX=0 axisY=1 axisZ=0 />
...
```

In this phase it is also required that models are drawn with **VBOs**, as opposed to immediate mode used in the previous phases.

The demo scene is a dynamic solar system, including a comet with a trajectory defined using a Catmull-Rom curve. The comet must be built using Bezier patches, for instance with the provided control points for the teapot.

Figura 1.1: Enunciado

## 1.2 Resumo

Esta fase do trabalho prático era dividida em três temas sobre os quais, de seguida, explicamos a abordagem que tivemos em cada um deles.

- Relativamente às Curvas pretendia-se simular a translação dos planetas à volta do Sol. Para o conseguir criou-se um *vector* para as translações, complementado por uma *struct translate* onde guardamos um valor *float* "time" que representa o tempo que um planeta demora a realizar uma volta completa ( $360^\circ$ ) e um *vector* "pontos". Alteramos o nosso parser do ficheiro *xml* para reconhecer os novos valores que iremos guardar nas *structs* acima definidas. Entraremos em mais detalhe na secção 2.1.
- Para as Superfícies Cúbicas, fizemos *parse* do ficheiro de *patches* para obter-mos os pontos de controlo utilizados no cálculo das superfícies. A explicação do método encontra-se na secção 2.2.
- Por fim, para os VBOs, criamos um *buffer* único para todos os objetos da cena, onde guardamos informações relativas à sua posição inicial e número de pontos. Ver secção 2.3 para mais informações.

## Capítulo 2

# Apresentação das Soluções

### 2.1 Curvas

Como referimos na contextualização, definimos várias estruturas para nos ajudar a atingir o objetivo final. Começamos por definir a *struct* "translate" onde guardamos um *float* "time" que representa o tempo que um planeta demora a concretizar uma volta completa ao sol ( $360^\circ$ ). Criamos também um *vector* "pontos". Em seguida alteramos o parser para poder ler os valores a guardar nas structs como vemos nas figuras a seguir:

```
struct translate{  
    float time;  
    vector<struct point*> cp;  
};
```

(a) Definição da struct

```
<translate time = 20>  
  <point X="10"/>  
  <point X="7.07" Z="-7.07"/>  
  <point Z="-10"/>  
  <point X="-7.07" Z="-7.07"/>  
  <point X="-10"/>  
  <point X="-7.07" Z="7.07"/>  
  <point Z="10"/>  
  <point X="7.07" Z="7.07"/>  
</translate>
```

(b) Exemplo de uma curva no ficheiro xml

Sempre que aparecer a *tag* "translate", criamos a *struct* "translate", atribuímos ao *float* "time" o valor que o *parser* lê e guardamos no vector "pontos" todos os pontos lidos pelo *parser*. Esses pontos serão usados para desenhar a trajetória que o planeta irá traçar. Feito isto no vector "sequencia" (Nota: Este vector já foi utilizado na fase 2 do trabalho) colocamos o valor 4, que refere a uma translação.

Com todos estes dados podemos então traçar a trajetória, recorrendo ao uso de *Catmull-Rom Curves*. Calculamos o resto da divisão do tempo global do sistema pelo valor guardado em "time", multiplicamos esse valor pelo número de pontos e assim obtemos "t", valor referente à posição na curva.

Calculamos o inteiro inferior mais próximo de "t" e usamos esse ponto juntamente com os 3 seguintes para multiplicarmos pela matriz de *catmull-rom* obtendo assim uma curva. Depois utilizando "t" calculamos a posição nessa curva e também a sua derivada, fazemos uma translação para essa posição. Por fim, para manter a direcção do objecto fazemos uma rotação, em que a componente X é a derivada que calculamos, Z é o produto externo entre X e Y, e por fim Y é o produto externo entre Z e X. Com as 3 componentes criamos a matriz de rotação.

## 2.2 Superfícies Cúbicas

Começamos por fazer *parse* do ficheiro de patches de Bézier (no caso deste trabalho, referente ao *teapot*), guardamos os *patches* num vector de vectores, os pontos num vector de pontos (Print do código de *parse* no capítulo Anexos). Depois para cada patch, utilizamos 16 pontos de controlo e para cada divisão, com esses 16 pontos calculamos 4 curvas (4 pontos para cada curva). Com cada uma dessas curvas calculamos 4 pontos. Com os 4 pontos obtidos, calculamos uma nova curva onde vai estar o nosso ponto "final".

```
void evalBezierPatch(const Point *controlPoints, const float &u, const float &v, Point *res) {
    Point uCurve[4];
    for (int i = 0; i < 4; ++i) {
        evalBezierCurve(controlPoints + 4 * i, u, &uCurve[i]);
    }
    evalBezierCurve(uCurve, v, res);
}
```

Figura 2.1: Cálculo do ponto da superfície

Terminados os cálculos de todos os pontos "finais" vamos ter  $(divs+1)*(divs+1)*4$  pontos por *patch*, em que "divs" refere ao número de divisões. Esses pontos formam uma grelha mas, no entanto, nós pretendemos ter triângulos. Para tal, num ficheiro escrevemos o número total de pontos e as coordenadas de cada um na ordem correta para obtermos os triângulos pretendidos. À semelhança de todos os outros modelos que constem no xml, é carregado após ser *parsed*.

## 2.3 VBOs

Para esta parte da 3ª fase, criamos um *buffer* único para todos os objetos da cena, em que, para cada objeto guardamos a sua posição inicial e o número total de pontos utilizados para esse objeto. Quando vamos desenhar o objeto na cena, verificamos a sua posição no buffer e desenhamos o seu número total de pontos. Isto melhora a performance pois, em vez de desenharmos uma figura de cada vez, enviamos todas as informações relativas aos vértices para a GPU, para posteriormente serem desenhados todos os polígonos simultaneamente. O uso de VBOs facilita tanto em termos de código como em processamento.

## Capítulo 3

# Conclusão

Nesta fase tivemos hipótese de trabalhar vários aspetos da Computação Gráfica e aprender a trabalhar com todos eles. Com as curvas aprendemos como atribuir trajetórias aos objetos e como fazer os mesmos terem movimento em relação a essas trajetórias. O uso das curvas de Bezier deu-nos a conhecer um método de desenhar objetos diferente do que estávamos habituados em OpenGL. Por fim, os VBOs oferecem-nos opções que ajudam a tornar o nosso código mais percéptivel e eficiente, ao mesmo tempo que melhoram a performance dos vários programas que criamos.



# Capítulo 4

## Anexos

```
void generate3D(int tessellation) {
    vectorFinal.clear();
    int divs = tessellation;

    int vertices[divs * divs * 4];
    int nvertices = (divs) * (divs) * 4;

    Point *P = new Point[(divs + 1) * (divs + 1)];

    Point *controlPoints = new Point[16]; // matriz de 16 pontos de controle

    for (int np=0; np < patches.size(); ++np) {
        // set the control points for the current patch
        for (int i=0 ; i < 16 ; ++i) {
            controlPoints[i] = *points[patches[np][i]];
        }
        // generate grid
        for (int j=0, k=0; j <= divs; ++j){
            for (int i=0 ; i <= divs; ++i, ++k ){
                evalBezierPatch(controlPoints, i / (float)divs, j / (float)divs , &P[k]);
            }
        }
        int o=0;

        for (int j=0, k=0; j < divs; ++j) {
            for (int i=0; i < divs; ++i, ++k) {

                for (int j=0, k=0; j < divs; ++j) {
                    for (int i=0; i < divs; ++i, ++k) {
                        /*
                        v3-----v0
                        |         |
                        |         |
                        |         |
                        v2-----v1
                        */
                        o +=4;
                        vertices[k * 4] = (divs + 1) * j + i;
                        vertices[k * 4 + 1] = (divs + 1) * (j + 1) + i;
                        vertices[k * 4 + 2] = (divs + 1) * (j + 1) + i + 1;
                        vertices[k * 4 + 3] = (divs + 1) * j + i + 1;
                    }
                }

                vector<struct point*> templ;
                templ.clear();

                for(int i=0; i < nvertices; ++i){
                    Point *a = (struct point*) malloc(sizeof(struct point));
                    a->x = P[vertices[i]].x;
                    a->y = P[vertices[i]].y;
                    a->z = P[vertices[i]].z;
                    templ.push_back(a);
                }

                // bezier to normal
                for(int size=0; size < templ.size()-3; size+=4){
                    vectorFinal.push_back(templ[size]); // v0
                    vectorFinal.push_back(templ[size+3]); // v3
                    vectorFinal.push_back(templ[size+1]); // v1
                    vectorFinal.push_back(templ[size+2]); // v2
                }
            }
        }
    }
}
```

Figura 4.1: Função de cálculo das curvas para a superfície