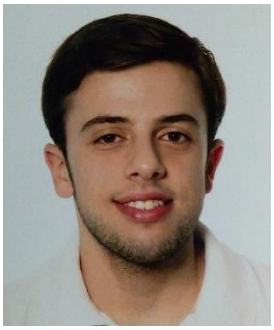
Universidade do Minho

Departamento de Informática

Curves, Cubic Surfaces and VBOs

Mestrado Integrado em Engenharia Informática

Computação gráfica



Jaime Leite (A80757)

Bruno Veloso (A78352)

Índice

Índice de figuras

Introdução

Partindo do enunciado que nos foi proposto, apresentaremos neste relatório o trabalho desenvolvido na terceira fase do projeto relativo à unidade curricular de *Computação Gráfica*.

Inicialmente, abordaremos a forma como alteramos as estruturas de dados e o parse, com o objetivo de ser possível manipular a informação para responder aos objetivos desta fase do trabalho prático.

Seguidamente, explicaremos a implementação dos VBOs, bem como o desenho do teapot, tendo por base um conjunto de patches de Bezier, e a aplicação das curvas de Catmull-Rom no desenho das órbitas do teapot, de cada planeta e da lua da Terra.

Numa fase final, apresentaremos o ficheiro XML, que serviu de teste para o desenho do sistema solar. Este ficheiro contém, para além da indicação das figuras a desenhar, os tempos de translação e de rotação das figuras, bem como os pontos de controlo para o desenho das suas órbitas.

Fase 3

Curvas, superfícies curvas e VBOs

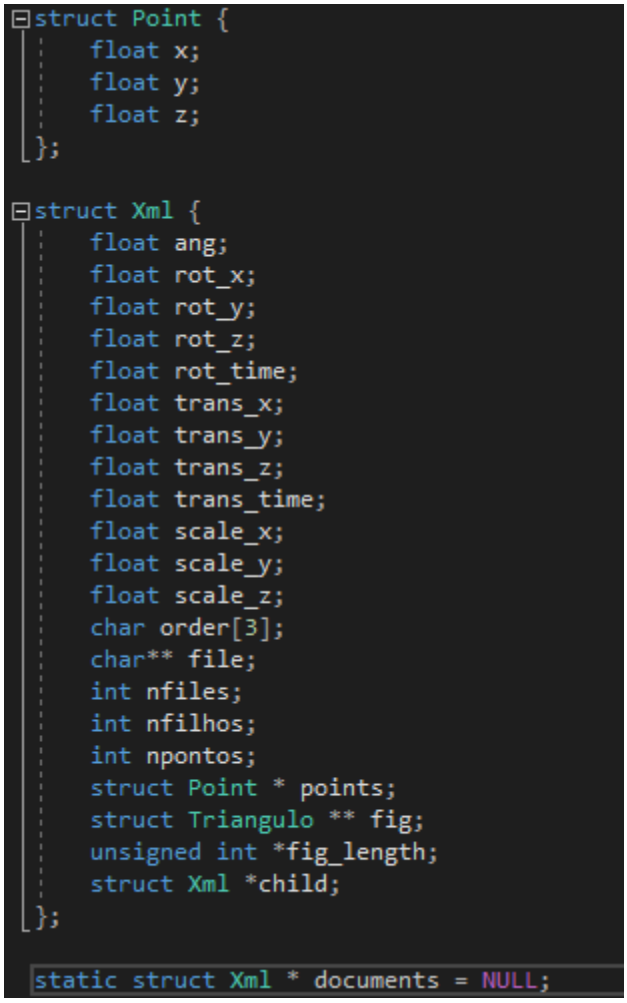
Apresentaremos de seguida o desenvolvimento dos requisitos propostos relativos à terceira fase do projeto da unidade curricular de *Computação Gráfica*.

1. Modificação das estruturas

Tendo em conta as exigências desta terceira fase do projeto e para responder às mesmas, foi necessário efetuar alterações nas nossas estruturas de armazenamento de dados.

As novas exigências desta fase, no que toca a guarda informação, são: o XML pode conter um parâmetro "time" na tag "rotate", o qual representa o tempo de rotação da própria figura; a tag "translate" pode igualmente conter um parâmetro "time", que representará o tempo de translação (à volta do Sol) da figura. A partir do momento em que a tag “translate” tem o parâmetro "time", serão então fornecidos dentro da mesma pontos de controlo que servirão de apoio para a construção da sua órbitra. No caso de o "translate" não ter parâmetro "time", recebe os parâmetros como na fase anterior.

Por forma a guardar o tempo de rotação, foi colocado na "struct Xml" um valor "float rot_time"; de maneira a guardar o tempo de translação foi posto um valor "float trans_time"; com o intuito de guardar os pontos da translação, foi armazenado na "struct Xml" um array de "struct Point" chamado "points", em que "struct Point" tem um valor "float x", "float y" e "float z", que representam as coordenadas de um ponto. Por fim, para saber o número de pontos para a translação foi guardado um valor "int npontos", que representa o número de pontos para o desenho da curva Catmull-Rom.



```
struct Point {
    float x;
    float y;
    float z;
};

struct Xml {
    float ang;
    float rot_x;
    float rot_y;
    float rot_z;
    float rot_time;
    float trans_x;
    float trans_y;
    float trans_z;
    float trans_time;
    float scale_x;
    float scale_y;
    float scale_z;
    char order[3];
    char** file;
    int nfiles;
    int nfilhos;
    int npontos;
    struct Point * points;
    struct Triangulo ** fig;
    unsigned int *fig_length;
    struct Xml *child;
};

static struct Xml * documents = NULL;
```

Figura 1: structs para armazenamento de dados

2. Modificação do parse

Nesta fase foi utilizado o parse anteriormente desenvolvido, mas com umas pequenas alterações. Agora para guardar a informação de uma tag "translate", primeiro verificamos se esta tem o atributo "time". Caso não tenha, consideramos que é um "translate" como a fase anterior. Caso tenha o atributo "time", guardamos o seu tempo, contamos o número de pontos de controlo que vai ter, para alocar memória tendo em conta o número de pontos, e de seguida guardamos ponto a ponto.

```
translate = group->FirstChildElement("translate");
if (translate) {
    name = translate->Attribute("time");
    if (name == NULL) {
        name = translate->Attribute("X");
        if (name) dcc[z].trans_x = atof(name);
        name = translate->Attribute("Y");
        if (name) dcc[z].trans_y = atof(name);
        name = translate->Attribute("Z");
        if (name) dcc[z].trans_z = atof(name);
    }
    else {
        dcc[z].trans_time = atof(name);
        npoints = pointsT(file, translate);
        dcc[z].points = (struct Point*) calloc(npoints + 1, sizeof(struct Point));
        l = 0;
        dcc[z].npontos = npoints;
        po = translate->FirstChildElement("point");
        while (po) {
            name = po->Attribute("X");
            if (name) dcc[z].points[l].x = atof(name);
            name = po->Attribute("Y");
            if (name) dcc[z].points[l].y = atof(name);
            name = po->Attribute("Z");
            if (name) dcc[z].points[l].z = atof(name);
            po = po->NextSiblingElement("point");
            l++;
        }

    }
```

Figura 2: parse do translate

Para guardar a informação do "rotate", apenas foi introduzida a hipótese de este ter "time" ou não, sendo que tanto deve aceitar um "angle" como um "time".

```
rotate = group->FirstChildElement("rotate");
if (rotate) {
    name = rotate->Attribute("time");
    if (name) dcc[z].rot_time = atof(name);
    name = rotate->Attribute("angle");
    if (name) dcc[z].ang = atof(name);
    name = rotate->Attribute("axisX");
    if (name) dcc[z].rot_x = atof(name);
    name = rotate->Attribute("axisY");
    if (name) dcc[z].rot_y = atof(name);
    name = rotate->Attribute("axisZ");
    if (name) dcc[z].rot_z = atof(name);
}
```

Figura 3: parse do rotate

3. VBOs

Para melhorar o desempenho da aplicação, todos os modelos passaram a ser desenhados com VBOs.

Para isso foram criados três "buffers", onde serão guardadas as coordenadas de cada ponto a desenhar. Um dos buffers representa a esfera, outro o teapot e outro o disco de Saturno. Cada um destes buffers, no fim de "preenchidos", conterá a informação para a renderização de um modelo. No momento de desenhar, usamos as funções “glVertexPointer” e “glDrawArrays”, como alternativa ao desenho de triângulo a triângulo, como as fases anteriores.

Com a implementação dos VBOs verificou-se um aumento de desempenho de 6 a 7 vezes (nos computadores dos programadores do projeto), comparando com a implementação sem VBOs. Este desempenho deve-se ao facto de, com VBOs, a renderização dos modelos ser feita através da placa gráfica do computador, em vez de estar a guardar na memória do sistema toda esta informação para desenhar, tornando a renderização dos modelos muito mais eficiente.

```
if (!strcmp(file, "..\\..\\..\\Fig_files\\sphere.txt")) {
    glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);
    glVertexPointer(3, GL_FLOAT, 0, 0);
    glDrawArrays(GL_TRIANGLES, 0, size * 3);
}
if (!strcmp(file, "..\\..\\..\\Fig_files\\disk.txt")) {
    glBindBuffer(GL_ARRAY_BUFFER, buffers[1]);
    glVertexPointer(3, GL_FLOAT, 0, 0);
    glDrawArrays(GL_TRIANGLES, 0, size * 3);
}
if (!strcmp(file, "..\\..\\..\\Fig_files\\teapot.txt")) {
    glBindBuffer(GL_ARRAY_BUFFER, buffers[2]);
    glVertexPointer(3, GL_FLOAT, 0, 0);
    glDrawArrays(GL_TRIANGLES, 0, size * 3);
}
```

Figura 4: exemplificação de como desenhar um VBO

```
void fillBuffers(struct Xml * documents) {
    int sphere = 0;
    int disk = 0;
    int teapot = 0;
    int k = 0;
    int n = 0;

    for (int i = 0, j = 0; documents[i].nfiles > 0; i++) {

        j = 0;

        while (j < documents[i].nfiles) {

            if (!strcmp((documents[i].file[j]), "..\\..\\..\\Fig_files\\sphere.txt") && sphere == 0) {
                float *arraySphere = (float *)malloc(documents[i].fig_length[j] * 9 * sizeof(float));
                for (k = 0, n = 0; k < documents[i].fig_length[j]; k++) {
                    arraySphere[n++] = documents[i].fig[j][k].x1;
                    arraySphere[n++] = documents[i].fig[j][k].y1;
                    arraySphere[n++] = documents[i].fig[j][k].z1;
                    arraySphere[n++] = documents[i].fig[j][k].x2;
                    arraySphere[n++] = documents[i].fig[j][k].y2;
                    arraySphere[n++] = documents[i].fig[j][k].z2;
                    arraySphere[n++] = documents[i].fig[j][k].x3;
                    arraySphere[n++] = documents[i].fig[j][k].y3;
                    arraySphere[n++] = documents[i].fig[j][k].z3;
                }
                sphere = 1;
                glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);
                glBufferData(GL_ARRAY_BUFFER, n * sizeof(float), arraySphere, GL_STATIC_DRAW);
            }
            if (!strcmp((documents[i].file[j]), "..\\..\\..\\Fig_files\\disk.txt") && disk == 0) {
```

Figura 5: parse dos VBOs

4. Aplicação dos patches de Bezier no desenho do Teapot

No desenho do teapot, aplicando os patches de Bezier, partiu-se de um ficheiro previamente fornecido pelo docente da unidade curricular de *Computação Gráfica.* Este ficheiro contém um conjunto de patches e um conjunto de pontos de controlo. No primeiro conjunto, cada patch é um aglomerado de índices que representam a posição dos pontos no segundo conjunto. Tendo por base esta informação, na função "bezierPatch" guardaram-se em estruturas globais os patches e os pontos contidos no ficheiro.

```
ponto *cpoints;      // control points
int **indexes;       // indexes of the points for each patch
int patches;         // number of patches
int ncpoints;        // number of control points
```

Figura 6: armazenamento dos pontos de controlo e dos patches

Na função "geradorBezier", são escritos num ficheiro auxiliar os pontos necessários para o programa "engine" desenhar o teapot. Para cada patch contido na estrutura global, e tendo em consideração o nível de tesselation requerido na chamada da função, fizeram-se invocações à função "bezierPoint", por forma a serem obtidos os pontos do teapot de forma correta.

```
int geradorBezier(char *outfile, int tesselation) {
    int linhas = 0;
    ponto pv[16];
    int divs = tesselation; // change this to change the tesselation level
    ostringstream pontos;
    ofstream out;

    out.open(outfile);
    if (!out.is_open()) {
        perror("ofstream.open");
        return 1;
    }
    for (int i = 0; i < patches; i++) {
        for (int j = 0; j < 16; j++) {
            pv[j] = cpoints[indexes[i][j]];
        }
        for (int u = 0; u < divs; u++) {
            float p1[3];
            float p2[3];
            float p3[3];
            float p4[3];
            for (int v = 0; v < divs; v++) {
                bezierPoint(u / (float)divs, v / (float)divs, pv, p1);
                bezierPoint((u + 1) / (float)divs, v / (float)divs, pv, p2);
                bezierPoint(u / (float)divs, (v + 1) / (float)divs, pv, p3);
                bezierPoint((u + 1) / (float)divs, (v + 1) / (float)divs, pv, p4);
```

Percorrer cada patch armazenado

Obter os pontos relativos a cada patch

Obter quatro pontos com ajuda das operações sobre matrizes

Figura 7: função que armazena num ficheiro os pontos do teapot

Cada linha desse ficheiro contém nove valores, que representam os três pontos de um triângulo.

```
ponto pontoA;
pontoA.y = p1[0];
pontoA.x = p1[1];
pontoA.z = p1[2];

ponto pontoB;
pontoB.y = p3[0];
pontoB.x = p3[1];
pontoB.z = p3[2];

ponto pontoC;
pontoC.y = p4[0];
pontoC.x = p4[1];
pontoC.z = p4[2];

pontos << formTriangle(pontoA, pontoB, pontoC) << endl;

ponto pontoD;
pontoD.y = p2[0];
pontoD.x = p2[1];
pontoD.z = p2[2];

ponto pontoE;
pontoE.y = p1[0];
pontoE.x = p1[1];
pontoE.z = p1[2];

ponto pontoF;
pontoF.y = p4[0];
pontoF.x = p4[1];
pontoF.z = p4[2];

pontos << formTriangle(pontoD, pontoE, pontoF) << endl;
```

Figura 8: escrita dos pontos do teapot num ficheiro auxiliar

As aplicações das operações sobre matrizes (contidas no formulário) estão presentes na função designada por “bezierPoint”, como se pode verificar através da figura 9.

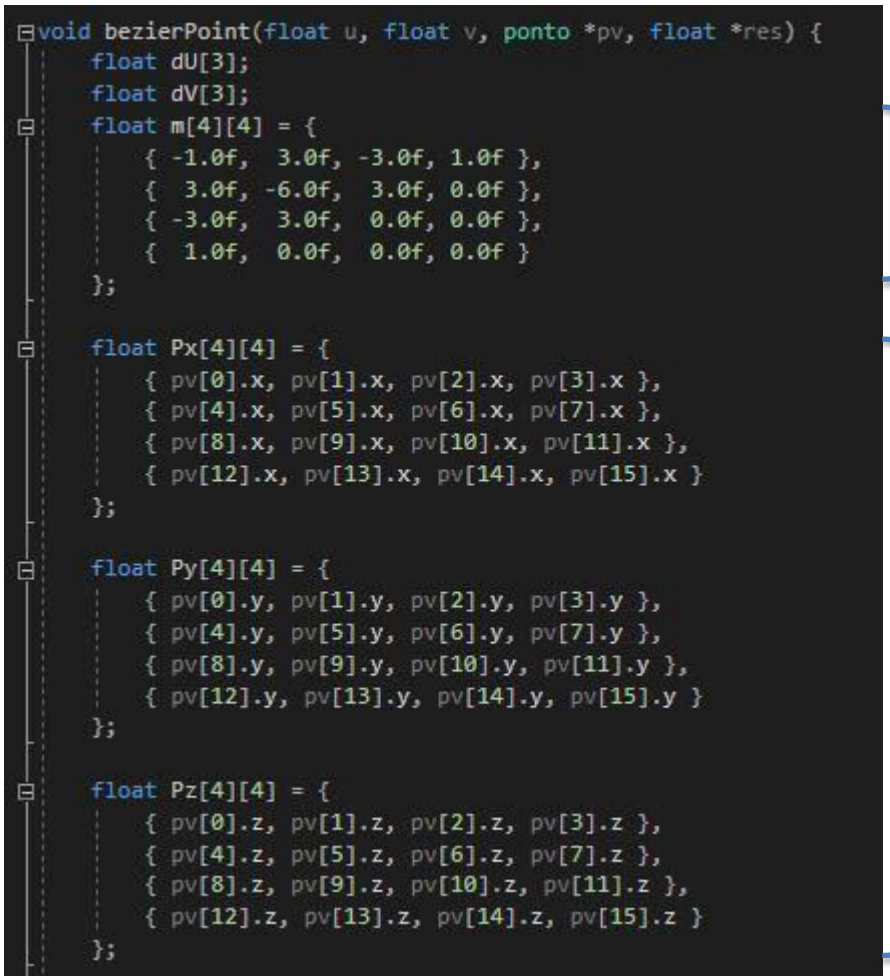


Figura 9: obtenção de um ponto usando operações sobre matrizes

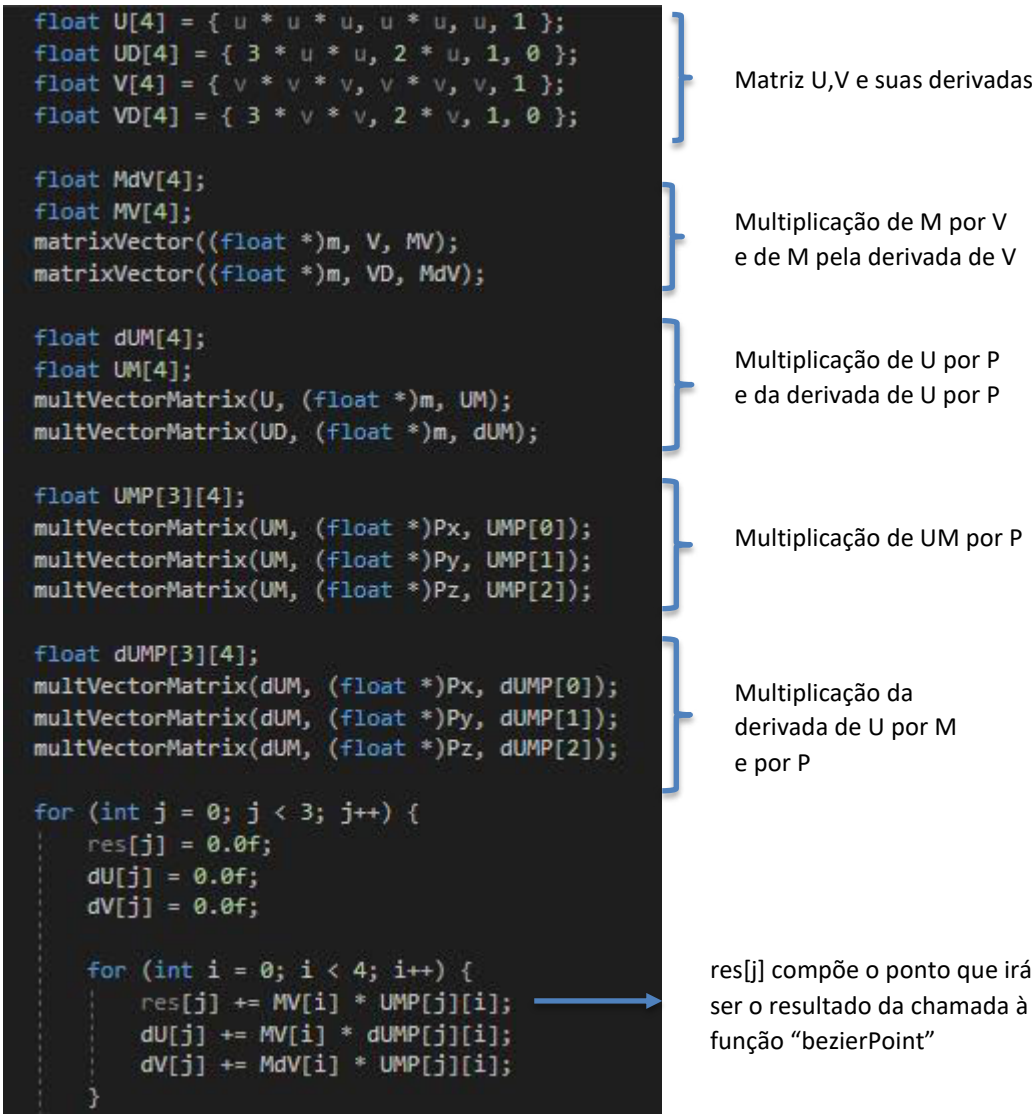


Figura 10: aplicação de operações sobre matrizes

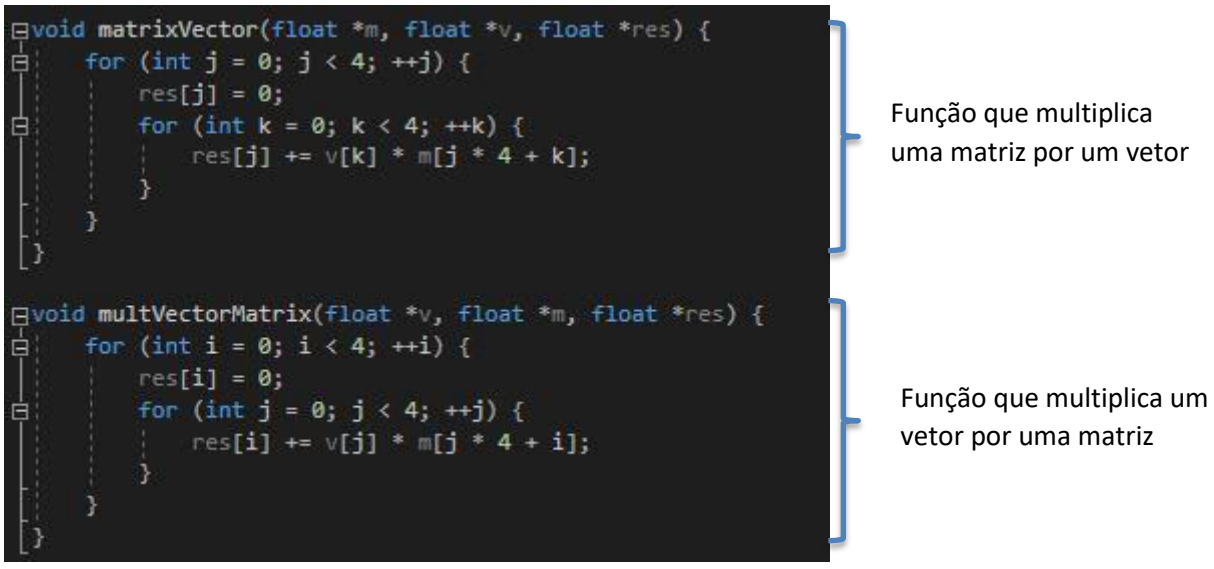


Figura 11: funções de multiplicação entre matrizes e vetores

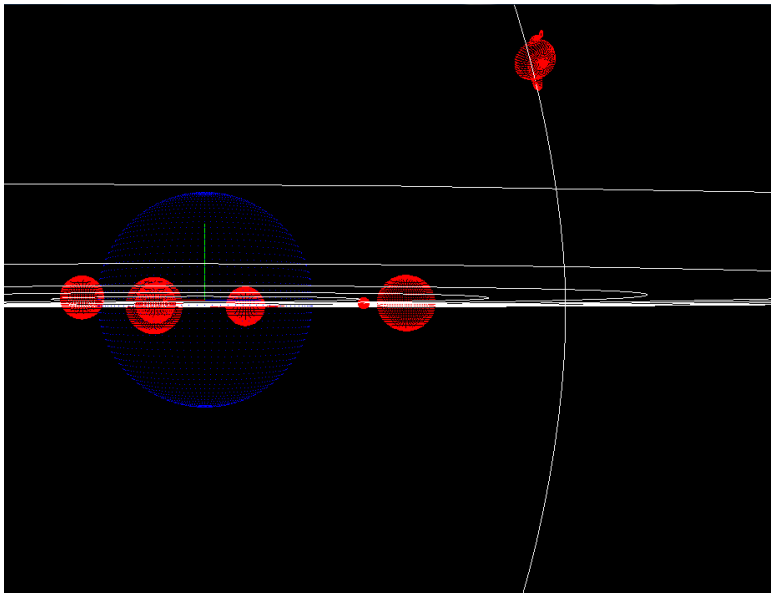


Figura 12: desenho do teapot usando patches de Bezier

5. Catmull-Rom

Neste capítulo irá ser abordada a forma como foram desenhadas as órbitas dos planetas, bem como irão ser mencionadas as suas translações segundo as curvas de Catmull-Rom.

Relativamente às figuras a desenhar, foram aplicadas translações e desenhadas as suas órbitas, partindo das operações sobre matrizes que são fornecidas no formulário de "Curvas e Superfícies".

Tanto num como noutro caso, as coordenadas dos pontos a aplicar (cada ponto representado pela variável pos) foram dadas através da aplicação da função "getCatmullRomPoint".

```
while (j < documents[i].nfiles) {
    if (documents[i].npontos > 0) {

        desenhar_Catmull(documents[i]);

        float pos[3];
        float der[3];
        getGlobalCatmullRomPoint(tempos1[i], pos, der, documents[i]);

        glTranslatef(pos[0], pos[1], pos[2]);
        rotate(i);
```

Figura 13: desenho das óritas e aplicação do translate a cada figura

```
void desenhar_Catmull(struct Xml dc) {
    int i = 0;
    glColor3f(1, 1, 1);
    glBegin(GL_LINE_LOOP);

    for (i = 0; i < 30; i++) {
        float pos[3];
        float derivada[3];
        getGlobalCatmullRomPoint(i / 100.f, pos, derivada,dc);
        glColor3f(1, 1, 1);
        glVertex3f(pos[0], pos[1], pos[2]);
    }
    glEnd();
}
```

Figura 14: função que desenha a órbita de uma figura

A função “getGlobalCatmullPoint”, partindo de pontos de controlo que estão definidos num ficheiro XML, e que foram armazenados numa estrutura auxiliar, invoca a função “getCatmullRomPoint”. São assim definidos pontos que vão servir não só para serem usados no desenho da órbita, como também na translação à figura em que são aplicados.

```
void getGlobalCatmullRomPoint(float gt, float *pos, float *derivada,struct Xml dc) {
    float t = gt * POINT_COUNT;
    float p[POINT_COUNT][3];
    int index = floor(t);
    t = t - index;

    for (int i = 0; i < POINT_COUNT; i++) {
        p[i][0] = dc.points[i].x;
        p[i][1] = dc.points[i].y;
        p[i][2] = dc.points[i].z;
    }

    int inddexes[4];
    inddexes[0] = (index + POINT_COUNT - 1) % POINT_COUNT;
    inddexes[1] = (inddexes[0] + 1) % POINT_COUNT;
    inddexes[2] = (inddexes[1] + 1) % POINT_COUNT;
    inddexes[3] = (inddexes[2] + 1) % POINT_COUNT;

    getCatmullRomPoint(t, p[inddexes[0]], p[inddexes[1]], p[inddexes[2]], p[inddexes[3]], pos, derivada);
}
```

Estrutura onde são armazenados os pontos de controlo para uma curva

Figura 15: função que opera sobre os pontos de controlo definidos em XML

```
void getCatmullRomPoint(float t, float *p0, float *p1, float *p2, float *p3, float *pos, float *derivada) {
    // 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} };

    float a[3][4];
    // Compute A = M * P
    //float *v[4] = {p0, p1, p2, p3};
    float x[4] = { p0[0],p1[0],p2[0],p3[0] };
    float y[4] = { p0[1],p1[1],p2[1],p3[1] };
    float z[4] = { p0[2],p1[2],p2[2],p3[2] };
    matriz_multiplicacao(m[0], x, a[0]);
    matriz_multiplicacao(m[0], y, a[1]);
    matriz_multiplicacao(m[0], z, a[2]);

    // Compute pos = T * A
    float T[4] = { (float)pow((double)t,3), (float)pow((double)t,2), t, 1 };
    pos[0] = { T[0] * a[0][0] + T[1] * a[0][1] + T[2] * a[0][2] + T[3] * a[0][3] };
    pos[1] = { T[0] * a[1][0] + T[1] * a[1][1] + T[2] * a[1][2] + T[3] * a[1][3] };
    pos[2] = { T[0] * a[2][0] + T[1] * a[2][1] + T[2] * a[2][2] + T[3] * a[2][3] };

    float d[4] = { (float)pow((double)3 * t,2), 2 * t, 1, 1 };
    derivada[0] = { d[0] * a[0][0] + d[1] * a[1][0] + d[2] * a[2][0] + d[3] * a[3][0] };
    derivada[1] = { d[0] * a[0][1] + d[1] * a[1][1] + d[2] * a[2][1] + d[3] * a[3][1] };
    derivada[2] = { d[0] * a[0][2] + d[1] * a[1][2] + d[2] * a[2][2] + d[3] * a[3][2] };
}
```

Matriz M

Multiplicação de m[0] por cada um dos vetores de P0,P1,P2 e P3

Expressão (23) do formulário

Figura 16: função que obtém de um ponto usando operações do formulário

6. Rotações, translações e ficheiro XML

Para obtermos os tempos reais das translações e rotações de cada planeta, foram usados sites de apoio. No entanto nenhum destes tempos está representado à escala, pois caso isso acontecesse, haveriam planetas que se encontravam "parados", uma vez que têm tempos de translação enormes.

```
<group>
  <rotate time="10" axisX="0" axisY="1" axisZ="0"/>
  <scale X="70" Y="70" Z="70"/>
  <models>
    <model file="..\..\..\Fig_files\sphere.txt" />
  </models>
</group>

<group>
  <rotate time="3" axisX="0" axisY="1" axisZ="0"/>
  <translate time="0.0009">
    <point X="0.000000" Y="0" Z="200.000000"/>
    <point X="141.421356" Y="0" Z="141.421356"/>
    <point X="200.000000" Y="0" Z="0.000000"/>
    <point X="141.421356" Y="0" Z="-141.421356"/>
    <point X="0.000000" Y="0" Z="-200.000000"/>
    <point X="-141.421356" Y="0" Z="-141.421356"/>
    <point X="-200.000000" Y="0" Z="0.000000"/>
    <point X="-141.421356" Y="0" Z="141.421356"/>
  </translate>
  <scale X="12" Y="12" Z="12"/>
  <models>
    <model file="..\..\..\Fig_files\sphere.txt" />
  </models>
</group>
```

Figura 17: excerto do ficheiro XML de teste

Numa fase anterior ao desenho de cada figura, aplicou-se uma rotação a cada uma delas, em que o ângulo de rotação varia consoante a sua posição no conjunto das figuras.

```
void rotate(int i){
    float time = 5000;

    float static angle[20];

    angle[i] += 360 / time;

    glRotatef(angle[i], 0,1,0);
}
```

Figura 18: aplicação da rotação a uma figura

De forma a descobrir os pontos de controlo das órbitras, foi desenvolvido um script auxiliar, que recebe um raio e através desse nos dá oito pontos de controlo da órbitra que pretendemos.

```c
#include <stdio.h>
#include <math.h>
#include <iostream>
#include <fstream>

int main(){
  FILE * pFile;
  int n;

  pFile = fopen ("pontos.txt","w");


  fprintf (pFile,"<point X\"0\"  Y=\"%lf\" Z=\"%lf\"/>\n",0.00,raio);
  fprintf (pFile,"<point X=\"0\" Y=\"%lf\" Z=\"%lf\"/>\n",raio*cos(45 * M_PI / 180),raio*sin(45 * M_PI / 180));
  fprintf (pFile,"<point X=\"0\" Y=\"%lf\" Z=\"%lf\"/>\n",raio,0.00);
  fprintf (pFile,"<point X=\"0\" Y=\"%lf\" Z=\"%lf\"/>\n",raio*cos(45 * M_PI / 180), -raio * sin(45 * M_PI / 180));
  fprintf (pFile,"<point X=\"0\" Y=\"%lf\" Z=\"%lf\"/>\n",0.00,-raio);
  fprintf (pFile,"<point X=\"0\" Y=\"%lf\" Z=\"%lf\"/>\n",-raio * cos(45 * M_PI / 180),-raio * sin(45 * M_PI / 180));
  fprintf (pFile,"<point X=\"0\" Y=\"%lf\" Z=\"%lf\"/>\n",-raio,0.00);
  fprintf (pFile,"<point X=\"0\" Y=\"%lf\" Z=\"%lf\"/>\n",-raio * cos(45 * M_PI / 180),raio*sin(45 * M_PI / 180));

  fclose (pFile);

   return 0;
}
```

Figura 19: script para obtenção dos pontos de controlo de uma órbita

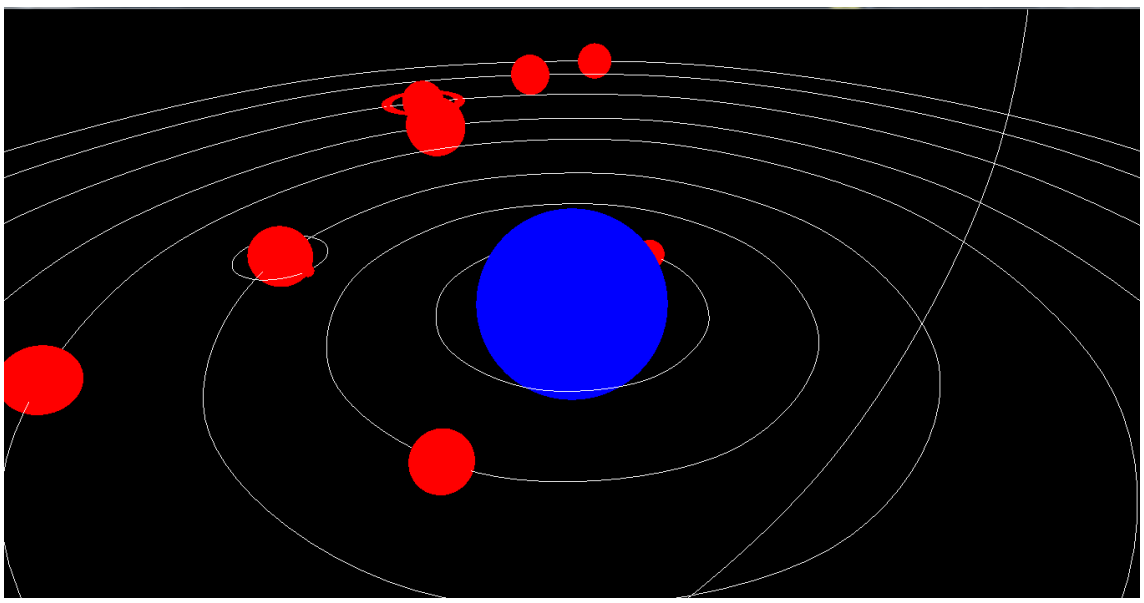


Figura 20: desenho do sistema solar

Conclusão

O grupo considera que alcançou os objetivos propostos para a terceira fase do projeto. Foram desenhadas todas as figuras representativas do sistema solar, implementando VBOs para o efeito. O desenho do teapot foi conseguido com o uso dos patches de Bezier e todas as figuras se movem em órbitas definidas por curvas de Catmull-Rom. Para além disto, todas as figuras do sistema solar rodam sobre si mesmas.

O grupo verificou que a implementação dos VBOs aumenta significativamente o desempenho no desenho das figuras. O parse teve de ser editado para conseguirmos responder aos requisitos propostos para esta fase do projeto.

Com a finalização desta fase, o projeto assume proporções bastante significativas, tendo o grupo concluído que o trabalho realizado corresponde a uma boa implementação das bases fornecidas pelo docente da Unidade Curricular de *Computação Gráfica*.