# Relatório Computação Grafica - Fase 3

Marco Sousa[1,2[62608]], José Malheiro[1,2[93271]], and Miguel Fernandes[1,2[94269]]

[1] Universidade do Minho
[2] Licenciatura em Engenharia Informática, Braga, Portugal

**Resumo** No seguimento lógico da construção de um mundo gráfico virtual estático, vem a evolução para um modelo dinâmico, sendo todos os elementos envolventes definidos com a sua própria animação. Para tal foi necessário extender as transformações desenvolvidas na fase anterior, a rotação e a translação. Ambas devem poder ser realizadas com base num determinado tempo, sendo que a animação recomeça quando alcança o fim. No caso da translação, deve ser permitido construir curvas utilizando as estratégias de *Catmull-Rom*, *Bezier* e *Hermite*, a partir de um conjunto de pontos de controlo. Neste sentido, a transformação deve permitir mover um objeto de acordo com uma curva, ao longo de um determinado tempo. Assim, alterou-se o modelo do sistema solar construído na fase anterior para incluir animações para todos os elementos, sendo feita a adição de um cometa que segue a trajetória de uma curva definida.
Para o desenho do cometa foram utilizadas *Bezier patches* para construir as suas superfícies, sendo passados os pontos de controlo para o desenho.

**Keywords:** OpenGL · GLUT · Figuras Geométricas · 3D · C++ · Tesselation · Bezier Curves · Catmull-Rom Curves · Bezier Patches

## 1 Introdução

### 1.1 Contextualização

No seguimento da fase II do projeto da disciplina de Computação Gráfica da Licenciatura em Engenharia Informática da Universidade do Minho, foi proposta a aplicação de animações no esquema do sistema solar previamente definido, com a possibilidade de desenhar e movimentar objetos ao longo de curvas. Adicionalmente, deveria haver a inclusão de um novo elemento no modelo, um cometa, construído com base em *bezier patches*, para ser animado segundo uma trajetória definida.

### 1.2 Breve Descrição do Enunciado Proposto

O cerne da fase III do projeto implica alterações ao nível do *generator* e do *engine* previamente definidos. Deste modo, pretende-se:

*generator* Adição de um novo tipo de modelo, com base em *bezier patches*.
*engine* Alteração das transformações para permitir a animação do esquema, através da integração do **tempo** a serem realizadas e de **curvas**.

No que toca ao *generator*, o novo modelo deve criar um conjunto de pontos para desenhar os *patches*, ou superfícies, dado um ficheiro *.PATCH* e o nível de tesselação desejado. O ficheiro *.PATCH*, neste caso fornecido pela equipa docente, segue:

*Número de patches* Número de superfícies do objeto a desenhar.
*Patches* Os 16 pontos de controlo para cada *patch*. Haverá tantas linhas como o número de patches em cima definido. Os pontos de cada superfície encontram-se na forma de índices relativamente à posição dos pontos entre sí no ficheiro.
*Pontos* Os pontos usados na construção dos *patches*. São da forma - $xyz$.

O nível de tesselação é utilizado para construir o objeto com um grau de definição concreto e a sua utilização varia o número de vértices finais a serem desenhados. Novamente, as superfícies de um objeto com base neste modelo vão ser desenhadas a partir de triângulos, traduzindo-se no conceito de *triângulos de bezier*.

Relativamente à *engine*, o trabalho nesta implica evoluir as transformações de **translação** e **rotação** previamente definidas:

*Rotação* Permitir que uma rotação receba o tempo que demora a fazer um rotação de 360º.
*Translação* Permitir serem passados os pontos de controlo para constituir a curva e o tempo que demora a percorre-la. O objeto a ser movido terá de seguir esta trajetória e pode ser dada a opção de ele estar alinhado com a curva.

A partir destas alterações apresentar todo o sistema solar como um modelo animado, os planetas, luas e Sol e incluir um novo elemento no modelo, um *cometa*, que tem a sua própria trajetória.

## 2 Trabalho Realizado

Funcionalidades Implementadas:

1. *Generator*
   (a) Gerar triângulos para a construção de superfícies 3D a partir de pontos de controlo. (2.1)
2. *Engine*
   (a) Translação com base no tempo e em pontos de controlo,
      i. através de curvas de *Catmull-Rom/Hermite/Bezier*. (2.2)
      ii. através de uma matriz definida pelo utilizador.
   (b) Rotação em torno de eixo especificado, com base no tempo. (2.2)
   (c) *VBOs*
      i. Renderização do mundo com recurso a *VBOs*. (2.3)
3. Sistema Solar
   (a) Transição de um modelo estático para dinâmico.(2.4)

### 2.1 *Bezier Patches*

Numa fase inicial, houve alguma dificuldade, por parte do grupo, na compreensão sobre a forma de construção das superfícies 3D, através da utilização da estratégia de *Catmull-Rom*. Isto deveu-se, essencialmente, a não se estar a efetuar uma interpolação *bilinear*, nem a construção de triângulos. Apenas de linhas. Depois de se compreender, de facto, como deveria ser a implementação, procedeu-se ao desenvolvimento do código apresentado de seguida.

Neste sentido, após uma análise do material fornecido pela equipa docente foram construídas 3 classes para facilitar a definição dos triângulos das superfícies 3D:

- `class Matrix`
- `class PointMatrix`
- `class BezierTriangles`

**Classe Matrix** Envés de representar as matrizes como apontadores para valores, como foi o caso das aulas práticas, procurou-se definir um módulo que facilmente representaria uma matriz.

**CGDraw** Source Code 1.1: Classe Matrix

```
14 class Matrix
15 {
16 protected:
17         float* _mat;
18         int _m;
19         int _n;
```

Deste modo, encontram-se na classe todas as operações relativamente a uma matriz e entre matrizes. Não obstante a possibilidade de **transpor** e **clonar** uma matriz, foi definida a multiplicação entre a matriz local e outra; como a matriz local pode encontrar-se antes ou depois, recorreu-se ao **polimorfismo** da linguagem para construir os dois métodos considerando as possibilidades. Foram definidos construtores que permitissem a criação através do número de linhas e colunas e aatravés de um conjunto de matrizes default, cf 1.2.

**CGDraw** Source Code 1.2: Enum de tipos de matrizes default

```
7  enum DefMat
8  {
9          CATMULL,
10         BEZIER,
11         HERMITE
12 };
```

...como são sempre estáticos facilita o grupo ao usá-los nos cálculos dos pontos de uma curvas.

**Class PointMatrix** Apesar da `class Matrix` tratar corretamente de matrizes do tipo $float$, decidiu-se criar um outro tipo de matrizes, com cálculo particulares: **matrizes de pontos**. Tendo por base a `class Point` e as necessidades determinadas no desenvolvimento da presente fase, criou-se um conjunto de métodos que permitissem a multiplicação entre uma matriz de valores e uma matriz de pontos, considerando a multiplicação à esquerda e à direita.

Surgiu a `Classe PointMatrix`:

**CGDraw** Source Code 1.3: Matrix

```
6  class PointMatrix
7  {
8          Point* _mat;
9          int _m;
10         int _n;
```

...contêm ainda a possibilidade de transpor e clonar a matriz.

A criação desta classe permitiu a implementação das funcionalidades previstas de uma forma simplificada e elegante, como se poderá ver ao longo do relatório.

**Class BezierTriangles** Com classes criadas no âmbito de realizar operações entre matrizes, procedeu-se à criação de uma última: `class BezierTriangles`. A partir dos pontos de controlo de um *patch*, permite calcular o ponto da curva, de acordo com os vetores **u** e **v** resultante dos nível de tesselação. Neste sentido, foi utilizado o médodo apresentado na figura 1.

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

Figura 1: Cálculo de um ponto da superfício, definida com $P$ pontos de controlo.

Assim, segue-se:

**CGDraw** Source Code 1.4: BezierTriangles

```
5  class BezierTriangles
6  {
7  private:
8          PointMatrix _points;        // 4 x 4
9          PointMatrix _const;                 // 4 x 4 :: M . P . M^T
10         PointMatrix _u_mpm;                 // 1 x 4 :: u . const
11         PointMatrix _mpm_v;                 // 1 x 1 :: .. v
12         Matrix _m;                                  // 4 x 4
13         Matrix _m_t;                         // 4 x 4
14         Matrix _u;                                  // 1 x 4
15         Matrix _v;                                  // 4 x 1
```

Para reduzir o número de cálculos a realizar em cada iteração, considerou-se relevante efetuar um conjunto de pré-calculos que correspondem à multiplicação da transformação de *Bezier*, $M$, com a matriz dos pontos de controlo da superfície, $P$, e, consequentemente, o cálculo da resultante, $MxP$, pela transposta de $M$, $M^T$.

***create_bezier*** No âmbito de calcular todas as superfícies criou-se, no módulo **shapes**, um método capaz de contruir um objeto, recebendo um vetor com os índices dos pontos de controlo de cada *patch*, *pacthes*, um vetor com todos os pontos, *points* e o nível de tesselação, *level*, cf. 1.5.

**CGDraw** Source Code 1.5: create_bezier

```
44 t_points create_bezier(vector<vector<int>> patches, vector<Point> points,
   ↪  int level);
```

**Adicional** A extração de informação encontra-se dentro do módulo **writer**, na função *main* do *generator*, tendo-se utilizado **expressões regulares**, através da biblioteca `<regex>`. Em particular, utiliza-se a expressão 1.6.

**CGDraw** Source Code 1.6: Regex para captura de pontos, utilizando grupos

```
regex regexp(R"(([+-]?\d+(?:\.\d+)?), ([+-]?\d+(?:\.\d+)?), ([+-]?\d+(?:\.\d+)?))");
```

## 2.2 Animação com Transformações Geométricas

A implementação das transformações geométricas podem ser divididas em dois tipos:

- Translação
- Rotação

Para auxiliar a sua implementação, foi criado uma classe que permite calcular a taxa de atualização da renderização. Este valor é essencial para ambos os tipos de transformação.

Especificamente, consiste em ter uma variável que armazene o tempo total que a transformação deve acontecer.

**CGDraw** Source Code 1.7: Class TimeControl

```
6  protected:
7          float _last_refresh;
8          float _refresh_rate;
9          float _total_time;
10 public:
11         TimeControl();
12         TimeControl(float _total_time);
13         void setTime(float total_time);
14         void updateRate();
15         float getRate();
```

Assim, pode-se inicializar com o tempo total de transformação ou definir-se um tempo posteriormente. A cada iteração será executado o `updateRate` para manter o $\_refresh\_rate$ atualizado.

**CGDraw** Source Code 1.8: Atualização da taxa de atualização

```
22         float time = glutGet(GLUT_ELAPSED_TIME) / 1000.0;
23         _refresh_rate += (time - _last_refresh) / _total_time; //calculte
    ↪  the new translate_rate
24         _refresh_rate -= floor(_refresh_rate);
25         _last_refresh = time;
```

**Translação** Implementação realizada através da utilização de duas classes auxiliares: $Curve$ e $TranslateCurve$. A primeira permite o cálculo da posição e da sua derivada. Este, é efetuado com recurso a um vetor de pontos de controlo e utiliza-se a estratégia de escolher um conjunto de 4 pontos que correspondem aos adjacentes do local em que o objeto se encontra. Depois de escolhidos os 4 pontos, é populada uma matriz $4x3$ com os valores $x, y, z$ dos mesmos. O cálculo efetivo consiste em multiplicar a matriz de transformação (predefinida é a Catmull-Roll) pela matriz de pontos criada. Após calcular o vetor $\overrightarrow{t}$ e $\overrightarrow{d}$, este é multiplicado pela resultado do cálculo anterior, originando os valores da posição e da derivada.

A classe $Curve$ é utilizada pela $TranslateCurve$ para calcular a posição e a derivada num determinado instante de tempo. Para tal, utilizou-se as fórmulas de produto externo entre vetores e a sua normalização. Posteriormente, é construído uma matriz de rotação para ser aplicado pelo $glMultMatrixf$.

O instante de tempo é calculado com recurso à classe *TimeControl*.

Para facilitar a visualização das curvas de translação, acrescentou-se a interação com o utilizador através das teclas 'u' e 'U'. A primeira, 'u', ativa a renderização da curva de posição, i.e. o local por onde o objeto vai passar. A segunda, 'U', vai ativar a renderização das linhas de derivada em cada posição.

Por último, apesar das únicas matrizes de transformação utilizadas neste projeto corresponderem às pré-definidas, é possível utilizar matrizes definidas pelo utilizador para serem aplicadas às curvas de translação. Assim, é permitido definir qualquer uma das curvas default através do acréscimo do atributo *curve="0—1—2"* ou personalizadas através do código 1.9.

**CGDraw** Source Code 1.9: XML Exemplo com matriz personalizada

```
<translate time = "10" align="True" curve="-1">
<matrix m="4" n="4">
    <elem>-0.5</elem>
    <elem>1.5</elem>
    <elem>-1.5</elem>
    <elem>0.5</elem>
    <elem>1.0</elem>
    <elem>-2.4</elem>
    <elem>10</elem>
...
</matrix>
```

**Rotação** A aplicação da rotação sobre um determinado eixo, foi conseguida através do cálculo do instante de tempo (cf. *TimeControl*) e a sua multiplicação por 360 deg.

### 2.3 VBOs

No seguimento de terem sido aplicados *VBOs* sem índices na fase II, a implementação dos modelos com o uso de índices era a próxima evolução lógica. Tinha sido previsto progredir para a implementação de índices. No entanto, tal funcionalidade será implementada na fase *IV*.

### 2.4 Modelo do Sistema Solar

O sistema solar estático anteriormente contruído foi atualizado, implementando o movimento de rotação do Sol e dos Planetas à sua volta, bem como dos seus satélites naturais.

**Movimento de Translação** O movimento de translação(de cada planeta à volta do Sol) foi feito com a primitiva *rotate* com um vetor unitário centrado na origem, que representa o centro do Sol, a apontar para $y = 1$.

Neste sentido,

**CGDraw** Source Code 1.10: Movimento de Translação da Terra.

```
<transform>
    <rotate time="36.5" x="0" y="1" z="0" />
</transform>
```

**Movimento de Rotação** O movimento de rotação(de cada planeta à sua volta), também é feito com o *rotate* porém com o vetor centrado no próprio astro, na mesma a apontar para $y = 1$. Os satélites naturais de Terra e Marte, possuem também movimento de rotação e de translação sobre o planeta em que orbitam, sendo estes feitos de forma análoga ao movimento dos planetas. É importante realçar que para tornar o sistema solar o mais perto da realidade, todos os tempos dos movimentos utilizados tem por base os valores reais, aplicando apenas uma escala nestes com o intuito de melhorar a experiência visual do sistema solar.

Deste modo, a escala usada:

$$TempoRot_{x_{modelo}} = (\frac{TempoRotx_{real}}{(10 * 60 * 24)})secs \quad (1)$$

$$TempoTransl_{x_{modelo}} = (\frac{TempoTranslx_{real}}{(60 * 24)})secs \quad (2)$$

. . . sendo os valores em segundos.

**Cometa** No que toca ao movimento do cometa, o grupo inspirou-se no movimento traçado pela trajetória do cometa *Halley*, uma vez que é o mais conhecido do sistema solar. Para representar a sua órbita **elíptica**, foram escolhidos 4 pontos que definem um **losango** e o centro deste foi escolhido de maneira a tentar respeitar o seu movimento, que num período de tempo está muito próximo do Sol e depois permanece afastado durante muito tempo. Desta forma, na *demo-scene* foi utilizada a translação com os 4 pontos que definem uma curva *Catmull-Rom*, que é precedida por um *rotate* com um ângulo de $17.7graus$, de maneira a aproximar a trajetória da rota real do cometa. Rota esta que será seguida por um teapot desenhado com base nos *Bezier patches* fornecidos pela equipa docente.

Assim, os pontos usados:

**CGDraw** Source Code 1.11: Pontos usados para trajetória do cometa.

```
<transform>
<rotate angle="17.7" x="1" y="0" z="0" />
<translate time = "60" align="True" >
        <point x = "0" y = "0" z = "1400" />
        <point x = "-3000" y = "0" z = "-4400" />
        <point x = "0" y = "0" z = "-10200" />
        <point x = "3000" y = "0" z = "-4400" />
```

```
</translate>
<scale x="30" y="30" z="30" />
</transform>
```

. . . onde é possível verificar os 4 pontos de controlo usados para construir a curva de *Catmull-Rom*, bem como a rotação para obter o ângulo desejado. Neste caso, o parâmetro *Align* encontra-se a *True*, pelo que o cometa encontra-se alinhado com a curva na sua trajetória.

**Anéis de Saturno** Uma animação para o anéis de Saturno foi adicionada, para apresentar o **aninhamento de transformações**. Neste caso, são apresentadas como três rotações para permitir os anéis rodarem à volta do planeta. Do modo como a engine encontra-se desenvolvida, as transformações vão ser aplicadas pela ordem que aparecem, sendo permitida esta dinâmica; no caso as rotações estão a ser efetuadas por tempos e ao longo de vetores diferentes.

O excerto do ficheiro *.XML* do sistema solar:

**CGDraw** Source Code 1.12: Rotação dos anéis de Saturno

```
<group>
    <transform>
        <rotate angle="-45" x="1" y="0.45" z="1" />
        <rotate time="3" x="0" y="1" z="0" />
        <rotate time="4" x="0.5" y="0" z="0" />
        <rotate time="10" x="0" y="0" z="0.5" />
    </transform>
    <models>
        <!-- Ring A -->
        <model file="ring_a.3d" />
        <!--generator torus 179 10 15 5 ring_a.3d -->
        <!-- Ring B -->
        <model file="ring_b.3d" />
        <!-- generator torus 199 10 15 5 ring_b.3d -->
    </models>
</group>
```

. . . sendo que, neste caso, as transformações são aplicadas aos dois **tórus**, pelo que ambos rodam do mesmo modo.

## 3 Conclusões

Como consequência do esforço do grupo ao longo do projeto, a implementação de novas funcionalidades permanece facilitada dado à arquitetura desenvolvida, i.e. a modularidade do código traduz-se na fácil inserção de novos módulos ou parâmetros sem afetar a estrutura final. A otimização de ambos programas (*generator* e *engine*) mantém-se um dos objetivos principais do grupo, pelo

que esta organização estrutural do projeto é essencial. Não obstante todos os objetivos a atingir no desenvolvimento deste relatório, manteve-se a certeza de que o consumo de memória não crescesse de modo incontrolável; algo que ocorreu no decorrer da presente fase, nomeadamente no desenho de curvas.

Analogamente às fases anteriores, foi possível implemetar todos os requisitos propostos no enunciado, com a adição de alguns parâmetros extra. É de notar nesta fase:

- Criação de um módulo para a construção de superfícies de *Bezier*.
- Rotação com base num dado tempo.
- Translação por uma curva, dados o tempo e os pontos de controlo.
- Translação com base nos pontos de controlo definidos pelo utilizador, numa matriz.
- Transformação do sistema solar para um modelo dinâmico.
- Adição ao modelo de um cometa, que segue a sua própria trajetória, definida numa curva *Catmull-Rom*.

Um dos pontos mencionados, na forma de trabalho futuro, era relativo à evolução dos *VBOs* para permitirem o uso de índices. Na presente iteração permanece uma componente a adicionar, sendo de alto valor na procura de uma solução otimizada congruentemente.