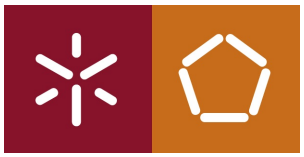

Universidade do Minho

Computação Gráfia

2018/2019

Phase 3 – Curves, Cubic Surfaces and VBOs

Trabalho realizado por:	*Número de Aluno:*
Nelson Gonçalves	A78713
João Aloísio	A77953
César Henriques	A64321
Ricardo Ponte	A79097

20 de Abril de 2019

Conteúdo

1 Introdução

A vertente prática da unidade curricular de Computação Gráfica tem como base a utilização do OpenGL, recorrendo à biblioteca GLUT, para a construção de modelos 3D.

A produção dos modelos referidos envolve diversos temas, entre os quais, transformações geométricas, curvas e superfícies, iluminação e texturas. Com o objetivo de aplicar e demonstrar a aprendizagem destes tópicos, foi proposta a realização de um projeto prático, que se encontra dividido em quatro fases.

A existência destas quatro fases tem como objetivo a evolução gradual do desenvolvimento do projeto, sendo que cada uma das fases deverá respeitar os requisitos inicialmente estipulados no enunciado do trabalho.

Este relatório diz respeito à realização da terceira fase do projeto mencionado, que incide sobre curvas, superfícies cúbicas e VBOs. Para uma melhor compreensão do objetivo desta fase, na secção seguinte será devidamente descrito o problema a resolver.

2 Descrição do Problema

A terceira fase deste projeto tem como objetivo o desenvolvimento de novas funcionalidades aplicadas ao Gerador e ao Motor desenvolvidos na fase 2, sendo que desta vez as novas funcionalidades serão desenvolvidas com recurso a **Bezier patches** e **curvas de Catmull-Rom**. Dito isto, os requisitos estipulados para esta fase são:

- Gerador (recebe como parâmetros o tipo da primitiva gráfica, parâmetros relativos ao modelo e o nome do ficheiro onde vão ser guardados os vértices):
  - **bezier *Bezier_patch_file tessellation file.3d***
    Calcula os pontos necessários, utilizando as fórmulas de Bezier, para desenhar a primitiva com base no ficheiro que contém o *Bezier Patch* e o valor correspondente à tecelagem. Após este processo os pontos calculados são guardados no ficheiro **file.3d**.
- Motor :
  - desenha modelos através de **VBOs**;
  - realiza animações com **rotações/translações**;
  - desenha curvas de **Catmull-Rom**.
- Modelo do Sistema Solar :
  - utiliza a **esfera, o anel e o teapot** como primitivas, para representar o sol, os planetas os satélites e um cometa;
  - utiliza **curvas de Catmull-Rom** para representar todas as trajetórias, quer sejam dos planetas em torno do Sol, dos satélites em torno dos planetas respectivos ou até a trajetória do cometa.
  - utiliza o tempo como medida para representar as rotações do Sol, dos planetas e dos satélites sobre si mesmos.

3 Resolução do Problema

Nesta fase serão realizadas alterações, tanto ao Motor como ao Gerador desenvolvidos anteriormente, com o objectivo de cumprir com os requesitos propostos para a terceira fase do projeto.

3.1 Gerador

Nesta fase o gerador tem de ser capaz de desenvolver um novo modelo baseado num **Bezier Patch**. Posto isto, desenvolveu-se uma primitiva que calcula os pontos necessários para desenhar o modelo final baseado num ficheiro que contém os pontos de controlo.

3.1.1 Bezier Patch

Para além do ficheiro que contém o *patch* de Bezier, é também necessário fornecer como argumento o valor de tecelagem e o ficheiro onde vão ser escritos os pontos calculados. O ficheiro de input será lido sendo que primeiramente serão lidos o número de patches seguidos dos indices de cada um. Estes indices serão guardados num array bi-dimensional de floats, correspondendo assim a uma matriz. O próximo passo será percorrer cada um dos *patches* e armazenar os pontos respetivos a cada um num array bi-dimensional.

Após isto será invocada a função *calculate_control_points* que receberá como argumento o array criado anteriormente e calculará 3 matrizes produto, sendo que cada uma corresponde às coordenadas x,y e z.

Nesta função serão criadas 9 matrizes (4x4) temporárias , sendo que as fórmulas de cálculo dessas matrizes são :

- temp_matrix_x[i][j] = controlPoints[k1][0];
- temp_matrix_y[i][j] = controlPoints[k1][1];
- temp_matrix_z[i][j] = controlPoints[k1][2];
- temp_const_matrix_x[i][j] = m[i][k] * temp_matrix_x[k][j] ;
- temp_const_matrix_y[i][j] = m[i][k] * temp_matrix_y[k][j];
- temp_const_matrix_z[i][j] = m[i][k] * temp_matrix_z[k][j];
- temp_prod_matrix_x[i][j] = temp_const_matrix_x[i][k] * m[k][j];
- temp_prod_matrix_y[i][j] = temp_const_matrix_y[i][k] * m[k][j];
- temp_prod_matrix_z[i][j] = temp_const_matrix_z[i][k] * m[k][j];
- i,j,k $\in [0, 3]$;
- $k1 = [0, N]$, N = number of control points.

O array bi-dimensional de floats **m[][]** corresponde à matriz de Bezier que é dada por:

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

A matriz *temp_prod_matrix* é então dada seguindo as expressões abaixo e tendo em conta o valor de M.

$$CP = \begin{bmatrix} x_{00} & x_{01} & x_{02} \\ x_{10} & x_{11} & x_{12} \\ x_{20} & x_{21} & x_{22} \\ x_{30} & x_{31} & x_{32} \\ \vdots & \vdots & \vdots \\ x_{k0} & x_{k1} & x_{k2} \end{bmatrix}$$

- temp_prod_matrix $= CP \times M \times M^T$

Sendo que CP corresponde à matriz dos pontos de controlo lidos e as matrizes *temp_prod_matrix_x*,*temp_prod_matrix_y* e *temp_prod_matrix_z* dizem respeito,respecitvamente, às componentes x,y,z da matriz calculada.

As matrizes produto temporárias depois são armazenadas como variáveis globais (prod_matrix_x , prod_matrix_y e prod_matrix_z) e os seus valores serão utilizados na função de cálculo de um *patch* de Bezier *calculate_Bezier_Patch*.

Após os armazenamentos das matrizes produto serão então calculados os pontos. De seguida será apresentado o algoritmo de cálculo dos pontos com auxílio da figura abaixo.

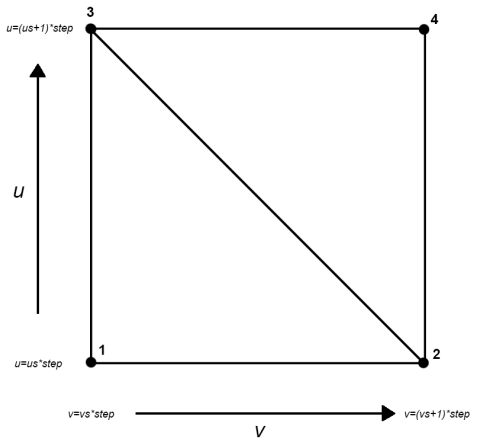

Figura 1: Representação do método de cálculo dos pontos dos triângulos com base em u e v.

Tendo em conta as fórmulas apresentadas, vamos calcular as coordenadas x, y e z de cada um dos pontos, sabendo que o que varia para cada um deles é o valor de u e v. Para além disto é preciso ter em conta no cálculo dos pontos 1,2,3 e 4 que:

- $step = \frac{1}{tecelagem}$
- $us = [0, tecelagem]$
- $vs = [0, tecelagem]$

Para cada $x \in [0, tecelagem]$ será calculado u e v, como também os pontos descritos na figura. Isto é feito na função *calculate_Bezier_Patch* que recebe como argumento os ponto u e v como também um indice

que indicará que coordenada (x,y,z) deve ser calculada. Neste caso se o indice for 0, será calculado o ponto x do *patch* de Bezier, caso seja 1 calcula-se o ponto y e finalmente se for 2 calcula-se o ponto z. Após receber os argumentos a função irá multiplicar a matriz produto (*prod_matrix*) da respectiva coordenada (x,y,z) selecionada ,pela matriz linha *u_line* e pela matriz coluna *v_column* sendo que estas são definidas por:

$$u\_line = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix}$$

$$v\_column = \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix}$$

Feito este processo, retorna-se o valor de *patch* calculado. Este valor de patch corresponderá a uma das coordenadas dos pontos 1,2,3 e 4 representados na figura 1. O que significa que serão calculadas as coordenadas x,y e z de cada um dos pontos, em cada iteração. Após estarem calculados os pontos, os triângulos serão desenhados tendo em conta a regra da mão direita. Dito isto, desenha-se o primeiro triângulo com os pontos 1,2 e 3, respetivamente , sendo que o segundo triângulo foi desenhado com os pontos 3,2 e 4. Nas figuras abaixo encontra-se o *patch* de Bezier gerado (neste caso o ficheiro de input foi o *teapot*) desenhado com diferentes tecelagems.

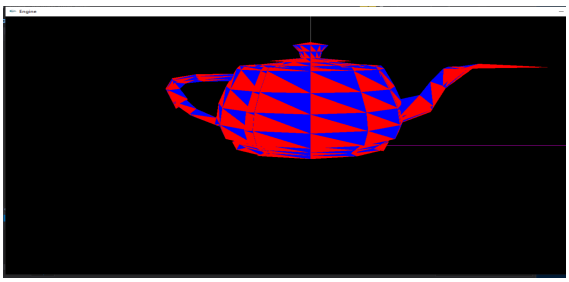

Figura 2: Representação do *teapot* gerado , com tecelagem 3

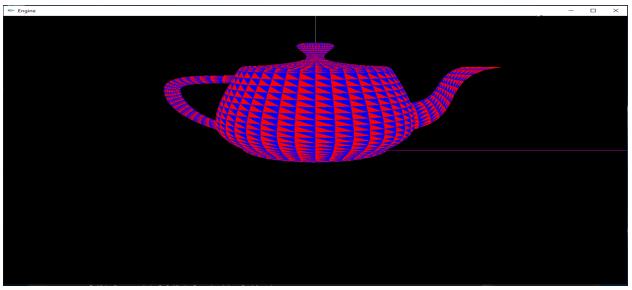

Figura 3: Representação do *teapot* gerado , com tecelagem 10

3.2 Motor

Senddo que nesta fase é necessário que o *Engine* desenhe curvas, é necessário que este armazene mais dados, tais como, os pontos de controlo da curva, o tempo de animação da curva e tempo de rotação do modelo. De referir as variáveis:

- *bool curved*: Esta variável vai permitir a distinção entre um modelo dinamico e um modelo estático.
- *ControlP contp*: Esta variável é um vector de 4 ou mais Vertex (caso seja dinamico), sendo que Vertex é uma estrutura de 3 *float* (x,y,z).

```
typedef std::vector<Vertex> ControlP;

typedef struct groupData{
    float traX, traY, traZ;
    float rotX, rotY, rotZ;
    float scaleX, scaleY, scaleZ;
}Group;

typedef struct pathInfo{
    Path path;
    float traX=0, traY=0, traZ=0;
    float rotX=0, rotY=0, rotZ=0;
    float scaleX=1, scaleY=1, scaleZ=1;
    ControlP contp;
    int nrcontp;
    bool curved;
    float trans_time=0;
    float rot_time=0;
}Paths;


typedef struct modelData{
    Model model;
    float traX=0, traY=0,traZ=0;
    float rotX=0, rotY=0, rotZ=0;
    float scaleX=1, scaleY=1, scaleZ=1;
    ControlP contp;
    int nrcontp;
    bool curved;
    float trans_time=0;
    float rot_time=0;
}ModelData;
```

Posto isto, foi declarada uma variável global que irá ser o array de buffers, permitindo a aplicação de *VBO's* e uma variável *number_of_groups* que irá permitir percorrer o array *paths* e armazenar os valores do *time* na devida posição.

```
GLuint *buffers;
```

3.2.1 Leitura do ficheiro XML

Nesta fase foram feitas pequenas alterações à leitura do ficheiro XML, sendo estas o armazenamento do atributo da variável *time* na estrutura *Paths* ao encontrar *rotate*, e no caso do *translate* se for encontrado a variável *time* é armazenado o atributo e alterado a valor para *true* da variável *curved* na estrutura *Paths* no devido índice. Para armazenar os pontos de controlo foi implementado uma função auxiliar *readPoints*.

3.3 Utilização de VBO's

Nesta fase era também pedido a aplicação de *Vertex Burffer Objects*, que permite a eficiência no desenho dos pontos necessários por partes da placa gráfica, visto que estes são passados diretamente para a placa gráfica e não triângulo a triângulo.

3.3.1 Inicialização

A inicialização dos *VBO's* é feita da seguinte forma:

- Tendo todos os modelos carregados nas estruturas, a variável buffer é inicializada com um tamanho igual ao número de modelos lidos;
- É chamada a função *glEnableClientState(GL_VERTEX_ARRAY)*, para que seja possível a utilização de arrays no desenho dos vértices e também é chamda a função *glGenBuffers(paths_size,buffers)*, para que sejam gerados os buffers que irão armazenar os pontos dos modelos;
- Posto isto, é chamada a função *preparaBuffers()* que tem como objectivo inserir os vértices dos modelos nos buffers, sendo que cada modelo tem um buffer, isto através das funções *glBindBuffer()* que liga o devidoo buffer ao devido id e *glBufferData()* que copia os dados para o buffer.

3.3.2 Desenho

Com a fase de inicialização finalizada, é necessário passar passar para a fase de desenho que é feita da seguinte forma:

- Para cada modelo:
  1. É feita a ligação para cada buffer com a função *glBindBuffer(GL_ARRAY_BUFFER, buffers[j])*;
  2. É indicado o tipo de apontador que está nos arrays, com a função *glVertexPointer(3, GL_FLOAT, 0, 0)*;
  3. É pedido o desenho do modelo, através da função *glDrawArrays(GL_TRIANGLES, 0, model.size())*;

3.4 Curvas de Catmull-Rom

De modo a animar as curvas de Catmull-Rom foi criado outro módulo de suporte para efetuar esses cálculos. Nas secções seguintes são explicadas as funções que tornam possível a animação com as curvas referidas.

3.4.1 Cálculo da posição

Para ilustrar as curvas de Catmull-Rom primeiro é necessário calcular a posição do objeto (P(t)). Esse cálculo será feito com base na matriz pré-definida de Catmull-Rom e com 4 pontos pertences à curva formada por P(0), P(1), P(2) e P(3), dado um instante t. Segue-se abaixo as definições utilizadas para este cálculo.

$$T = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix}$$

$$M = \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

$$P = \begin{bmatrix} P_{0x} & P_{0y} & P_{0z} \\ P_{1x} & P1_{1y} & P_{1z} \\ P_{2x} & P_{2y} & P_{2x} \\ P_{3x} & P_{3y} & P_{3z} \end{bmatrix}$$

Depois destes valores serem conhecidos, serão aplicadas as fórmulas enunciadas abaixo:

$$A = P \times M$$

$$P(t) = A \times T$$

Estes cálculos são feitos na função **getCatmullRomPoint**. É de realçar, que de forma semelhante ao cálculo feito nos patches de Bezier, as matrizes definidas acima estão divididas em sub-matrizes representativas das respetivas coordenadas x,y e z (à exceção da matriz M e da matriz T). Tendo isto em conta os cálculos serão efetuados na seguinte ordem:

$$A_x = Px \times M$$

$$A_y = Py \times M$$

$$A_z = Pz \times M$$

$$P(t)_x = Ax \times T$$

$$P(t)_y = Ay \times T$$

$$P(t)_z = Az \times T$$

3.4.2 Cálculo da matriz rotação

Tendo feito o processo de cálculo da posição do objeto será agora necessário calcular a sua derivada de modo a obter a matriz de rotação que alinha o objeto com a curva. As fórmulas abaixo foram usadas para este cálculo, sendo apenas necessário recorrer à matriz T' e à matriz A, foi possível obter a derivada da posição, P'(t). Note-se que a matriz A diz respeito à matriz calculada para a posição do objeto.

$$T' = \begin{bmatrix} 3t^2 & 2t & 1 & 0 \end{bmatrix}$$

$$P'(t) = A \times T'$$

Como as matrizes P'(t) e A encontram-se divididas em sub-matrizes onde estão armazenadas as suas coordenadas x,y e z , respetivamente, as fórmulas de cálculo adaptadas agora serão:

$$P'(t)_x = A_x \times T'$$

$$P'(t)_y = A_y \times T'$$

$$P'(t)_z = A_z \times T'$$

Para obter a matriz de rotação, precisou-se de calcular X,Y e Z , sendo estes dados pelas fórmulas enunciadas abaixo:

$$X = ||P'(t)||$$

$$Y_i = \frac{Z \times X}{||Z \times X||}$$

$$Y_0 = (0, 1, 0)$$

$$Z = \frac{X \times Y_{i-1}}{||X \times Y_{i-1}||}$$

Tendo isto feito, de seguida , foi utilizada a função **build_Rotation_Matrix** e, posteriormente a **glMultMatrixf** para multiplicar a matriz de rotação pela atual. O cálculo de P'(t) é também realizado na função **getCatmullRomPoint**.

3.4.3 Desenho da trajetória

A função **renderCatmullRomCurve** é responsável pelo desenho da trajetória. Esta função recebe um conjunto de pontos pertencentes à curva e de seguida passa-os como argumento à função **getGlobalCatmullRomPoint** para calcular as diferentes posições dos restantes pontos pertencentes à curva.

Como é demonstrado pela figura acima, a função **getGlobalCatmullRomPoint** é chamada para valores de $i \in [0.01, 100[$ com um incremento de 0.01 por iteração.Esta função terá 10000 iterações sendo assim gerados 10000 vértices pertencentes à curva. Estes vértices serão ligados por uma linha, isto será gerado recorrendo à primitiva *GL_LINE_LOOP*.

```
void renderCatmullRomCurve(float controlPoints[][3], int number_points) {
    float deriv[3];
    glBegin(GL_LINE_LOOP);
    for(float i=0.01; i< 100; i += 0.01) {
        getGlobalCatmullRomPoint(i,pos,deriv, controlPoints, number_points);
        glVertex3f(pos[0],pos[1],pos[2]);

    }
    glEnd();

}
```

3.4.4 Cálculo do tempo global t

Visto que agora a translação vai estar dependente do tempo será necessário calcular, a cada instante, qual o instante *global_t* atual para que os planetas e satélites possam se mover de acordo com o mesmo. Este tempo será cálculado de acordo com a seguinte fórmula:

$$global\_t = \frac{glutGet(GLUT\_ELAPSED\_TIME)}{1000 * m.translate\_time}$$

Esta fórmula relaciona o tempo que já passou desde o *render* inicial da cena (dado pela variável $GLUT\_ELAPSED\_TIME$) com o tempo de translação do modelo (em segundos, dado pela variável $m.translate\_time$), permitindo assim que a translação seja feita iterativamente em função do tempo passado.

3.5 Modelo do Sistema Solar

No modelo do sistema solar, descrito no ficheiro XML, serão representados o Sol, os 8 planetas (Mercúrio, Vênus, Terra, Marte, Júpiter, Saturno, Urano e Neptuno). Para representar os planetas foram utilizadas as mesmas escalas da fase anterior, no entanto, foi preciso obter novas escalas para obter os tempos de rotação e translação dos planetas.
Para os tempo de translação foi definido o tempo máximo de 180 segundos que corresponde ao Neptuno, que é o planeta que tem um maior tempo de translação, tendo isto em conta obtemos os tempos dos restantes planetas. No entanto como a maior parte dos tempos eram demasiado curtos, estes foram aumentados de forma a tornar o modelo mais operceptíve.
Para o tempo de rotação foi utilizado o mesmo método de escala mas desta vez o tempo máximo é de 100 segundos que corresponde ao planeta Vénus.

Nome	**Rotação(segundos)**	**Translação(segundos)**
Sol	50	-
Mercúrio	60	70
Vénus	100	80
Terra	10	95
Marte	12	105
Júpiter	6	130
Saturno	7	150
Urano	7.5	170
Neptuno	8	180

Tabela 3: Tempos de rotação e translação de cada planeta.

De seguida, definimos as curvas de Catmull-Rom, para cada curva vamos calcular 8 pontos tendo em conta a distancia do planeta ao Sol que vai definir o raio da circunferência.
Para calcular os pontos vamos definir d como a distancia do plaena ao sol.

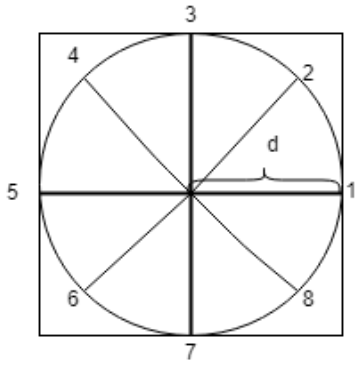

Figura 4: *Pontos da trajétoria dos planetas.*

- **1** x = d; z=0
- **2** x = $\frac{d\sqrt{2}}{2}$; z = -$\frac{d\sqrt{2}}{2}$

- **3** x = 0; z = -d
- **4** x = -$\frac{d\sqrt{2}}{2}$; z = -$\frac{d\sqrt{2}}{2}$
- **5** x = -d; z = 0
- **6** x = $\frac{d\sqrt{2}}{2}$; z = $\frac{d\sqrt{2}}{2}$
- **7** x = 0; z = d
- **8** x = $\frac{d\sqrt{2}}{2}$; z = $\frac{d\sqrt{2}}{2}$

Em relação á trajectória do cometa foi utilizada este metodo, no entao, adicionamos valores á componente ***Y*** de maneira a tornar a sua orbita menos linear.
As

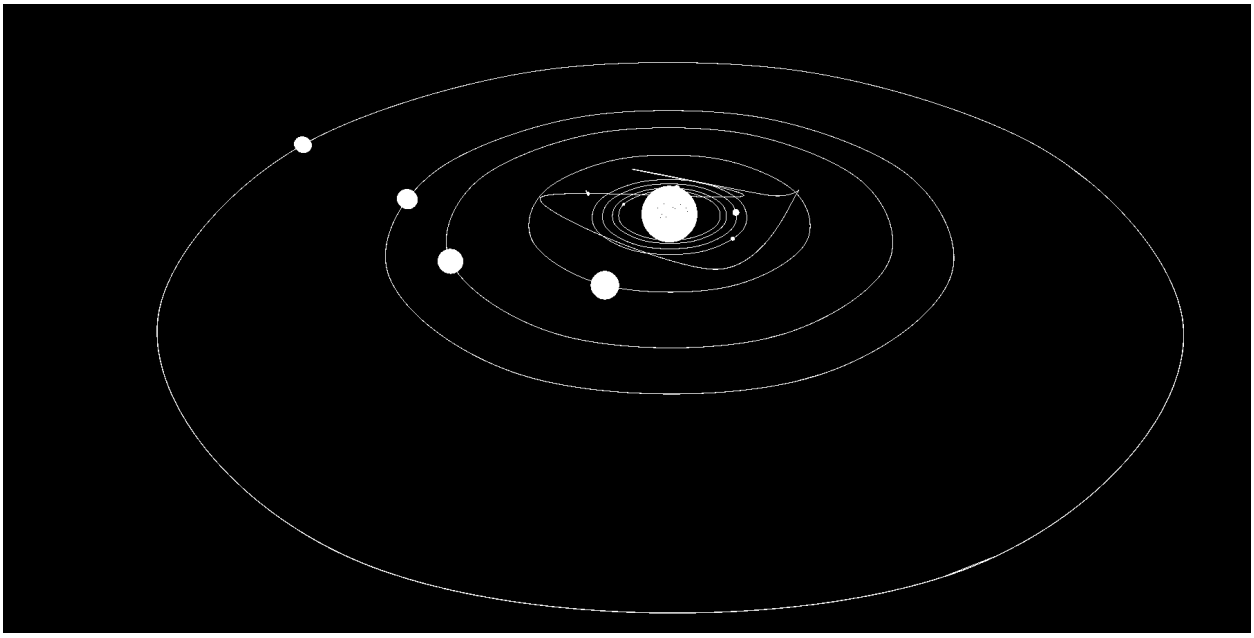

Figura 5: *Modelo do Sistema Solar*

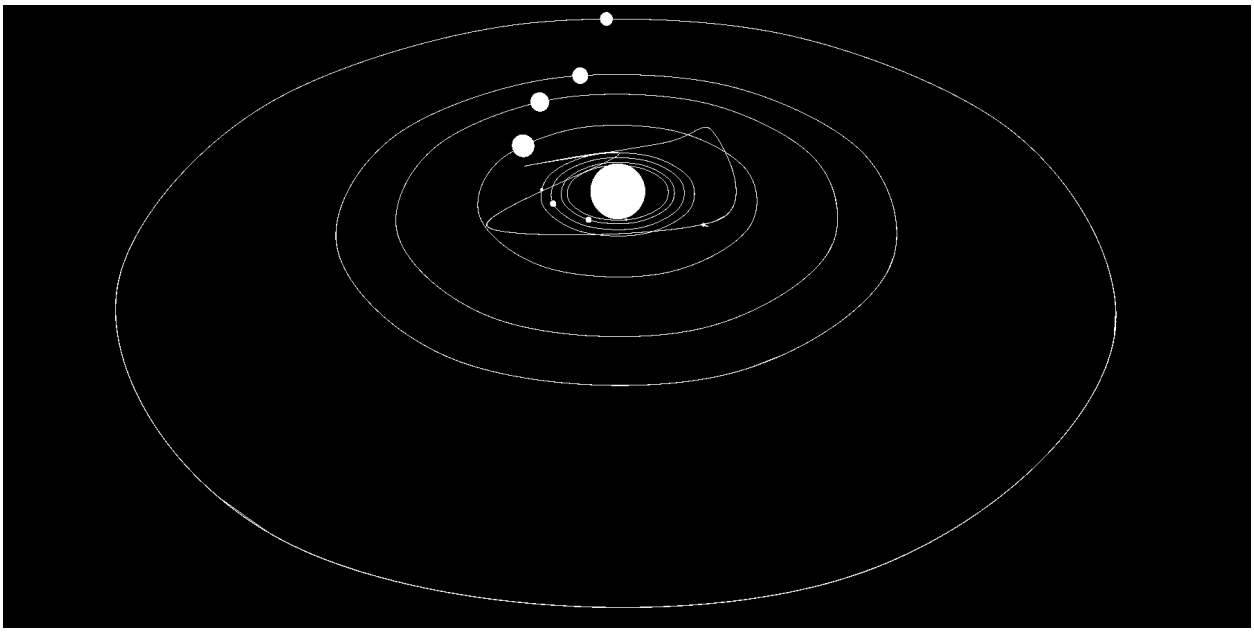

Figura 6: *Modelo do Sistema Solar*

4 Conclusões

Após a finalização do trabalho que conclui a terceira fase do trabalho prático, é possível obter uma avaliação crítica sobre o mesmo.

Os resultados conseguidos pelo grupo são satisfatórios, visto que conseguimos um engine capaz de executar animações baseadas em tempos e estáticas, possibilitando assim a criação da cena "Sistema Solar", que demonstra a rotação dos planetas em torno do Sol e a rotação sobre si mesmos.

De referir também as dificuldades que o grupo encontrou, tais como, as fórmulas de rotação, a contrução do cometa utilizando os patches Bezier que foram ultrapassadas. Porém, não foi possível executar a rotação da lua em torno da terra, a qual se encontra em falta na cena.

Concluindo, o trabalho desenvolvido apresenta uma qualidade razoável e responde aos critérios pedidos inicialmente, apesar deste não estar completo, principalmente pela situação da rotação das luas em torno dos seus planetas, sendo assim esta a fase que apresentou mais dificuldades até ao momento.