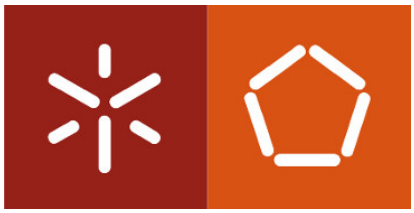

Universidade do Minho

Mestrado Integrado em Engenharia Informática
Licenciatura em Engenharia Informática

Computação Gráfica - Trabalho Prático - Fase 3
Ano Letivo 2021/2022
Grupo 6

Gonçalo Braz (a93178) Simão Cunha (a93262)
Tiago Silva (a93277) Gonçalo Pereira (a93168)

5 de junho de 2022

Resumo

Este relatório inicia-se com uma breve introdução do trabalho a efetuar nesta fase bem como alterações efetuadas na fase anterior. Segue as alterações efetuadas às duas aplicações previamente desenvolvidas: o *generator* e o *engine*. Para o gerador são indicadas as alterações responsáveis por gerar primitivas usando curvas de Bézier e para o motor gráfico são indicadas as alterações realizadas para suportar translações e rotações dinâmicas. Por fim, é apresentado duas cenas do Sistema Solar: uma com apenas o desenho da trajetória do cometa e outra com o desenho de todas as trajetórias dos planetas, terminando este relatório com uma conclusão sobre o trabalho desenvolvido.

Conteúdo

1 Introdução

Este relatório surge no âmbito da terceira fase do trabalho prático da UC de Computação Gráfica. Os objetivos desta fase foram o desenvolvimento de modelos baseados em `patches de Bézier` - construímos um *teapot* desta forma - e o desenho das trajetórias dos modelos - o requisito mínimo foi para desenharmos apenas para o cometa. Para tal, foram extendidas as transformações geométricas `translação` e `rotação` a partir das curvas de Catmull-Rom. Além disso, foram efetuadas algumas correções da fase anterior, que irão ser descritas abaixo.

1.1 Correções da fase anterior

Na fase anterior, notamos ter um pequeno problema na hora de ler as transformações do ficheiro XML, uma vez que as líamos por uma ordem especifica (translate → rotation → scale), que está obviamente errado. Nesta fase, isso foi corrigido e agora as transformações são lidas/executadas pela ordem que aparecem no ficheiro XML.

Com este aspeto retificado, as *demos* que deveriam aparecer no relatório anterior são:

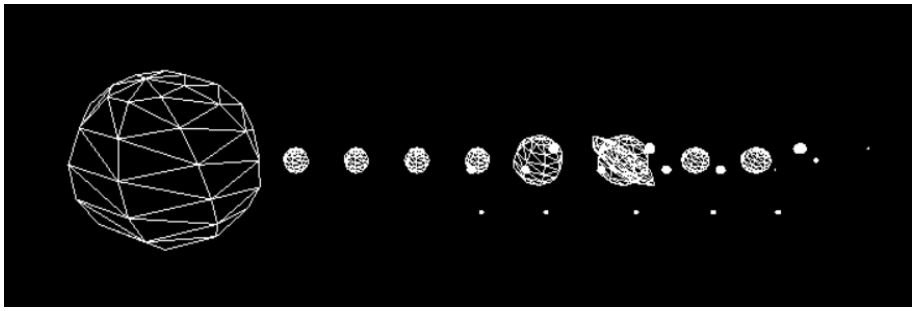

Figura 1: *Demo* da fase II com os planetas em linha reta

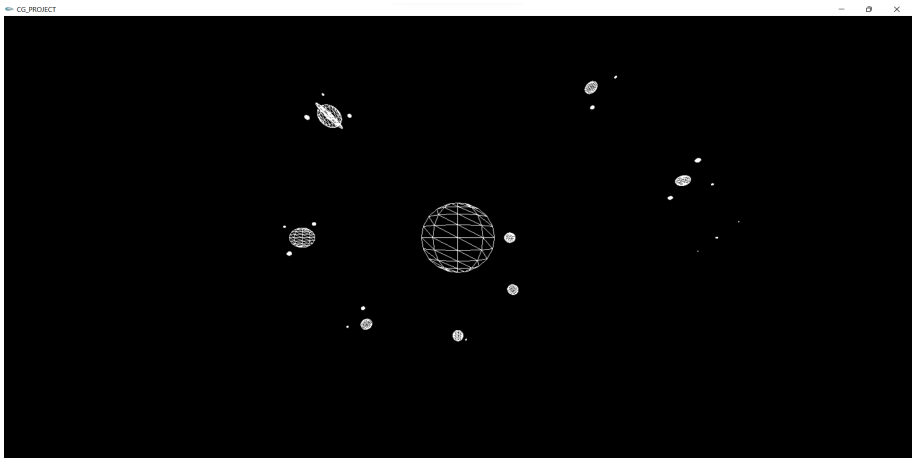

Figura 2: *Demo* da fase II com os planetas em espiral

Nota: Na figura 1, os planetas apenas sofrem translação em relação ao centro do referencial (centro do Sol). Já na figura 2, os planetas sofrem translação e uma rotação de 45^{o} em relação ao planeta anterior.

2 Arquitetura da solução

2.1 Patches de Bézier

`Patches de Bézier` ou `superficies de Bézier` derivam de um conjunto de quatro curvas de Bézier de grau 3 distintas. Cada uma dessas curvas é definida por quatro pontos chamados *pontos de controlo*. Variando um parametro `u`, vai ser calculado um novo ponto em cada curva. Esses novos quatro pontos calculados formam um nova curva. A partir dessa curva, variando um parâmetro `v`, obtêm-se um novo ponto. Esta explicação teórica pode-se traduzir na seguinte fórmula:

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

A matriz M é a chamada `matriz de Bézier` e têm os seguintes valores:

$$M = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{2}$$

Como é possível verificar M é uma matriz simétrica pelo que $M^T = \text{M}$.

2.1.1 Implementação dos patches de Bézier

Antes de qualquer implementação da fórmula descrita acima, é feito o *parse* de um ficheiro que contêm todos os pontos de controlo de todas as superfícies de Bézier que são pretendidas desenhar e que segue o formato específico do ficheiro exemplo fornecido pelos docentes. Depois do *parse* deste ficheiro, surgem duas estruturas de dados:

```
1 vector<vector<float> > controlPoints;
2 vector<int> indexes;
```

Passando agora à implementação, como foi referido na introdução teórica das superfícies de Bézier, existem dois parâmetros `u` e `v` que variam ente 0 e 1 dependendo do nível de tesselação requerido. O nível de tesselação é que marca o número de intervalos entre 0 e 1. Para cada *patch*, ou seja para cada conjunto de 16 pontos (formato matriz) são calculadas os novos pontos variando `u` e `v` como mostra a lógica abaixo. De notar que a matriz dos 16 pontos é passada como 3 matrizes distintas (matrizes das componentes dos pontos).

```
point grid[tessellation+1][tessellation+1];
for(int u=0;u<=tessellation;u++){
    float coords[3];
    for(int v=0;v<=tessellation;v++){
        compute_point((float) u/tessellation,
                      (float) v/tessellation,
                      (float **) pX, (float **) pY, (float **) pZ,
                      coords);
        grid[u][v] = point(coords[0],coords[1],coords[2]);
    }
}
```

A função *compute_point* é que faz a aplicação da fórmula descrita acima para o cálculo dos pontos das superfícies de Bézier. Começa por fazer a multiplicação da matriz M pelo vector V, formando o vector MV. Mutliplicando as matrizes das componentes de P(pX,pY,pZ) obtêm-se a PMV com as mesmas três componentes. Posteriormente, multiplica-se M pelos vetores das três componentes de PMV, formando a matriz MPMV. Por fim, na variável é calculada a multiplicação do vector U pelo vector das componentes e os valores resultantes de cada componente (x,y,z) são guardados no vector coords. Segue-se um pseudo-código da descrição realizada.

```
//MV = M * V
MV = multMatrixVector(M,V);

//PMV = P * MV
PMV[0] = multMatrixVector((float *)pX,MV);
PMV[1] = multMatrixVector((float *)pY,MV);
PMV[2] = multMatrixVector((float *)pZ,MV);

//MPMV = M * PMV
MPMV[0] = multMatrixVector(*M,PMV[0]);
MPMV[1] = multMatrixVector(*M,PMV[1]);
MPMV[2] = multMatrixVector(*M,PMV[2]);

//U * MPMV
for(int i = 0; i < 3; i++)
    for(int j = 0; j <= 3; j++)
        coords[i] += U[j] * MPMV[i][j];
```

Uma otimização que poderia ter sido feita seria pré-calcular a matriz MPM, visto que os pontos de controlo são constantes para cada *patch*. Essa otimização não foi implementada já que a função que usamos para realizar os cálculos só multiplica matriz por vetor pelo que temos que seguir a ordem dos cálculos descrita acima. Numa fase futura facilmente conseguimos efetuar essa otimização, fazendo uso da biblioteca *GLM* que implementa funções para multiplicação de matrizes.

Após o cálculo dos pontos que irão formar a superfície, procede-se à triangulação dos mesmos. A triangulação é feita quadrado a quadrado. Como cada quadrado é formado por dois triângulos são guardados seis pontos de cada vez.

```
for(int u=0;u<tessellation;u++){
       for(int v=0;v<tessellation;v++){

           ps.add_point(grid[u][v]);
           ps.add_point(grid[u+1][v]);
           ps.add_point(grid[u+1][v+1]);

           ps.add_point(grid[u+1][v+1]);
           ps.add_point(grid[u][v+1]);
           ps.add_point(grid[u][v]);

       }
   }
```

Figura 3: Teapot com tessellation = 10

2.2 VBO

Nesta fase, foi-nos requerida que obtivéssemos os vértices das diversas primitivas com recursos a VBO (*Vertex Buffer Object*). No entanto, já implementamos este aspeto na fase anterior e no relatório dessa fase já está descrita como foi o processo.

2.3 Translação

Nesta fase foi adicionada uma vertente diferente de uma translação, isto é, a animação. A estrutura da translação sofreu modificações, já que agora no ficheiro XML, o campo `translate` tem dois atributos *time* (tempo que demora a percorrer a curva/realizar animação) e *align* (um *boolean* para indicar se o objeto em causa está ou não alinhado com a curva). Além disso têm que haver no mínimo quatros pontos especificados dentro do `translate`. Esses pontos é que irão compor a chamada curva de Catmull-Rom.

2.3.1 *Catmull Rom Curves*

As curvas de Catmull Rom são das curvas mais simples de calcular daí serem usadas em animações. O que destingue esta curva das outras é que basta definir um conjunto de pontos por onde queremos que a curva passe sem ser preciso especificar mais nada nem haver a preocupação de como os pontos se ligam. As curvas de Catmull seguem a seguinte fórmula:

$$p(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} P0 \\ P1 \\ P2 \\ P3 \end{bmatrix} \tag{3}$$

Para proceder ao alinhamento do objeto com a curva, constrói-se a seguinte matriz de rotação:

$$\begin{cases} X_i = p'(t) \\ Z_i = X_i * Y_{i-1} \\ Y_i = Z_i * X_i \end{cases} \rightarrow \texttt{Matriz de rotação} = \begin{bmatrix} Xx & Yx & Zx & 0 \\ Xy & Yy & Zy & 0 \\ Xz & Yz & Zz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4}$$

2.3.2 Implementação da curva de Catmull-Rom

De modo a realizar as animações usandos as curvas de Catmull-Rom, desenvolvemos a função *getGlobalCatmullRomPoint* que, dado um instante de tempo t e o conjunto de pontos lidos do XML, devolve a posição do objeto nesse instante e a derivada nesse ponto. A função chama *getCatmullRomPoint* - que é onde encontram as implementadas operações matemáticas abordados acima.

```
// compute MP = M * P
MP[0] = multMatrixVector((float*)M, MP[0]);
MP[1] = multMatrixVector((float*)M, MP[1]);
MP[2] = multMatrixVector((float*)M, MP[2]);

// compute pos = T * MP
for (int i = 0; i < 3; i++) {
    pos[i] = 0;
    for (int j = 0; j < 4; j++)
        pos[i] += T[j] * MP[i][j];
}

// compute deriv = dT * MP
float dT[4] = { 3 * t * t, 2 * t, 1, 0 };
for (int i = 0; i < 3; i++) {
    deriv[i] = 0;
    for (int j = 0; j < 4; j++)
        deriv[i] += dT[j] * MP[i][j];
}
```

Para proceder ao alinhamento do objeto com a curva são realizadas as seguintes operações:

```
//Xi = p'(t)
//deriv = X

//Zi = Xi * Yi-1
cross(deriv,Y,Z);
//Yi = Zi * Xi
cross(Z,deriv,Y);

//todos os vectors têm de estar normalizados
normalize(deriv);
normalize(Y);
normalize(Z);
```

Fazendo uso da derivada previamente calculada e dado um Y inicial definido por defeito, obtemos três vetores normalizados que constituem a matriz de rotação que queremos construir. Estando a matriz construída, fazemos uso da função *glMultMatrixf* (disponibilizada pelo `OpenGL`) para realizar a rotação do objeto.

2.4 Rotação

Nesta fase,foi-nos pedido que adicionássemos uma varíavel de tempo opcional substituta ao ângulo já implementado nas fases anteriores. O tempo seria o número de segundos que demoraria para a rotação completar 360 graus em torno do eixo especificado. Tendo isto em mente, e com a ajuda da função glutGet(GLUT_ELAPSED_TIME) que nos dá o tempo em milisegundos desde que a glutInit foi chamada, fizemos uma simples regra de três simples: se 360 graus está para tempo dado, então o ângulo da rotação a ser efetuado está para o tempo recebido da função glutGet(GLUT_ELAPSED_TIME).

3 *Demos scenes*

Para esta fase, a equipa docente uma *demo*, que consiste numa continuação da cena da fase anterior, mas aqui foi-nos pedido que desenhássemos a trajetória de um cometa, este que vai ser desenhado com `patches de Bézier`, através de `curvas de Catmull-Rom`.

3.1 Sistema solar: órbita desenhada apenas para o cometa *Teapot*

O primeiro passo a tomar para a elaboração desta cena incide no facto de saber quais os pontos iniciais a usar para elaborar a curva de Catmull-Rom. Sabendo que a órbita deste cometa vai estar entre Marte e Júpiter, conseguimos saber o raio da circunferência com centro no Sol - neste caso o raio toma o valor de 55. Tendo este facto em mente, decidimos escolher 8 pontos para definir a curva em questão, tal como exemplificado na figura abaixo - consideramos ser um valor aceitável para a definição da trajetória, mas sabemos que quantos mais pontos escolhermos, mais "redonda"vai ser a circunferência formada:

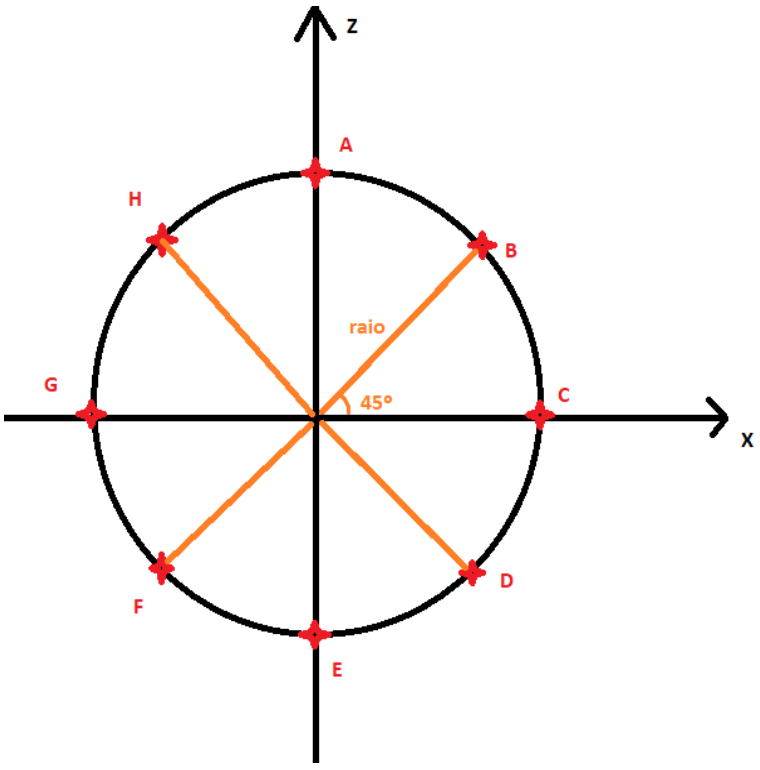

Figura 4: Círculo trigonométrico para descoberta dos pontos (assinalados a vermelho)

De forma a automatizar o processo de descoberta destes pontos, escrevemos um programa em `Python` que devolve 8 linhas em formato `XML` com a informação pretendida. Este calcula cada ponto da seguinte forma:

$$P(x,y,z) = \begin{cases} x = raio * cos(\alpha) \\ y = 0 \\ z = raio * sin(\alpha) \end{cases} , P \in \{A,B,C,D,E,F,G,H\} \quad (5)$$

```python
import math
import sys
n_points = 8
inc = math.pi/(n_points/2) #ângulo em radianos a somar
t = 0
radius= float (sys.argv[1])
for i in range(n_points):
    x = math.cos(t) * radius
    y = math.sin(t) * radius
    if ((x< 1 and x > 0) or (x > -1 and x < 0)):
        x = 0.0
    if ((y < 1 and y> e) or (y > -1 and y < 0)):
        y = 0.0
    t += inc
    print("<point x=\"" + str(x) + "\" y=\"0.0\" z=\"" + str(y) + "\"/>")
```

Listing 1: Programa em `Python` para a escrita dos pontos

Tendo obtido os pontos para a curva de Catmull-Rom para desenhar a trajetória do cometa *Teapot*, decidimos colocar os planetas a rodar em torno do Sol com recurso a um rotação.

Assim, obtivemos a seguinte cena:

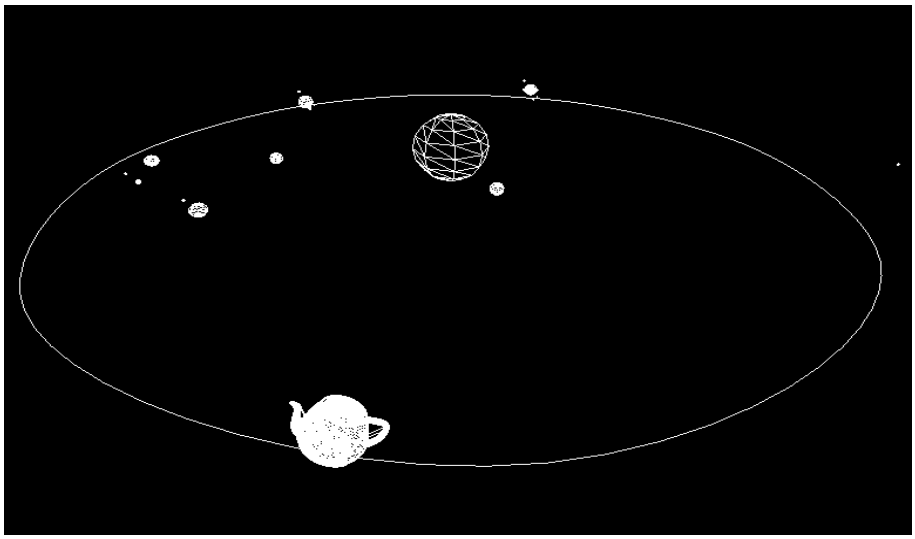

Figura 5: *Demo* com a trajetória do cometa.

3.2 Sistema solar: órbita desenhada para todos os planetas

De forma a generalizar o desenho das trajetórias, decidimos desenhar, da mesma forma que fizemos para o cometa, as órbitas de todos os planetas. Nesta cena, substituímos a *tag* do ficheiro XML `rotation` para `translation`.

Obtivemos, então, a seguinte cena:

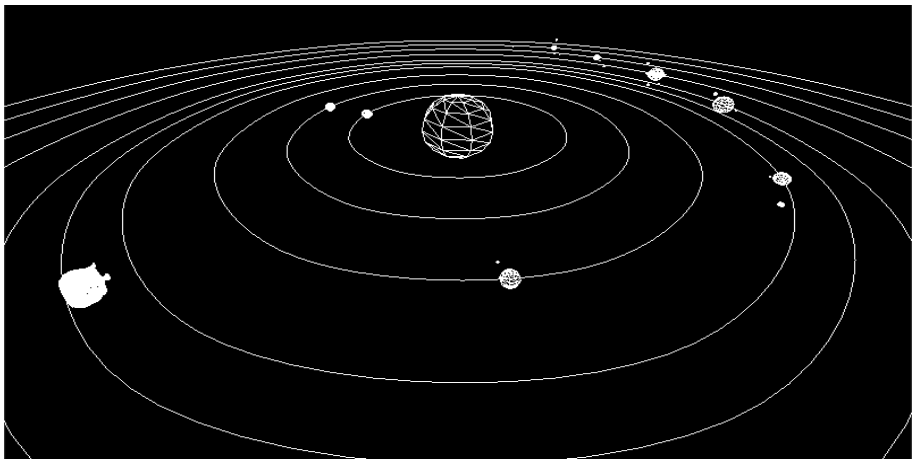

Figura 6: *Demo* com a trajetória de todos os planetas.

4 Conclusão

Nesta fase do trabalho prático, dedicamo-nos na aplicação de conhecimentos teórico-práticos relativamente a curvas de Bézier e Catmull-Rom. Implementámos uma nova forma de desenhar primitivas com recurso a curvas de Bézier, que foi utilizada para gerar um *teapot* dado o ficheiro `teapot.patch`. Além disso, implementámos novas versões das transformações `rotação` e `translação` com recurso a curvas de Catmull-Rom de forma a realizar animações.

Em suma, consideramos que os requisitos definidos pela equipa docente para esta fase do projeto foram alcançados.