Computação Gráfica

3º ano de LCC/MIEI

3º Fase de Entrega

Relatório de Desenvolvimento

Curves, Cubic Surfaces and VBOs

João Gomes

a70400

João Dias

A72095

Joel Morais

A70841

Luís Ventuzelos

a73002

1 de Maio de 2017

Resumo

Este documento apresenta o desenvolvimento do projeto de *Computação Gráfica* numa colaboração entre o Mestrado Integrado de Engenharia Informática *(MIEI)* e a Licenciatura de Ciências de Computaçao *(LCC)* correspondente ao ano lectivo *2016/2017*, analisando as estratégias utilizadas e as principais funções implementadas em cada uma das tarefas propostas.

O objetivo principal nesta nossa fase consiste em usarmos VBO (vertex buffer object), de modo a conseguirmos desenhar os objetos necessários. Irá ser preciso recorremos às superfícies de bézier de modo a atingirmos o objetivo de desenhar o nosso teapot, e por último, compete-nos implementar curvas a partir do método de catmull rom.

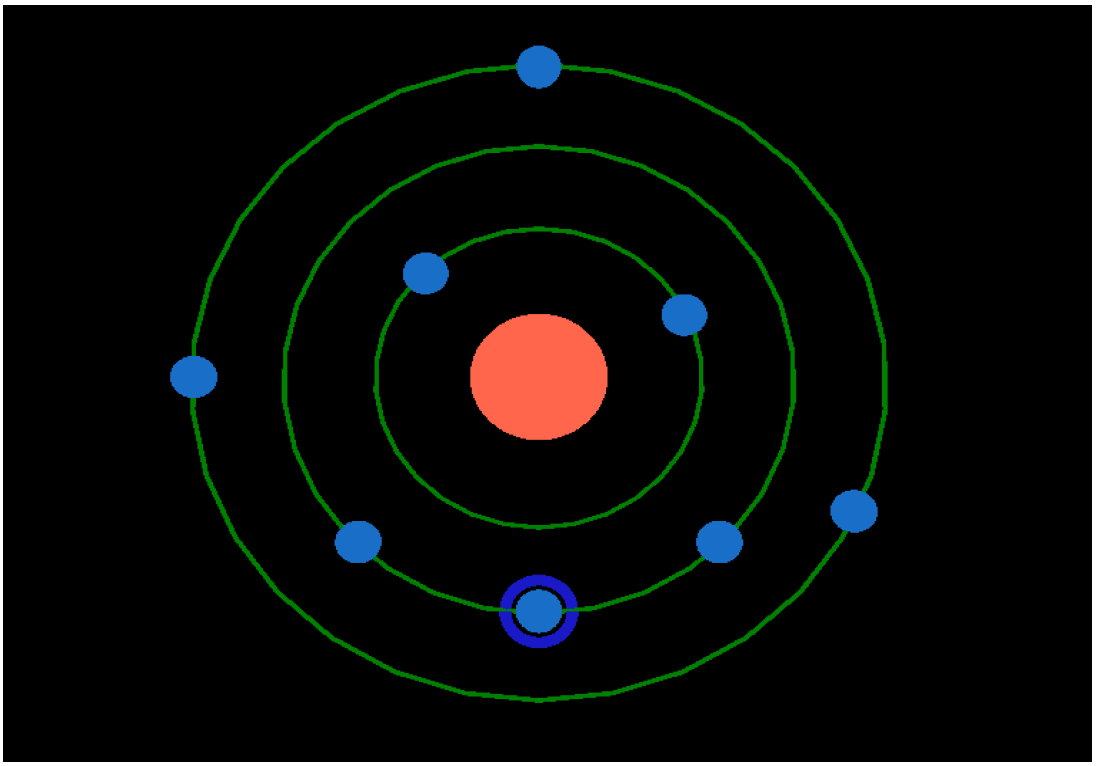


Figura 1: Sistema Solar

Capítulo 1

Introdução

Tal como já referido anteriormente, este relatório refere-se à terceira fase do nosso projeto de Computação Gráfica.

Neste momento, o nosso Sistema Solar irá conter o Sol, oito planetas, uma Lua e por último um cometa que terá a forma de um teapot.

As alterações que serão aplicadas nesta nossa fase relativamente ao gerador tratam-se de criar uma aplicação de modo a que seja feita a criação de listas de triângulos correspondentes às curvas de Bezier.

Já no caso do motor, iremos ter que criar curvas através de Catmull Rom, sendo que elas irão definir uma translação, recebendo o tempo que for necessário para percorrer essas mesmas curvas e os pontos de controlo que as irão definir, também a rotação irá sofrer esta mesma alteração, recebendo o tempo necessário para um planeta fazer uma rotação de 360º sobre si mesmo.

Por último, será necessário que os desenhos das primitivas passem a ser feitos através de VBO's.

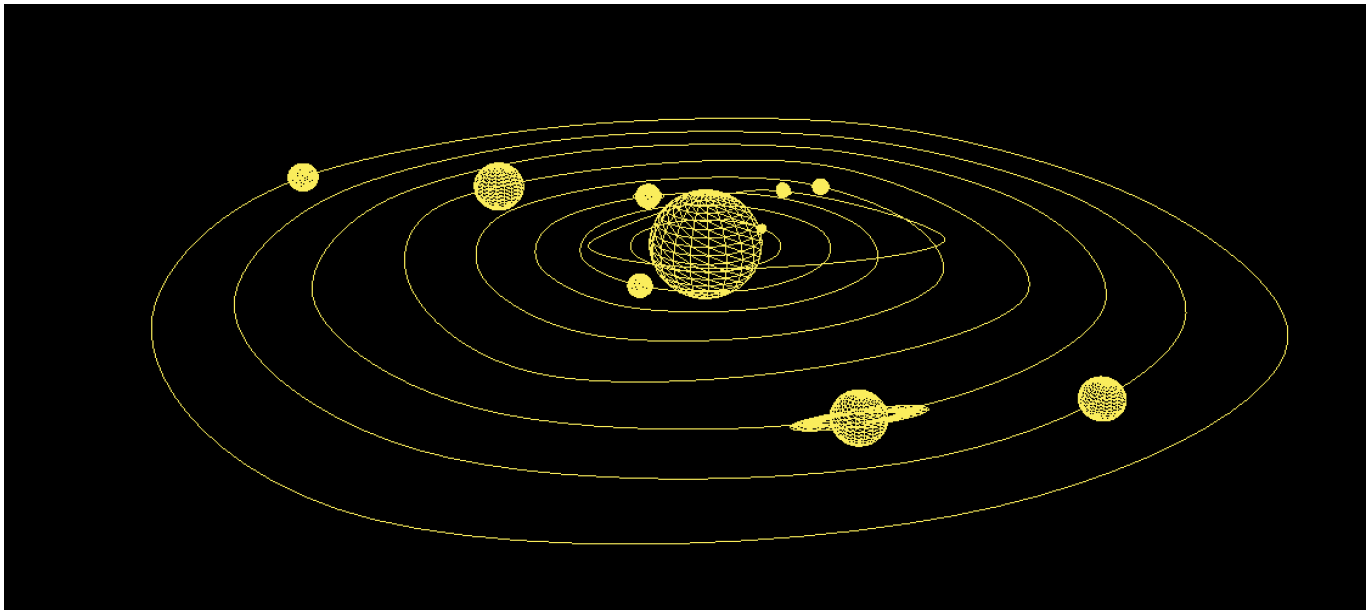


Figura 1.1: Sistema Solar

Capítulo 2

Gerador

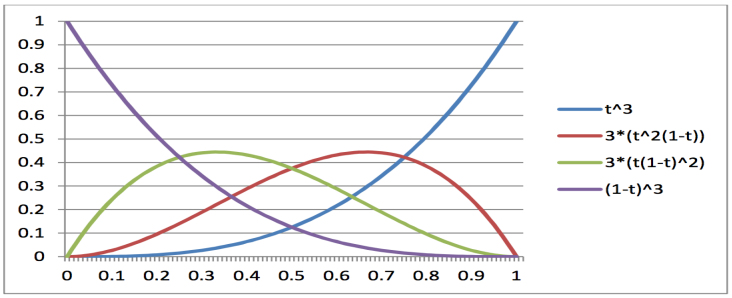
2.1 Curvas e superfícies de Bezier

Uma curva de Bezier é constituída por vários pontos, referidos como pontos de controlo, sendo que apenas o primeiro e o último deste conjunto pertencerão à curva, e os restantes servirão apenas como pontos de apoio para a definição da mesma, no nosso caso como se tratam apenas de curvas cúbicas estes restantes serão 2 pontos extra.

A partir desses nossos pontos de controlo poderemos construir a nossa curva de Bézier, que estará sempre contida dentro de um trapézio, que tem o nome de Polígono de Bézier.

Relativamente à construção da dita curva, será necessário recorrermos ao polinómio de Bernstein, visto que este polinómio é o único que nos permite obedecer a 2 fatores essenciais, sendo o primeiro, garantir que a nossa curva não sai fora da "caixa", ou seja, que a soma do pesos seja igual a 1, e por último,a nossa função terá de ser necessariamente cúbica.

A partir deste polinómio estamos aptos para conseguir desenhar a nossa curva de Bézier, que irá ser definida pela seguinte equação, onde $B_{i,n}$ corresponde ao polinómio de Bernstein, com n=3 pois queremos um polinómio cúbico. Ora, no seguinte gráfico podemos observar o comportamento para $i \in \{0, 1, 2, 3\}$, pois teremos 4 pontos de controlo:



$$B_{3,3}(t) = t^3$$
$$B_{2,3}(t) = 3t^2(1-t)$$
$$B_{1,3}(t) = 3t(1-t)^2$$
$$B_{0,3}(t) = (1-t)^3$$

Bernstein polynomials

$$B_{i,3} = \binom{3}{i} t^i (1-t)^{3-i}$$

Figura 2.1: Polinómio de Bernstein

Na seguinte figura temos um exemplo de uma curva de Bézier, onde irão haver 7 pontos de controlo, sendo desta maneira geradas duas curvas, uma de P0 até P3, e a outra de P3 até P6:

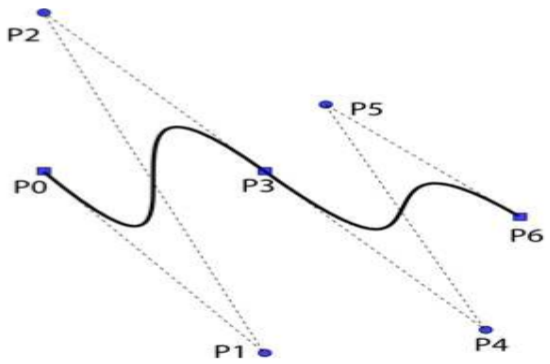


Figura 2.2: Equação da curva de Bézier

As superfícies de Bézier irão ser desenhadas à custa das curvas de Bézier, onde serão dados 16 pontos de controlo. Tal como demonstrado na seguinte imagem, podemos ver que estas superfícies irão ser formadas por várias curvas de Bezier, sendo cada uma definida por 4 pontos, formando assim uma espécie de "malha".

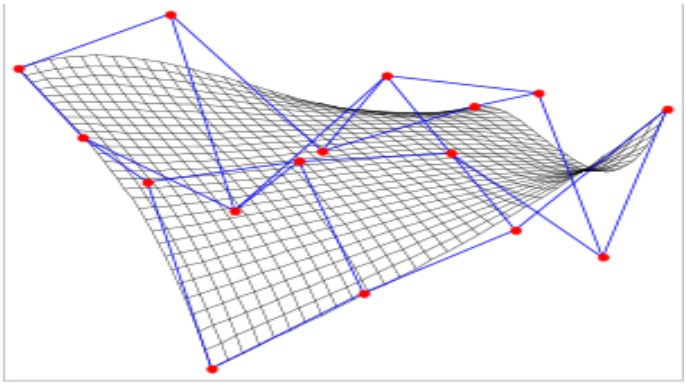


Figura 2.3: Exemplo superfície de Bézier

A superfície de Bezier irá ser definida pela seguinte equação:

$$B(u,v) = \sum_{j=0}^{3} \sum_{i=0}^{3} B_i(u) P_{ij} B_j(v)$$

Figura 2.4: Fórmula superfície de Bézier

2.1.1 Função makeMatrix

Realizamos a inserção dos pontos de controlo numa matriz 4*4, sendo que cada ponto é constituído por 3 coordenadas ficamos com o resultado final abaixo descrito

```
float mp[4][4][3];
for (i = 0; i < 4; i++) {
    for (j = 0; j < 4; j++, pos+=3) {
        mp[i][j][0] = p[pos]; // x
        mp[i][j][1] = p[pos+1]; // y
        mp[i][j][2] = p[pos+2]; // z
    }
}
```

Iniciamos pela criação das matrizes que estão descritas no formulário e que vamos necessitar:

$$\text{Let } U = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \text{ and } M = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Figura 2.5: criação da Matriz M e U

```
float M[4][4] = { { -1, 3, -3, 1 },{ 3, -6, 3, 0 },{ -3, 3, 0, 0 },{ 1, 0, 0, 0 } };

float U[1][4] = { {(float)pow(u, 3), (float) pow(u, 2), u,1 } };
```

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

Figura 2.6: criação da Matriz V

```
float V[4][1] = {{(float) pow(v, 3)}, {(float) pow(v, 2)}, {v }, {1 } };
```

Após a criação fazemos então a utilização destas matrizes para cálculos, tais como a criação da matiz auxiliar UM que resulta da multiplicação da matriz U pela M

```
float UM[1][4];

...
for (i = 0; i < 4; i++) {
    UM[0][i] = (U[0][0] *M[0][i])
             + (U[0][1] *M[1][i])
             + (U[0][2] *M[2][i])
             + (U[0][3] *M[3][i]);
}
```

Posteriormente criamos ainda outra matriz que corresponde ao calculo da multiplicação entre a Matriz UM pelos pontos que nos foram fornecidos pelo professor e devidamente armazenados na matriz mp

$$UM \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix}$$

Figura 2.7: criação da Matriz UMP

```
float UMP[1][4][3];
for (j = 0; j < 3; j++) {
    for (i = 0; i < 4; i++) {
        UMP[0][i][j] = (UM[0][0] * mp[0][i][j])
                     + (UM[0][1] * mp[1][i][j])
                     + (UM[0][2] * mp[2][i][j])
                     + (UM[0][3] * mp[3][i][j]);
    }
}
```

$$(VM)^T \quad \text{OU} \quad M^T \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix}$$

Figura 2.8: criação da Matriz MV

Criação da matiz auxiliar MV que resulta da multiplicação da matriz M pela V

```
for (i = 0; i < 4; i++) {
    MV[i][0] = (M[i][0] * V[0][0])
             + (M[i][1] * V[1][0])
             + (M[i][2] * V[2][0])
             + (M[i][3] * V[3][0]);
}
```

E por fim calculamos o ponto final que pretendemos

$$B(u,v) = UM \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} (VM)^T$$

Figura 2.9: criação da Matriz UMPMV

```
float UMPMV[3];
for (i = 0; i < 3; i++) {
    UMPMV[i] = (UMP[0][0][i] * MV[0][0])
               + (UMP[0][1][i] * MV[1][0])
               + (UMP[0][2][i] * MV[2][0])
               + (UMP[0][3][i] * MV[3][0]);
}
```

2.1.2 Teapot

Nesta secção iremos explicar o método que utilizamos para a leitura do ficheiro *.patch*, para que os dados fiquem da forma correta para a função explicada acima conseguir gerar os pontos de desenho do **teapot**. Todo o processo de leitura do ficheiro é feito com a função **readPatch**, que recebe como argumentos o nome do ficheiro *.patch* e número de slices e stacks que queremos que o nosso teapot tenha. Após abrirmos o ficheiro (com sucesso), ao qual chamamos **myfile**, teremos que passar pelas seguintes fases, até o parsing do ficheiro estar completo:

1. Ler o número de patches que vamos ter;
2. Por cada patch adicionamos a um *map*, o par [indice do patch, vetor com os indices dos pontos do patch];
3. Ler o número de pontos de controlo que vamos ter;
4. Guardamos os pontos de controlo todos num *vector* <*float*>;
5. Percorremos o *map* anteriormente criado e criamos um novo *map*, que para cada chave do *map* antigo, terá a si associado um *vector* com os pontos do patch. Este ultimo *map* está definido como variável global;

Passo 1:
Para lermos o número de patches que vamos ter, basta:

```
    getline(myfile,line);
    n_patches = stoi(line);
```

Passo 2:
Nesta fase, iremos ter tantas iterações do ciclo como o número de patches que temos no ficheiro. Vimo-nos também na necessidade de definir a função **splitString**, que recebe como argumentos a linha lida e uma flag (0 ou 1) que serve para sabermos se queremos guardar os valores lidos como *int* ou como *float*, e devolve um *vector* com os pontos lidos no ficheiro. Resumidamente, esta função serve para guardarmos separadamente cada valor lido, usando como delimitador o caracter ','. Assim, para cada patch no ficheiro **teapot.patch**, iremos chamar a função **splitString** passando como argumento a linha lida e a flag 0, pois queremos guardar os índices dos pontos como inteiros, e guardamos esses índices num *vector*, para depois adicionarmos o par [indice, *vector* com os indices] ao *map* **pontos_Aux**. No final de cada iteração do ciclo é importante apagarmos o conteudo do **vector** onde guardamos os indices, para não levarmos valores indesejados de uma iteração para a outra. Assim, o seguinte ciclo é a parte da nossa função de parsing responsável por este passo:

```
while(i<n_patches){
    getline(myfile,line); // lemos a proxima linha que ainda nao foi lida
    indices = splitString(line,0);
    pontos_Aux.insert(pair<int,vector<float>>(i,indices));
    i++;
    indices.clear();
}
```

Passo 3:
A estratégia para sabermos o número de pontos que vamos ler, é igual à estratégia usada no **passo 1**, como podemos observar:

```
    getline(myfile,line);
    int n_points = stoi(line);
```

Passo 4:
Após sabermos o número de pontos de controlo que vamos ler, basta aplicarmos o mesmo raciocínio usado no **passo 2**, isto é, temos tantas iterações quanto o número de pontos de controlo e por cada iteração, aplicamos a função **splitString** recebendo a linha lida e a flag 1 (desta vez já queremos guardar os pontos como *float*) como argumentos. Deste modo, vamos guardar cada ponto num *vector* de *float* e, mais tarde, para lermos cada ponto basta irmos ao índice que guardarmos no *vector* dos índices, e ler os primeiros ter elementos a partir do índice lido. Ao *vector* onde guardamos estes pontos chamamos **control_points**, como podemos verificar:

```
i=0;
vector <float> aux;
vector <float> control_points;
while(i<n_points){
    getline(myfile,line);
    aux = splitString(line,1);
    for(int j=0; j<aux.size(); j++)
        control_points.push_back(aux[j]);
    i++;
}
```

Passo 5:
Neste momento já temos armazenados os seguintes dados:

- Cada patch com os os indices dos seus pontos a si associados, no *map* **pontos_Aux**;
- Todos os pontos de controlo, guaradados no textitvector **control_points**;

Assim, com as duas estruturas mencionadas acima, já conseguimos preencher o *map* **pontos**, definido como variavel global, com os pontos de controlo de cada patch. Assim, basta iterarmos sobre o *map* **pontos_Aux**, e para cada patch, inserimos em **pontos**, o par [indice do patch, pontos de controlo do patch]. Para isso, basta irmos buscar cada um dos pontos de controlo associados a cada índice que está no *vector* de indices do *map* **pontos_Aux**. Em suma, preenchemos a nosso estrutura final da seguinte forma:

```
 for( auto map_iter = pontos_Aux.cbegin() ; map_iter != pontos_Aux.cend() ; ++map_iter ) {
          for(int vec_ind=0; vec_ind < map_iter->second.size() ; vec_ind++) {
              int x = map_iter->second[vec_ind]*3;
              aux.push_back(control_points[x]);
              aux.push_back(control_points[x+1]);
              aux.push_back(control_points[x+2]);
          }
          pontos.insert(pair<int,vector<float> >(i,aux));
          aux.clear();
          i++;
}
```

Deste modo, lêmos todo o ficheiro **teapot.patch**, e preparamos os dados para que o teapot possa ser desenhado, portanto, neste momento, basta apenas chamar a função **drawTeapot**, passando como argumento as slices e stacks que a função **readPatch** recebeu.

Capítulo 3

Motor

Neste capítulo irão ser explicadas as novas funcionalidades do nosso *Motor*, que passará desde as estruturas que fomos utilizando até ao desenho do modelo em si.

3.1 Vertex Buffer Object

Um VBO (Vertex Buffer Object) representa uma funcionalidade do OpenGL que nos irá fornecer certos métodos de modo a que possamos carregar os vértices (por exemplo, a posição, vetor normal, etc.) para o nosso dispositivo de vídeo, de modo a fazer um rendering de modo não imediato.

Ao usarmos os VBO's vamos estar a obter certas vantagens, como um desempenho mais elevado ao fazer o rendering de modo imediato, principalmente porque os dados irão estar a ser armazenados na placa gráfica, podendo ser o vídeo processado diretamente.

3.1.1 Criação e aplicação de VBO's

De modo a conseguirmos criar o nosso VBO para as primitivas, começámos por fazer uma alteração na nossa class Model, havendo necessidade de colocar um identificador para o nosso respetivo VBO.

```
class Model{
public:
    string nome; //nome do modelo desenhado
    GLuint buffer; // para dizermos qual e o nosso vbo
    vector <float> pontos; //vector que armazena os pontos
};
```

Após esta alteração, procedemos à criação de uma nova função, designada por groupToVBO, que irá receber um Group g.

Esta nova função por nós criada serve para conseguirmos armazenar num array todas as coordenadas dos pontos que estão na nossa mesma class Group, num vector chamado pontos.

Ao fazer esta armazenação vai proceder à criação do VBO, onde irá ser guardado o seu identificador no devido identificador que foi declarado na nossa class Model.

Visto que no caso do planeta Terra e de Saturno possuirmos um sub-grupo em cada, adicionámos também uma nova funcionalidade, em que irá entrar num ciclo diferente caso isto aconteça.

Com isto, já é possível o processamento dos vértices, estando pronto para o próximo passo.

De modo a conseguirmos proceder á inicialização desta mesma função, foi necessária a criação de outra designada por initVBO, que irá receber uma Scene s:

```
void initVBO(Scene* s){
    size_t i;
```

```
    for(i = 0;i<s->grupos.size();i++){ /// por cada grupo da Scene, vamos por os seus pontos no VBO, com a
        funcao groupToVBO
       groupToVBO(s->grupos[i]);
    }
}
```

3.2 Rotação e Translação

Tal como referido na Introdução, tanto na nossa Rotação, como na nossa Translação irão ser feitas diversas alterações.

No caso da translação, ela já não irá receber as coordenadas dos pontos para proceder à sua movimentação, mas irá sim receber um certo tempo, e um certo número de pontos. Esse tempo vai estar representado em segundos, e vai corresponder ao tempo que cada objeto vai demorar até passar pela curva inteira. Os pontos, serão apenas para controlo, podendo assim desenhar essa mesma curva e permitir que o devido objeto se possa mover.

Já relativamente à rotação, ela também irá receber um certo tempo, que tal como na translação, estará representado em segundos, representando o tempo o nosso objeto irá demorar até fazer uma rotação completa (360^o) sobre si mesmo.

Para isto, foi necessário que fossem feitas certas alterações na nossa classe Group, nomeadamente, procedemos à introdução de duas variáveis relativas ao tempo, uma da rotação e outra da translação; outra variável que irá servir como um contador para esse tempo (da translação). Por último, também tivémos que introduzir um vector chamado Pontos, onde irão ser armazenados os pontos de controlo da curva.

```
class Point{
public:
   float pontos[3];
};
```

```
class Group{
public:
    int timeR ,timeT;
    float before;
    float s; //contador de segundos
    int n;  //numero de pontos
    map<int,vector<float> > transf;
    vector<Point*> pontos;
    vector <Model*> mod;
    vector <Group*> grup;
};
```

3.2.1 Método Catmull Rom

É a partir deste método que vamos conseguir desenhar as nossas curvas, a partir dos pontos de controle, previamente armazenados.

A única maneira de conseguirmos fazer isto, é com 4 pontos, criando assim um segmento da curva.

```
 void getCatmullRomPoint(float t, int *indices, float *res,vector<Point*> p1) {
    float res_aux[4];
    int i;
    // catmull-rom matrix
    float m[4][4] = { {-0.5f, 1.5f, -1.5f, 0.5f},
```

```
                    { 1.0f, -2.5f, 2.0f, -0.5f},
                    {-0.5f, 0.0f, 0.5f, 0.0f},
                    { 0.0f, 1.0f, 0.0f, 0.0f}};

    for (i=0; i<4; i++)
       res_aux[i]= pow(t,3) * m[0][i] + pow(t,2) * m[1][i] + t * m[2][i] + m[3][i];

    for(i=0;i<3;i++){
        res[i]=res_aux[0]*p1[indices[0]]->pontos[i] + res_aux[1]*p1[indices[1]]->pontos[i]
            +res_aux[2]*p1[indices[2]]->pontos[i] + res_aux[3]*p1[indices[3]]->pontos[i];
}
}
```

Tal como aprendido, iremos precisar de 4 pontos de controle para podermos desenhar o nosso segmento de curva, para procedermos à explicação de cada uma das nossas funções anteriores, irá ser necessário (apenas para a facilitar) que apresentemos um exemplo.

Nesta seguinte figura, podemos observar que através de 4 pontos de controlo (PO, P1, P2 e P3) e através de uma variável t, que varia entre 0.0 e 1.0, sendo que t representa a porção da distância entre os dois pontos de controlo mais próximos (P1 e P2).

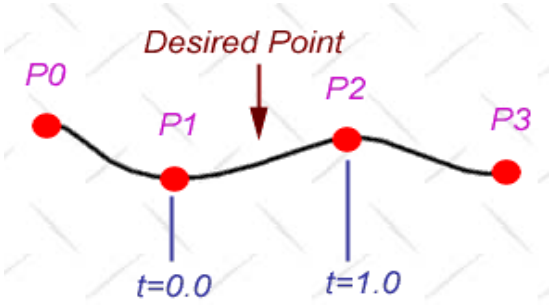


Figura 3.1: Curva Catmull-Rom

Relativamente à nossa função getGlobalCatmullRomPoint, esta irá receber um Group g, que é onde se irá buscar os pontos de controle previamente falados no nosso relatório, calculando os índices do nosso array, que irão corresponder aos pontos de controle a usar para calcular o respetivo segmento da curva.

Já a nossa primeira função, getCatmullRomPoint irá receber os índices, que irão corresponder aos 4 pontos de controle, calculando assim a devida coordenada no segmento da curva.

Por último, a nossa função getCatmullRomCurve tem como objetivo principal o desenho de cada um dos segmentos da curva para cada diferente t, através das coordenadas que foram obtidas no nossa função anterior getGlobalCatmullRomPoint.

```
// given global t, returns the point in the curve
void getGlobalCatmullRomPoint(Group* g,float gt, float *res) {

    float t = gt * g->n; // this is the real global t
    int index = floor(t); // which segment
    t = t - index; // where within the segment

    // indices store the points
    int indices[4];
    indices[0] = (index + g->n-1)%g->n; indices[1] = (indices[0]+1)%g->n;
    indices[2] = (indices[1]+1)%g->n; indices[3] = (indices[2]+1)%g->n;

    getCatmullRomPoint(t, indices, res,g->pontos);
}

void renderCatmullRomCurve(Group* g) {
    float gtt;
    float res[3];
    // desenhar a curva usando segmentos de reta - GL_LINE_LOOP
```

```
    glBegin(GL_LINE_LOOP);
    for (gtt=0; gtt<1; gtt+=0.01){
        getGlobalCatmullRomPoint(g,gtt,res);
        glVertex3fv(res);
    }
    glEnd();
}
```

3.2.2 Tempo - Rotação/Translação

Tal como dito anteriormente, vamos introduzir uma variável tanto para a rotação (tempo que o objeto demora a rodar 360º), como para a translação (tempo que o objeto demora a percorrer a curva).

Para conseguirmos realizar este feito, foi necessário utilizarmos mais funcionalidades do OpenGL, que neste caso optámos por usar glutGet(GLUT_ELAPSED_TIME), que irá ser chamado na nossa renderScene, de modo a poder contabilizar estes tempos anteriormente referidos.

- Translação:

Foi criada uma variável para calcular o tempo, que designámos por currentTime, que irá ser inicializada a zero e incrementada com a passagem do tempo, será aqui que iremos utilizar a funcionalidade do OpenGL anteriormente falada (glutGet(GLUT_ELAPSED_TIME). Para podermos guardar o tempo anterior foi criada uma variável na nossa class Group chamada before.

A partir destas duas variáveis podemos então calcular o nosso tempo final, ou seja, caso before seja igual a zero, o seu valor será atualizado para o do currentTime, a partir daí calculamos a diferença entre estes dois tempos, sendo necessário também passar para milissegundos, daí o *1000, somando este valor final ao resultado do contador do tempo.

- Rotação:

Já no caso do rotação, irá ser necessário que, por cada grau rodado do objeto, o tempo vá aumentando 1 segundo, sendo que no final da rotação teremos o nosso tempo final.

3.3 Alterações na leitura do ficheiro.xml

De maneira a satisfazer os novos requerimentos pedidos nesta fase procedemos a uma ligeira alteração na leitura do ficheiro **XML**. A única alteração mais significativa foi feita na função readGroup() que iremos explicar de seguida.

3.3.1 Função readGroup()

Visto que agora, tanto numa translação como numa rotação,recebemos tempos decidimos acrescentar esses valor ao nosso map das transformações como exemplificado em baixo

```
}else if(tag == "Rotate")
       {

           atri = filho ->Attribute("time");
           if(atri != NULL) aux.push_back(atof(atri));
```

No caso de uma translação também foi necessário fazer a leitura dos pontos de controlo e armazena-los na nossa estrutura Point. Para tal procedemos a esta alteração

```
if(elemN == "point"){
                  Point* ponto1 = new Point();
```

```
            ponto1->pontos[0] = atof(elem3->Attribute("x"));
            ponto1->pontos[1] = atof(elem3->Attribute("y"));
            ponto1->pontos[2] = atof(elem3->Attribute("z"));
            g->pontos.push_back(ponto1);
            nPontos++;
        }
        g->n=nPontos;
    }
```

3.4 Alterações no desenho das figuras

Nesta parte do código simplesmente alterou-se a função drawFigures() de maneira a satisfazer as novas transformações e de maneira a satisfazer a implementação dos VBO's.

3.4.1 Função drawFigures()

Caso a nossa transformação seja uma Translação procedemos à execução do método o Catmull Rom para conseguir desenhar as nossas curvas, a partir dos pontos de controle, previamente armazenados.

```
    if(it->first == T ){
      timeT = aux[0];
         renderCatmullRomCurve(p);
         getGlobalCatmullRomPoint(p,(p->s),res);
         glTranslatef(res[0],res[1],res[2]);
     }
```

No caso da transformação ser uma Rotação vamos calcular o tempo que o objeto demora a rodar 360 sobre si mesmo. Irá ser necessário que, por cada grau rodado do objeto, o tempo vá aumentando 1 segundo, sendo que no final da rotação teremos o nosso tempo final.

```
  if(it->first == R){
         long int tempo = aux[0];
         glRotatef(360*(currentTime%tempo)/(tempo),aux[1],aux[2],aux[3]);
     }
```

Para a Escala o processo mantém-se chamando apenas a função do glut glScalef().

Caso o tempo anterior estiver a 0 atualizamos com o tempo corrente,caso contrario atualizamos a variável s de modo a demorar o tempo de translação que é o tempo que o objeto demora a percorrer a sua órbita.

```
    //se o tempo anterior estiver a 0 actualize com o tempo currente
    if(p->before==0)
        p->before = currentTime;
    else{//actualiza a variavel s de modo que demora o tempo de translacao
     timeT = aux[0];
        p->s += ((currentTime-(p->before))/(timeT*1000));
        p->before = currentTime;
    }
```

Capítulo 4

Sistema Solar

Depois de efetuadas todas as nossas transformações, o resultado final será um Sistema Solar dinâmico em que cada planeta tem a sua própria órbita (ou seja gira em torno do Sol) e gira em volta de si mesmo. No caso da lua também se acrescentou uma órbita em torno da Terra. Como pedido no enunciado também é possível verificar que o cometa(teapot) está em órbita.

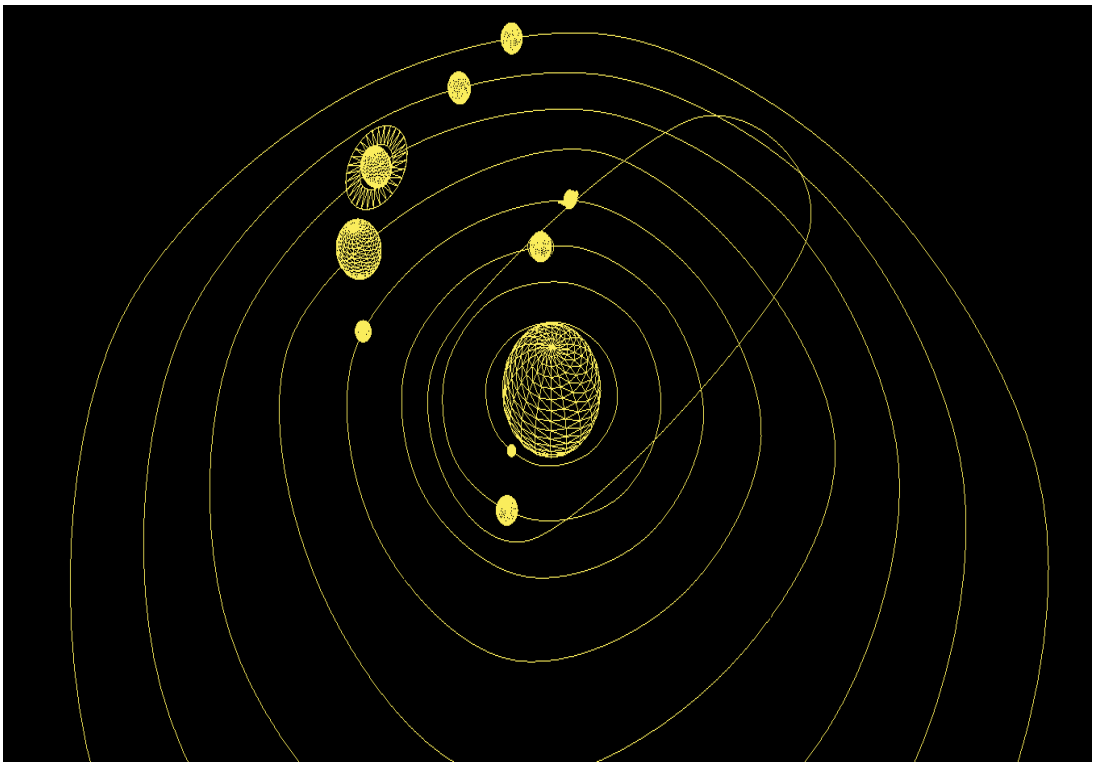


Figura 4.1: Sistema Solar

Capítulo 5

Conclusão

A realização deste projeto tem como objetivo a criação de um sistema solar impresso de movimento que, no nosso caso, é constituído por 11 corpos celestes (o Sol,8 planetas, 1 lua e 1 cometa). Consideramos que esta fase se dividiu essencialmente em quatro etapas chave:

- Leitura dos novos parâmetros do XML para a nossa estrutura de dados;
- Utilização de VBO's;
- Patches de Bezier, parsing e passagem para ficheiro .3d;
- Método de Catmull Rom para o desenho de órbitas.

A primeira, dificuldade encontrada, foi a maneira como iríamos adaptar das nossas estruturas de dados de maneira a satisfazer os novos parâmetros do ficheiro XML. Depois de ultrapassada esta dificuldade o desenho das orbitas foi feito com relativa facilidade. A segunda dificuldade encontrada foi na ativação e utilização dos VBO's pois a sua utilização ainda era muito nova para nós pelo que demorou um bocado até perceber seu objetivo e a melhor maneira de implementar. Também tivemos algumas dificuldades no patch de Bézier, principalmente na maneira como liamos o respetivo patch fornecido pelo professor, de modo a conseguirmos ter as estruturas adequadas para guardarmos os dados fornecidos no ficheiro *.patch*.

Todavia sendo estas as questões principais do nosso projeto, houve alguns detalhes que não conseguimos aprimorar como pretendíamos, um desses casos é a orientação do teapot , correspondente ao cometa, no qual a orientação do bico não acompanha a direção da tangente à curva como era suposto. No Sistema Solar a Lua, apesar de apresentar um movimento de rotação sobre si própria, encontra-se estacionaria e não a orbitar o planeta Terra como era suposto.

Resumidamente, consideramos a realização desta fase foi satisfatória visto que cumprimos com a maioria dos requisitos pedidos, porém não consideramos o trabalho concluindo havendo sempre espaço para pequenas melhorias, principalmente nas falhas mencionadas anteriormente.

Bibliografia

[1] Documentação fornecida pelo professor

[2] http://www.grinninglizard.com/tinyxmldocs/annotated.html

[3] http://www.grinninglizard.com/tinyxml2/

[4] https://www.mvps.org/directx/articles/catmull/

[5] http://www.cplusplus.com

[6] https://en.wikipedia.org/wiki/VertexBufferObject

[7] http://joshworth.com/dev/pixelspace/pixelspace_solarsystem.html

[8] http://www.songho.ca/opengl/glvbo.html

[9] https://web.cs.wpi.edu/ matt/courses/cs563/talks/surface/bez_surf.html