Computação Gráfica: Fase Nº3

Universidade do Minho

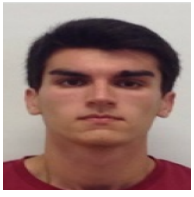
Mestrado Integrado em Engenharia Informática

Grupo 14

Flávio Martins
A65277

Gonçalo Costeira
A79799

José Ramos
A73855



Rafael Silva
A74264

Contents

List of Figures

1 Introdução

Na terceira fase do trabalho foi proposto ao grupo a continuação do trabalho realizado na fase anterior, agora recorrendo novos métodos que foram lecionados ao longo do semestre para melhorar a qualidade do características projeto.
Nesta fase implementa-mos curvas e superfícies cúbicas e da leitura de um patch que contem pontos de controlo e graus de tesselação para gerar as imagens.
A demonstração desta fase será feita através dum modelo em movimento do sistema solar, todos os corpos do projeto anterior e mesmo um teapot em órbita do sol.

1.1 Resumo

O trabalho desenvoldido durante a primeira e segunda fase será adaptado e desenvolvido um novo tipo de modelo baseado em Bezier Patches.
Nesta fase irão haver alterações quer ao nível do motor como do gerador.
O gerador este deverá ser capaz de criar um novo modelo baseado em Bezier patches. O gerador passará a receber como parâmetros o nome de um ficheiro, neste ficheiro irão estar definidos os pontos de controlo dos patches e um nível de tecelagem, retornará entao um ficheiro contendo uma lista de triângulos que definem essa superfície.
Quanto ao motor os elementos translate e rotate dos ficheiros XML sofrerão modificações.
O elemento translation será acompanhado por um conjunto de pontos que definem uma Catmull-Rom curve e o número de segundos para percorrer essa curva. Com base nestas curvas será possível criar animações.
Quanto ao elemento rotation o ângulo definido poderá ser substituído pelo número de segundos que o objeto demora a efectuar uma rotação de 360 graus, assim será alterado não só o parser que lê esses ficheiros mas tambem o modo como a informação recebida é processada para gerar o cenário.
Por fim são alterados os modelos gráficos, estes agora serão desenhados com o auxílio de VBOs.
Deste modo será possível gerar um modelo do Sistema Solar mais realista passando de um modelo estático para um modelo dinâmico.

2 Objetivos

Nesta fase de desenvolvimento do projeto devemos , em relação à fase anterior, desenvolver um gerador capaz de implementar um modelo de criação de triângulos que receberá um ficheiro de pontos de controlo e graus de tesselação que correspondera à superfície de Bezier.

Para alem disso o projeto continuará a incorporar o sistema solar desenvolvido na fase anterior que sofrera algumas modificações em prol de melhorar o resultado final, isto é, pretendemos implementar características como curvas, translação, rotação e a noção de tempo.

3 Contextualização

3.1 VBO's

Nesta fase procuramos uma melhoria na eficiência do software, e para tal utilizamos a VBO's, isto é "vertex buffer object", aonde recorremos a um buffer em grelha que guarda pontos de triângulos a desenhar, permitindo maior rapidez no desenho bem como parsing dos pontos destes.
Assim, é necessário sabermos quantos pontos necessitamos para renderizar um dado objeto pois a partir do momento em que o sabemos temos de abrir o respetivo buffer e alimentado com a respetiva informação (os pontos).
Em seguida o render invocará um conjunto de funções que trataram do processamento da informação no buffer.
Para acomodar o software a este novo método, tivemos de alterar a função GRload definida na classe estruturas.h, elaborada anteriormente, tal que fosse possível a alimentação dos dados de VBO's nesta(isto é , inserção das coordenadas de cada um dos pontos) .

```
verticeB = (float*) malloc(nPoints*sizeof(float));
glEnableClientState(GL_VERTEX_ARRAY);

while(getline(f,line)){
    float x,y,z;
    sscanf(line.c_str(),"%f %f %f\n",&x,&y,&z);
    verticeB[j++] = x; verticeB[j++] = y; verticeB[j++] = z;
}
```

3.2 Curvas de Bezier

Curva de Bezier é um tipo de curva polinomial que exprimimos através de uma interpolação linear entre pontos representativos que escolhemos, chamados estes pontos de controlo. De modo a construir uma dada curva de Bezier necessitamos, no mínimo, de 3 pontos de controlo, havendo a possibilidade de utilizar mais destes, sendo que a forma mais de terceira ordem (com quatro pontos de controlo) a *curva cúbica de Bezier* é a mais utilizada. Estes quatro pontos destingem-se por:

- 2 endpoints (também conhecidos como pontos ancoras).
- 2 control points (que não pertencem a curva mas definem a sua forma e curvatura.

A linha que une um ponto âncora ao seu ponto de controlo é a reta tangente à curva no ponto âncora sendo ela que determina o declive da curva neste ponto. Esta método de especificação de curvas de Bezier encontra-se ilustrada na figura seguinte.

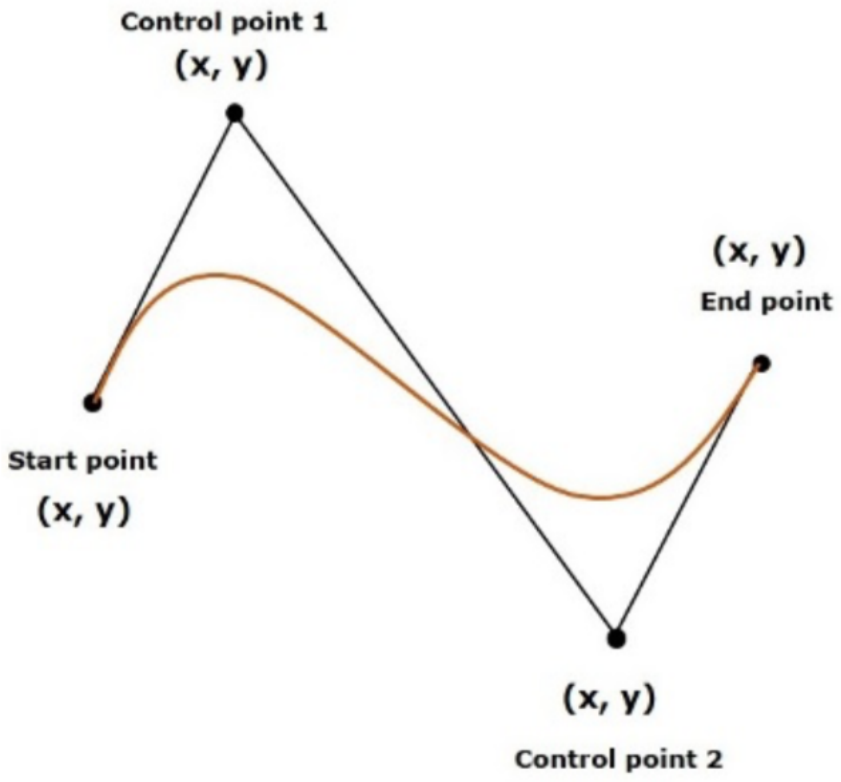


Figure 1: Curva Bezier com 4 pontos

3.3 Superficies de Bezier

Após desenvolvermos capacidades no tratamento e aplicação das curvas de Bezier debruçamo-nos sobre uma aplicação mais importantes desta teoria, passando a desenvolver superfícies aplicando o método, apenas utilizando a deslocação a dois eixos. Este método de renderização de superfícies permite nos também tabalhar a curvatura das suas linhas, sendo que a quantos mais segmentos de reta recorrermos, maior será a definição do objeto (superficie) renderizada.

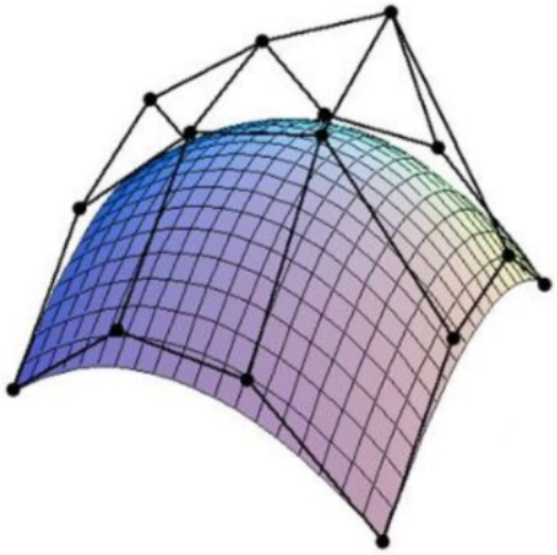


Figure 2: Exemplo de superficie Bezier

4 Estruturas

Para esta terceira fase foi necessário acrescentar mais alguns dados á classes definidas na fase anterior, para corresponder as especificações requeridas. decidimos também representar os três tipos de transformações através de novas classes:

- Translation
  As primeiras três variáveis servem de definição do movimento de translação. Adicionamos a variável tempo representante do intervalo de tempo que um determinado objeto demora a executar a translação.

```
            //Define uma translação
class Translation
{
    public:
            float coordX,coordY,coordZ,time;
            list<Point> points;
};
```

- Rotation

  Nesta transformação acrescentamos o float tempo que trata intervalo de tempo que um objeto demora a realizar uma rotação.

```
            //Define uma Rotacao
class Rotation
{
    public:
            float angle,axisX,axisY,axisZ,time;
};
```

- Scale

  Esta classe, confere suporte de modo a se poder fazer alterações ao nível das dimensões do objeto.

```
class Scale
{
    public:
            float scale_x,scale_y,scale_z;
};
```

- Orbit
  Esta classe representa os raios de uma orbita necessários a definição da elipse (2 raios) bem como nControl que trata os pontos de controlo da curva de Bezier.

```
            //Define os raios da orbita
class Orbit
{
    public:
        float radiusX,radiusZ;
        int nControl;
};
```

- Model
  Adicionamos a variavel buffers que guardará inteiros do OpenGL.

```
            //Define uma lista de pontos
class Model
{
    public:
            string file;
            int points;
        GLuint buffers[2];
};
```

- Group
  Adicionamos a variavel buffers que guardará inteiros do OpenGL.

```
            //Reunião de todas as estruturas
class Group
{
    public:
            string type;        // define se é planeta ou orbita
            Translation translation;
            Rotation rotation;
            Scale scale;
            Color color;
            Orbit orbit;
            list<Model> models;
            list<Group> groups;
};
```

5 Gerador

5.1 Patch

Nesta fase adicionamos um novo método drawPatch que lê o ficheiro Patch, transformando a informação que contida neste e a guarda no (savefilename). O passo intermédio consiste em na interpretar os pontos de controlo através das regras necessárias e aplicar com o algoritmo de Bezier (encontra os pontos finais através dos pontos de controlo) renderizando as superfícies necessárias.

5.1.1 Estrutura

- A primeira linha um inteiro correspondente ao número de patches;
- De seguida, encontra-se a representação de todos os patches, cada um com 16 ındices;
- Depois, uma linha com um inteiro que corresponde ao número de pontos de controlo;
- Por fim, é apresentada uma lista com todas as coordenadas dos respetivos pontos de controlo;

A função drawPatch, está dividida em duas partes, a primeira relacionada com o parse do ficheiro que é passado como parâmetro e a segunda alusiva ao cálculo dos pontos e à inserção destes num ficheiro final.

5.1.2 Parsing

O parsing é realizado no inicio da função drawPatch, é aberto o ficheiro que é recebido como argumento na função e recolhida a informação (seguindo a ordem da estrutura descrita na secção anterior). Em primeiro lugar é guardado o número de patches e de pontos composto por dois inteiros. Os índices de cada patch são guardados numa estrutura do tipo vector $<$ vector $<$ int $>>$, que tem como dimensão a multiplicação do número de patches com o número de índices, 16 neste caso.

$$\text{Dimensão} = \text{nP} * \text{nI}$$

Cada ponto de controlo é guardado numa célula da estrutura, vector $<$ Point $>$, que tem de tamanho o valor já guardado no inteiro com o número de pontos.
De seguida apresenta-se o excerto da função descrita acima:

```
//--- PARSING PATCH ---//
    std::fstream f;
    f.open(patchfilename,std::ios::in);
    int patches=0;
```

```
int vertices=0;
std::vector<std::vector<int> > indicesPatch;
std::vector<Point> vertixs;

if(f.is_open()){
    std::string line;
    if(getline(f,line)) sscanf(line.c_str(),"%d\n",&patches);
    for(int i=0; i<patches ;i++){
         std::vector<int> aux;
        if(getline(f,line)){
            int n1=0,n2=0,n3=0,n4=0,n5=0,n6=0,n7=0,n8=0,n9=0,n10=0,
            n11=0,n12=0,n13=0,n14=0,n15=0,n16=0;
            sscanf(line.c_str(),"%d, %d, %d, %d, %d, %d, %d, %d, %d,
            %d, %d, %d, %d, %d, %d, %d\n",&n1,&n2,&n3,&n4,&n5,&n6,&n7,
            &n8,&n9,&n10,&n11,&n12,&n13,&n14,&n15,&n16);
            aux.push_back(n1);
            aux.push_back(n2);
            aux.push_back(n3);
            aux.push_back(n4);
            aux.push_back(n5);
            aux.push_back(n6);
            aux.push_back(n7);
            aux.push_back(n8);
            aux.push_back(n9);
            aux.push_back(n10);
            aux.push_back(n11);
            aux.push_back(n12);
            aux.push_back(n13);
            aux.push_back(n14);
            aux.push_back(n15);
            aux.push_back(n16);
        }

        indicesPatch.push_back(aux);
    }
    if(getline(f,line)) sscanf(line.c_str(),"%d\n",&vertices);
    for(int i=0; i<vertices ;i++){
        float x=0,y=0,z=0;
        if(getline(f,line)) sscanf(line.c_str(),"%f,%f,%f\n",&x,&y,&z);

        Point p;
```

```
            p.x = x; p.y = y; p.z = z;

            vertixs.push_back(p);
        }

        f.close();
    }
    else { printf("Error: Not possible acess patch file..."); exit(0); }
```

5.2 Calculo de Pontos

Após concluir o parsing é feito o calculo do número de pontos, segundo o algoritmo de Bezier, que vão definir as quatro curvas de Bezier, cada uma constituída por quatro vértices. A informação gerada irá ser guardada num ficheiro .3d fornecido para a sua escrita, em primeiro é guardado o número total de pontos elaborados e de seguida a lista de coordenadas de cada ponto.

```
float res[3];
float t;
int index, indices[4];
float q[4][tesselation][3],r[tesselation][tesselation][3],tess = 1/((fl
float pontos = patches*(tesselation)*2*(tesselation)*3*3;

std::fstream g;
g.open(savefilename,std::ios::out);

if(g.is_open()){   //numero de pontos
    g << pontos; g << '\n';
    for(int n=0; n<patches; n++){
        //recolher os vértices do array vertixs para o x y e z
        float p[16][3];
        for(int m=0; m<16; m++){
            p[m][0] = vertixs[indicesPatch[n][m]].x;
            p[m][1] = vertixs[indicesPatch[n][m]].y;
            p[m][2] = vertixs[indicesPatch[n][m]].z;
        }
        int j=0,k=0;
        //desenhar as 4 curvas
        for(int i=0; i<15; i+=4){
            indices[0] = i;
            indices[1] = i+1;
            indices[2] = i+2;
            indices[3] = i+3;
            //calcular a curva
            for(int a=0; a<tesselation-1; a++){
                t = a*tess;
                index = floor(t);
                t = t - index;
                res[0] = (-p[indices[0]][0] + 3*p[indices[1]][0]
                - 3*p[indices[2]][0] + p[indices[3]][0])*t*t*t
```

```
        + (3*p[indices[0]][0] - 6*p[indices[1]][0]
        + 3*p[indices[2]][0])*t*t + (-3*p[indices[0]][0]
        + 3*p[indices[1]][0])*t + p[indices[0]][0];
        res[1] = (-p[indices[0]][1] + 3*p[indices[1]][1]
        - 3*p[indices[2]][1] + p[indices[3]][1])*t*t*t
        + (3*p[indices[0]][1] - 6*p[indices[1]][1]
        + 3*p[indices[2]][1])*t*t + (-3*p[indices[0]][1]
        + 3*p[indices[1]][1])*t + p[indices[0]][1];
        res[2] = (-p[indices[0]][2] + 3*p[indices[1]][2]
        -3*p[indices[2]][2] +p[indices[3]][2])*t*t*t
        + (3*p[indices[0]][2] -6*p[indices[1]][2]
        + 3*p[indices[2]][2])*t*t + (-3*p[indices[0]][2]
        +3*p[indices[1]][2])*t + p[indices[0]][2];
        q[j][k][0] = res[0];
        q[j][k][1] = res[1];
        q[j][k][2] = res[2];
        k++;
    }

    t = 1;

    res[0] = (-p[indices[0]][0] +3*p[indices[1]][0]
    -3*p[indices[2]][0] +p[indices[3]][0])*t*t*t
    + (3*p[indices[0]][0] -6*p[indices[1]][0]
    + 3*p[indices[2]][0])*t*t + (-3*p[indices[0]][0]
    + 3*p[indices[1]][0])*t + p[indices[0]][0];
    res[1] = (-p[indices[0]][1] +3*p[indices[1]][1]
    - 3*p[indices[2]][1] +p[indices[3]][1])*t*t*t
    + (3*p[indices[0]][1] -6*p[indices[1]][1]
    + 3*p[indices[2]][1])*t*t + (-3*p[indices[0]][1]
    + 3*p[indices[1]][1])*t + p[indices[0]][1];
    res[2] = (-p[indices[0]][2] +3*p[indices[1]][2]
    - 3*p[indices[2]][2] +p[indices[3]][2])*t*t*t
    + (3*p[indices[0]][2] -6*p[indices[1]][2]
    + 3*p[indices[2]][2])*t*t + (-3*p[indices[0]][2]
    + 3*p[indices[1]][2])*t + p[indices[0]][2];

    q[j][k][0] = res[0];
    q[j][k][1] = res[1];
    q[j][k][2] = res[2];
    j++;
```

```
        k=0;
    }

    for(int j=0; j<tesselation; j++){
        for(int a=0; a<tesselation-1; a++){
            t = a*tess;;
            index = floor(t);
            t = t - index;

            res[0] = (-q[0][j][0] +3*q[1][j][0] -3*q[2][j][0]
            + q[3][j][0])*t*t*t + (3*q[0][j][0] - 6*q[1][j][0]
            + 3*q[2][j][0])*t*t + (-3*q[0][j][0] + 3*q[1][j][0])*t
            + q[0][j][0];
            res[1] = (-q[0][j][1] +3*q[1][j][1] -3*q[2][j][1]
            + q[3][j][1])*t*t*t + (3*q[0][j][1] - 6*q[1][j][1]
            + 3*q[2][j][1])*t*t + (-3*q[0][j][1] + 3*q[1][j][1])*t
            + q[0][j][1];
            res[2] = (-q[0][j][2] +3*q[1][j][2] - 3*q[2][j][2]
            + q[3][j][2])*t*t*t + (3*q[0][j][2] -6*q[1][j][2]
            + 3*q[2][j][2])*t*t + (-3*q[0][j][2] + 3*q[1][j][2])*t
            + q[0][j][2];
            r[j][k][0] = res[0];
            r[j][k][1] = res[1];
            r[j][k][2] = res[2];
            k++;
        }

        t = 1;

        res[0] = (-q[0][j][0] + 3*q[1][j][0] - 3*q[2][j][0]
        + q[3][j][0])*t*t*t + (3*q[0][j][0] -6*q[1][j][0]
        + 3*q[2][j][0])*t*t + (-3*q[0][j][0] + 3*q[1][j][0])*t
        + q[0][j][0];
        res[1] = (-q[0][j][1] + 3*q[1][j][1] - 3*q[2][j][1]
        + q[3][j][1])*t*t*t + (3*q[0][j][1] - 6*q[1][j][1]
        + 3*q[2][j][1])*t*t + (-3*q[0][j][1] +3*q[1][j][1])*t
        + q[0][j][1];
        res[2] = (-q[0][j][2] +3*q[1][j][2] -3*q[2][j][2]
        + q[3][j][2])*t*t*t + (3*q[0][j][2] -6*q[1][j][2]
        + 3*q[2][j][2])*t*t + (-3*q[0][j][2] +3*q[1][j][2])*t
        + q[0][j][2];
```

```
                r[j][k][0] = res[0];
                r[j][k][1] = res[1];
                r[j][k][2] = res[2];
                k=0;
            }

            for(int i=0; i<tesselation-1; i++)
                for(int j=0; j<tesselation-1; j++){
                    g << r[i][j][0]; g << ' '; g << r[i][j][1]; g << ' ';
                    g << r[i][j][2]; g << '\n';
                    g << r[i+1][j][0]; g << ' '; g << r[i+1][j][1];
                    g << ' '; g <<r[i+1][j][2];  g << '\n';
                    g << r[i][j+1][0]; g << ' '; g << r[i][j+1][1];
                    g << ' '; g << r[i][j+1][2]; g << '\n';

                    g << r[i+1][j][0];   g << ' '; g << r[i+1][j][1];
                    g << ' '; g << r[i+1][j][2];   g << '\n';
                    g << r[i+1][j+1][0]; g << ' '; g << r[i+1][j+1][1];
                    g << ' '; g << r[i+1][j+1][2]; g << '\n';
                    g << r[i][j+1][0];   g << ' '; g << r[i][j+1][1];
                    g << ' '; g << r[i][j+1][2];   g << '\n';
                }
        }
        g.close();
    }
    else { printf("Error: Cannot open file...\n"); exit(0); }
}
```

6 Motor

6.1 XML

A classe Spec da estrutura foi alterada, sendo que deu origem a duas tags novas no ficheiro XML.
Assim a tag spec foi separada em duas tags distintas:
Na órbita foi fragmentada em rotation e translation.
Nos planetas e luas dividida em rotation e scale.
Foi ainda necessário adicionar uma tag point, e ainda se inseriu um atributo time às tags rotation e translation, na tag orbit foi adicionado o atributo nControl.

```
<!-- orbit mercurio -->
<group>
    <rotation angle=7 axisZ=1 />
    <translation X=7.2 />
    <color R=1 G=0.5 B=0 />
    <orbit radiusX=35 radiusZ=34.25 nControl=16 />

    <!-- mercurio -->
    <group>
        <translation time=4.82 >
            <point X=35.000000 Z=0.000000 />
            <point X=32.335785 Z=13.106908 />
            (...)
        </translation>
        <rotation time=12.69 axisY=1 />
        <scale X=0.383 Y=0.383 Z=0.383 />
        <color R=0.7 G=0.7 B=0.7 />
        <models>
            <model file="./SolarSystem/mercury.3d" />
        </models>
    </group>
</group>
```

6.2 Parsing

Como já descrito, as classes rotation e translation viram adicionada uma variável time, esta variável indica o tempo que demora uma rotação/translação inteira. A classe translation contém ainda uma lista de pontos de controlo numa curva cúbica, esta curva cúbica contém uma lista de pontos de controlo associada á orbita declarada como nControl.
Na função que mostramos de seguida foram realizadas algumas alterações de modo a implementar as novas funcionalidades.

```
void GRload(TiXmlNode *currentNode, list<Group> *groups){
    //percorre os grupos
    for(TiXmlNode* gNode = currentNode->FirstChild("group") ; gNode;
    gNode = gNode->NextSiblingElement()){
        Group g;

        //Translacao
        Translation t;
        TiXmlNode* tNode = gNode->FirstChild("translation");
        if(tNode){
            TiXmlElement* tElem = tNode->ToElement();
            const char *tTime = tElem->Attribute("time");
            if(tTime) t.time = atof(tTime); else t.time = -1;

            if(t.time == -1){
            const char *tX = tElem->Attribute("X"),
            *tY = tElem->Attribute("Y"), *tZ = tElem->Attribute("Z");
            if(tX) t.coordX = atof(tX); else t.coordX = 0;
            if(tY) t.coordY = atof(tY); else t.coordY = 0;
            if(tZ) t.coordZ = atof(tZ); else t.coordZ = 0;
            }
        else {
            TiXmlNode* testNode = tElem->FirstChild("point");
            if(testNode){
                for(TiXmlElement* ptElem =
                tElem->FirstChild("point")->ToElement() ;
                ptElem; ptElem = ptElem->NextSiblingElement()){
                    Point point;
                    const char *tX = ptElem->Attribute("X"),
                    *tY = ptElem->Attribute("Y"),
                    *tZ = ptElem->Attribute("Z");
                    if(tX) point.coordX = atof(tX); else point.coordX = 0;
```

```
                if(tY) point.coordY = atof(tY); else point.coordY = 0;
                if(tZ) point.coordZ = atof(tZ); else point.coordZ = 0;
                t.points.push_back(point);
            }
        }
        else t.time = -1;
        }
    }
    else { t.time = -1; t.coordX = 0; t.coordY = 0; t.coordZ = 0; }
    g.translation = t;

    //Rotacao
    Rotation r;
    TiXmlNode* rNode = gNode->FirstChild("rotation");
    if(rNode){
        TiXmlElement* rElem = rNode->ToElement();
        const char *angle = rElem->Attribute("angle") ,
        *axisX = rElem->Attribute("axisX"),
        *axisY = rElem->Attribute("axisY"),
        *axisZ = rElem->Attribute("axisZ"),
        *rTime = rElem->Attribute("time");
        if(angle) r.angle = atof(angle); else r.angle = 0;
        if(axisX) r.axisX = atof(axisX); else r.axisX = 0;
        if(axisY) r.axisY = atof(axisY); else r.axisY = 0;
        if(axisZ) r.axisZ = atof(axisZ); else r.axisX = 0;
        if(rTime) r.time = atof(rTime); else r.time = -1;
    }
    else { r.angle = 0; r.axisX = 0; r.axisY = 0; r.axisZ = 0;
    r.time = -1; }
    g.rotation = r;

    //Scale
    Scale s;
    TiXmlNode* eNode = gNode->FirstChild("scale");
    if(eNode){
        TiXmlElement* eElem = eNode->ToElement();
        const char *eX = eElem->Attribute("X"),
        *eY = eElem->Attribute("Y"), *eZ = eElem->Attribute("Z");
        if(eX) s.scale_x = atof(eX); else s.scale_x = 1;
        if(eY) s.scale_y = atof(eY); else s.scale_y = 1;
        if(eZ) s.scale_z = atof(eZ); else s.scale_z = 1;
```

```
}
else { s.scale_x = 1; s.scale_y = 1; s.scale_z = 1; }
g.scale = s;

//Color
Color c;
TiXmlNode* cNode = gNode->FirstChild("color");
if(cNode){
    TiXmlElement* cElem = cNode->ToElement();
    const char *R = cElem->Attribute("R"),
    *G = cElem->Attribute("G"), *B = cElem->Attribute("B");
    if(R) c.r = atof(R); else c.r = 1;
    if(G) c.g = atof(G); else c.g = 1;
    if(B) c.b = atof(B); else c.b = 1;
}
else { c.r = 1; c.g = 1; c.b = 1; }
g.color = c;

//Orbita
Orbit o;
TiXmlNode* oNode = gNode->FirstChild("orbit");
if(oNode){
    TiXmlElement* oElem = oNode->ToElement();
    const char *rX = oElem->Attribute("radiusX"),
    *rZ = oElem->Attribute("radiusZ"),
    *nControl = oElem->Attribute("nControl");
    if(rX) o.radiusX = atof(rX); else o.radiusX = 0;
    if(rZ) o.radiusZ = atof(rZ); else o.radiusZ = 0;
    if(nControl) o.nControl = atoi(nControl); else o.nControl = 4;

    if(o.nControl<4) o.nControl = 4;
    g.type = "orbit";
}
else { o.radiusX = 0; o.radiusZ = 0; o.nControl = 4;
g.type = "planeta"; }
g.orbit = o;

//Models
TiXmlNode* mNode = gNode->FirstChild("models");
if(mNode){
    TiXmlElement* mElem = mNode->ToElement();
```

```
TiXmlNode* testNode = mElem->FirstChild("model");
if(testNode){
    for(TiXmlElement* modElem =
    mElem->FirstChild("model")->ToElement(); modElem;
    modElem = modElem->NextSiblingElement()){
        Model m;
        const char *file = modElem->Attribute("file");
        if(file) m.file = file;
        fstream f;
        f.open(m.file.c_str());
        if(f.is_open()){
            string line;
            int nPoints=0, j=0;
            float *verticeB = NULL;

            if(getline(f,line)){
                sscanf(line.c_str(),"%d\n",&nPoints);
                m.points = nPoints;
            }

            verticeB = (float*) malloc(nPoints*sizeof(float));
            glEnableClientState(GL_VERTEX_ARRAY);

            while(getline(f,line)){
                float x,y,z;
                sscanf(line.c_str(),"%f %f %f\n",&x,&y,&z);
                verticeB[j++] = x;
                verticeB[j++] = y; verticeB[j++] = z;
            }

            glGenBuffers(1,m.buffers);
            glBindBuffer(GL_ARRAY_BUFFER,m.buffers[0]);
            glBufferData(GL_ARRAY_BUFFER,m.points*sizeof(float)
            ,verticeB,GL_STATIC_DRAW);
            free(verticeB);

            f.close();
        }
        g.models.push_back(m);
    }
}
```

```
        }

        //Recursividade sobre grupos
        if(gNode) GRload(gNode,&g.groups);
        groups->push_back(g);
    }
}
```

6.3 Rendering

Por fim era necessário que as órbitas fossem desenhadas a partir das curvas Catmull-Rom. Sendo as curvas Catmull-Rom definidas por um conjunto de pontos, começamos por calcular uma matriz(nControl*3), que irá dividir a órbita em nControl segmentos iguais. De modo a obter um grau de precisão aceitável optamos por utilizar 16 pontos de controlo no Sistema Solar.
Os elementos da matriz irão ser utilizados como pontos de controlo e as curvas desenhadas com 100 segmentos uma vez que no ciclo for, em modo GL_LINE_LOOP, a variável será incrementada em 0.01, iniciando em 0 até ao valor de 1.

```
void drawOrbit(float radiusX, float radiusZ, int nControl){
    float res[3];
    float p[nControl][3];
    float t;
    int index, indices[4];

    float doisPi = 2*M_PI, slice = doisPi/nControl;
    for(int i=0; i<nControl ; i++){
        p[i][0] = cos(i*slice)*radiusX;
        p[i][1] = 0;
        p[i][2] = sin(i*slice)*radiusZ;
    }
    glBegin(GL_LINE_LOOP);
    for(float a=0; a<1; a+=0.01){
        t = a*nControl;
        index = floor(t);
        t = t - index;

        indices[0] = (index + nControl-1)%nControl;
        indices[1] = (indices[0]+1)%nControl;
        indices[2] = (indices[1]+1)%nControl;
        indices[3] = (indices[2]+1)%nControl;

        res[0] = (-0.5*p[indices[0]][0] +1.5*p[indices[1]][0]
        - 1.5*p[indices[2]][0] +0.5*p[indices[3]][0])*t*t*t
        + (p[indices[0]][0] -2.5*p[indices[1]][0]
        + 2*p[indices[2]][0] -0.5*p[indices[3]][0])*t*t
        + (-0.5*p[indices[0]][0] +0.5*p[indices[2]][0])*t
        + p[indices[1]][0];
        res[1] = (-0.5*p[indices[0]][1] +1.5*p[indices[1]][1]
```

```
        - 1.5*p[indices[2]][1] +0.5*p[indices[3]][1])*t*t*t
        + (p[indices[0]][1] -2.5*p[indices[1]][1]
        + 2*p[indices[2]][1] -0.5*p[indices[3]][1])*t*t
        + (-0.5*p[indices[0]][1] +0.5*p[indices[2]][1])*t
        + p[indices[1]][1];
        res[2] = (-0.5*p[indices[0]][2] +1.5*p[indices[1]][2]
        - 1.5*p[indices[2]][2] +0.5*p[indices[3]][2])*t*t*t
        + (p[indices[0]][2] -2.5*p[indices[1]][2]
        + 2*p[indices[2]][2] -0.5*p[indices[3]][2])*t*t
        + (-0.5*p[indices[0]][2] +0.5*p[indices[2]][2])*t
        + p[indices[1]][2];

        glVertex3f(res[0],res[1],res[2]);
    }
    glEnd();
}
```

De modo a implementar a nova forma de rotação, na classe getRotation, adicionamos uma váriável time, que indica o tempo em segundos para uma rotação de 360 graus.
Para aplicar o movimento de rotação dos objetos utilizamos as fórulas:

$$\text{angle} = 360/(\text{time}*1000);$$
$$\text{t} = \text{glutGet(GLUT\_ELAPSED\_TIME)};$$
$$\text{t}*\text{angle}$$

A função glutGet(GLUT_ELAPSED_TIME) dá o tempo decorrido desde a execução do glutInit, sendo que este irá aumentar durante a execução.
Ao dividir 360 (graus) pelo tempo multiplicado por 1000, obtem-se um valor decimal, que será a amplitude de rotação do objeto num instante. Assim o valor angle irá dar o ângulo a aplicar na funçao glRotatef, fazendo o objeto rodar uma certa porção num instante de tempo.
A rotação irá ser mais lenta quanto maior o valor da váriavel time.

```
void getRotation(float time , float axisx , float axisy , float axisz){
    float angle = 360/(time*1000);
    int t = glutGet(GLUT_ELAPSED_TIME);
    glRotatef(t*angle, axisx, axisy, axisz);
}
```

Além da rotação foi implementada uma nova forma de translação. Na translãção foi adicionada uma variável time, correspondente ao tempo dado no ficheiro .xml, esta variável

representa o tempo em segundos, necessário para o objeto completar uma volta na sua trajetória. Foi ainda adicionado um array (res[3]) necessário para colocar e alinhar o objeto com a trajetória.
O valor de a é conseguido como o do valor angle na rotação, ou seja, utilizando as seguintes fórmulas:

$$\text{tempo} = \text{glutGet(GLUT\_ELAPSED\_TIME)a} = \text{tempo}/(\text{time*1000});$$

Tal como na rotação o valor recebido pela função glutGet(GLUT_ELAPSED_TIME) é limitado, de seguida é obtido o valor de a, dividindo o tempo por time*1000. O coeficiente obtido em a corresponde à porção de translação na curva.

```
void getTranslation(float time , list<Point> points){
    int nControl = (int) points.size();
    float res[3];
    float p[nControl][3];
    float t,a;
    int index, indices[4], i=0;

    for(list<Point>::iterator itp = points.begin() ;
    itp!=points.end() ; itp++){
        p[i][0] = itp->coordX ;
        p[i][1] = itp->coordY ;
        p[i][2] = itp->coordZ ;
        i++ ;
    }

    float tempo = glutGet(GLUT_ELAPSED_TIME)%(int)(time*1000);
    a = tempo/(time*1000);

    t=a*nControl;
    index = floor(t);
    t=t-index;

    indices[0] = (index + nControl-1)%nControl;
    indices[1] = (indices[0]+1)%nControl;
    indices[2] = (indices[1]+1)%nControl;
    indices[3] = (indices[2]+1)%nControl;

    res[0] = (-0.5*p[indices[0]][0] + 1.5*p[indices[1]][0] -
    1.5*p[indices[2]][0] + 0.5*p[indices[3]][0])*t*t*t +
```

```
    (p[indices[0]][0]-2.5*p[indices[1]][0]
    +2*p[indices[2]][0] -0.5*p[indices[3]][0])*t*t
    +(-0.5*p[indices[0]][0]+0.5*p[indices[2]][0])*t + p[indices[1]][0];

    res[1] = (-0.5*p[indices[0]][1] +1.5*p[indices[1]][1] -
    1.5*p[indices[2]][1] +0.5*p[indices[3]][1])*t*t*t
    + (p[indices[0]][1] -2.5*p[indices[1]][1] +2*p[indices[2]][1] -
    0.5*p[indices[3]][1])*t*t +
    (-0.5*p[indices[0]][1] +0.5*p[indices[2]][1])*t +
    p[indices[1]][1];

    res[2] = (-0.5*p[indices[0]][2] +1.5*p[indices[1]][2]
    -1.5*p[indices[2]][2]+0.5*p[indices[3]][2])*t*t*t
    +(p[indices[0]][2] -2.5*p[indices[1]][2] +2*p[indices[2]][2]
    -0.5*p[indices[3]][2])*t*t +(-0.5*p[indices[0]][2]
    +0.5*p[indices[2]][2])*t + p[indices[1]][2];

    glTranslatef(res[0], res[1], res[2]);
}
```

Para possibilitar o uso de VBOs e realizar as alterações descritas nesta terceira fase a função renderGroup foi alterado de modo a implementer as animações.

```
void renderGroup(list<Group>::iterator g){
    // --- pushes original matrix orientation --- //
    glPushMatrix();

    if(!strcmp(g->type.c_str(),"orbit")){
        glRotatef(g->rotation.angle, g->rotation.axisX,
        g->rotation.axisY, g->rotation.axisZ);
        glTranslatef(g->translation.coordX, g->translation.coordY,
        g->translation.coordZ);
    }
    else{
        if(g->translation.time == -1) glTranslatef(g->translation.coordX,
        g->translation.coordY, g->translation.coordZ);
            else getTranslation(g->translation.time,g->translation.points);

        if(g->rotation.time == -1) glRotatef(g->rotation.angle,
        g->rotation.axisX, g->rotation.axisY, g->rotation.axisZ);
         else getRotation(g->rotation.time,g->rotation.axisX,
```

```
            g->rotation.axisY,g->rotation.axisZ);
    }

    glScalef(g->scale.scale_x, g->scale.scale_y, g->scale.scale_z);
    glColor3f(g->color.r,g->color.g,g->color.b);
    drawOrbit(g->orbit.radiusX,g->orbit.radiusZ , g->orbit.nControl);

    for(list<Model>::iterator itm = g->models.begin();
    itm != g->models.end(); itm++){
        glBindBuffer(GL_ARRAY_BUFFER,itm->buffers[0]);
        glVertexPointer(3,GL_FLOAT,0,0);
        glDrawArrays(GL_TRIANGLES,0,itm->points);

    }
    for(list<Group>::iterator itg = g->groups.begin();
    itg != g->groups.end(); itg++)
        renderGroup(itg);

    // --- returns to original matrix orientation --- //
    glPopMatrix();
}
```

7 Resultados

Nesta secção iremos demonstrar os resultados do trabalho realizado durante esta fase do projeto.
Irá ser possível verificar com estas imagens que optamos por distinguir as órbitas pelas suas cores.
Foi ainda gerado um cometa através da utilizações do ficheiro teapot.patch, A sua órbita pode ser distinguida das restantes pela cor, neste caso vermelha.

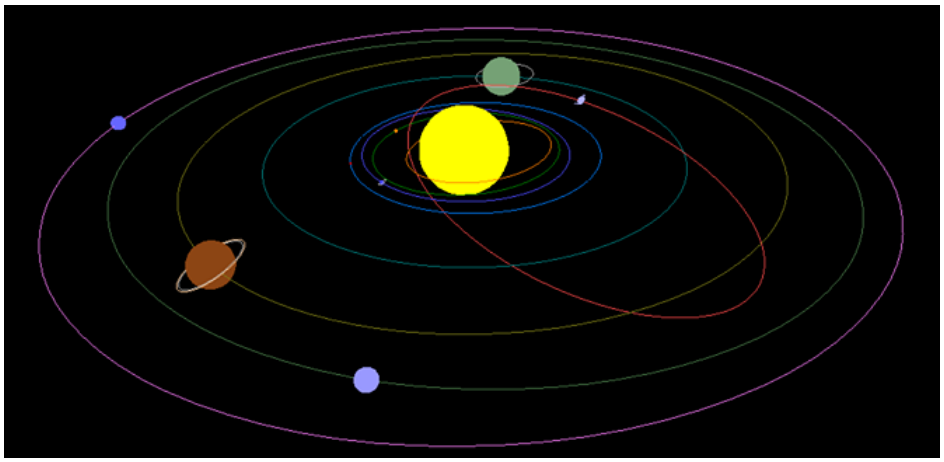


Figure 3: Sistema Solar

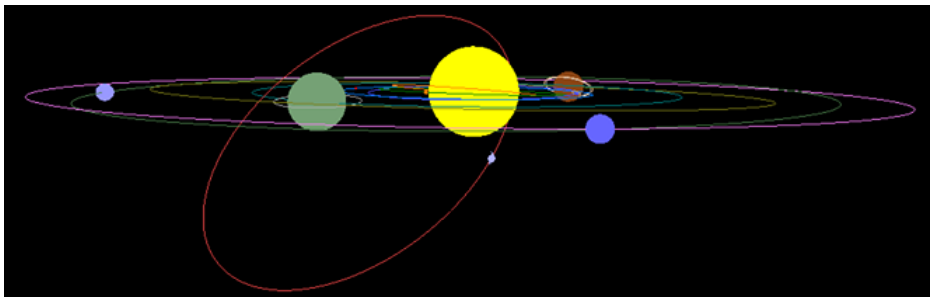


Figure 4: Vista lateral do Sistema Solar

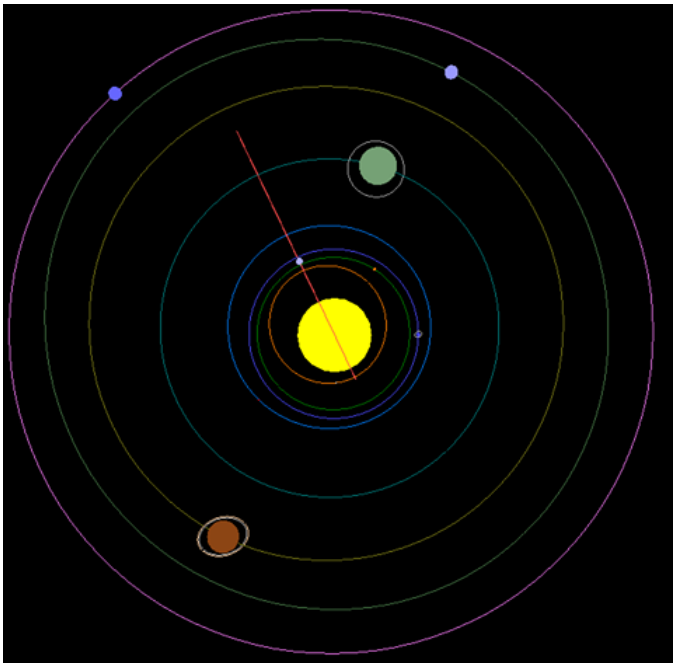


Figure 5: Vista superior do Sistema Solar

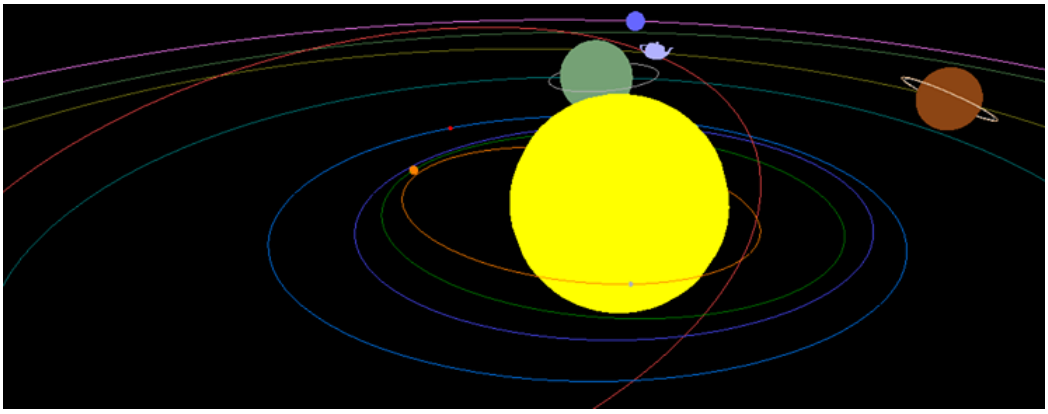


Figure 6: Vista demonstrativa do bule

8 Conclusão

No final de mais uma fase o grupo sente novamente que esta foi bastante produtiva de modo a assimilar os conteúdos lecionados na Unidade Curricular de Computação Gráfica. Foram implementadas curvas de Bézier, rotações e translações cíclicas dos constituintes do Sistema Solar e utilizadas funcionalidades VBO.
Nesta fase o grupo sentiu algumas dificuldades, principalmente na implementação de Bezier patches, para a sua implementação foi necessária alguma atenção especial e pesquisa.
No final o grupo considera que conseguiu concluir com sucesso mais uma fase do trabalho prático, sendo esta uma fase que permitiu ao grupo atingir um conhecimento cada vez mais aprofundado nesta área.