Universidade do Minho
Mestrado Integrado em Engenharia Informática
Departamento de Informática

Computação Gráfica

# Curvas, Superficies Cúbicas e VBOs

Isabel Sofia da Costa Pereira A76550
José Francisco Gonçalves Petejo e Igreja Matos A77688
Maria de La Salete Dias Teixeira A75281
Tiago Daniel Amorim Alves A78218

29 de Abril de 2018

# Conteúdo

# 1 Introdução

Neste projeto, desenvolvido no âmbito da UC de Computação Gráfica, foi proposto a realização de uma mini cena gráfica 3D.

Para o desenvolvimento do mesmo foi necessário utilizar certos recursos tais como C++ e OpenGL.

O trabalho está dividido em quatro fases, estando presente neste relatório uma explicação da abordagem tomada para o desenvolvimento da terceira fase, que consiste na extensão do *Rotate* e *Translate*, implementação de curvas de *Catmull-Rom*, desenvolvimento de figuras através de *Patches* de *Bezier* e a aplicação de *VBOs(Vertex Buffer Object)* no desenho das figuras.

## 1.1 Alterações e Conservações

Tendo em conta que esta é a terceira fase do projeto, existem, naturalmente, algumas mudanças na estrutura do código, assim como conservações, ou seja, funcionalidades implementadas na primeira e segunda fases que se mantêm.

Na fase anterior, o XML continha o nome dos ficheiros 3d com as coordenadas a serem representadas e, juntamente com estes, as transformações necessárias a aplicar sobre a figura em questão. No entanto, nesta terceira fase, visto que é necessário representar curvas de *Catmull-Rom*, o ficheiro XML passa a conter um conjunto de pontos fundamentais para a construção dessas mesmas curvas.

Para além disso, como um dos objetivos desta fase é o desenvolvimento de um Sistema Solar dinâmico, acrescentou-se a variável `time` ao ficheiro XML. Esta variável representa a velocidade de rotação do objeto sobre a curva de *Catmull-Rom*, caso esta seja representada num `translate`, ou a velocidade de rotação do objeto sobre si próprio, caso esta seja representada num `rotate`.

Para além das transformações, considerou-se conveniente conservar a informação relativa à cor (`color`) que a figura que se pretende desenhar possua.

Com as mudanças do ficheiro XML e a utilização de VBOs foi, logicamente, necessário reestruturar as estruturas (*struct* e *transform*) que armazenam a informação, o *parser* e o *engine*.

Relativamente à utilização de *Patches* de *Bezier* foi imprescindível alterar o *generator* e o *vertex* e criar a classe *patch*.

Para a demonstração do funcionamento da cena teve-se como objetivo a criação de um modelo dinâmico do Sistema Solar, incluindo um cometa com a trajetória definida usando a curva de *Catmull-Rom* e construído com *patches* de *Bezier*.

# 2 Desenvolvimento do Projeto

Para o desenvolvimento do trabalho foi conveniente utilizar o *generator* e o *engine* que representam as duas aplicações requeridas.

## 2.1 Gerador

O gerador, *generator*, tal como na primeira e segunda fase, é responsável pelo cálculo dos vértices de uma figura primitiva (`plane`, `box`, `cone`, `sphere` e `torus`), guardando todos esses num ficheiro 3d passado como paramêtro. Para tal, o *generator* utiliza a classe *vertex* como auxílio.

Nesta fase, é acrescentada a funcionalidade de criação de um novo modelo que, ao receber um ficheiro com a extensão *patch*, gera os vértices respetivos que elaboram os *Patches* de *Bezier*.

## 2.2 Engine

O *engine*, semelhante às fases anteriores, tem como objetivo apresentar uma janela exibindo os resultados processados através da leitura de um ficheiro XML. Devido às mudanças neste ficheiri, o *engine* foi sujeito a alterações que estão explicadas mais pormenorizadamente adiante.

## 2.3 Classes

Comparativamente à fase anterior, em que foi necessária a implementação de classes como *Transform* e *Struct* para o armazenamento de toda a informação proveniente do ficheiro XML, nesta fase foi necessária apenas a criação de uma nova classe, a classe *Patch*, que será descrita pormenorizadamente de seguida. No entanto, foi útil a alteração de algumas classes já existentes, tal como a classe *Transform*, *Struct* e *vertex*.

### 2.3.1 Point

Esta classe, tal como nas fases anteriores, representa um ponto num referencial a três dimensões, com as coordendas `X`, `Y` e `Z`, o que se torna bastante útil para a representação dos vértices utilizados posteriormente para o desenho dos triângulos que elaboram as figuras primitivas.

```
class Point{

        float x;
        float y;
        float z;

public:
        Point();
        Point(float,float,float);
        float getX();
        float getY();
        float getZ();
        void setX(float);
        void setY(float);
        void setZ(float);
};
```

### 2.3.2 Transform

Tal como na fase anterior, esta classe representa toda a informação de uma determinada transformação geométrica. Desta forma, a classe *Transform* armazena a designação da transformação, que pode ser `translate`, `rotate`, `scale` ou até mesmo uma transformação a nível da cor das figuras (`color`). Além da designação, armazena igualmente os restantes dados associados à transformação presentes no ficheiro XML, como o ângulo e as coordenadas que a constituem.

Como nesta fase houve a alteração do ficheiro XML, foi necessário adicionar a variável tempo e substituir o ponto, que representa as coordenadas da transformação, por um conjunto de pontos, já que o `translate` pode representar uma curva de *Catmull-Rom*.

```
class Transform{

        string name;
        float timeT, ang;
        vector<Point*> pointsL;

public:
        Transform();
        Transform(string,float,float,vector<Point*>);
        string getName();
        float getTime();
        float getAngle();
        vector<Point*> getPoints();
        Point* getPoint();
        void setName(string);
        void setTime(float);
        void setAngle(float);
        void setPoint(vector<Point*>);
        Transform* clone();
};
```

### 2.3.3 Struct

Tal como sugere o nome, esta é a classe principal de armazenamento, que guarda os dados de uma figura retirados do ficheiro XML, cumprindo assim o mesmo propósito que na fase anterior. Desta forma, é possível armazenar nesta estrutura todas as informações da figura, nomeadamente o seu nome, as várias transformações a serem aplicadas sobre esta e os vértices que a constituem.

Como se pretende implementar VBOs para o desenho das figuras, considerou-se conveniente que a classe passasse a possuir igualmente duas novas variáveis:

- GLuint buffer[1]
- float* vertexArray

Com estas novas variáveis é possível associar à figura o VBO correspondente aos seus pontos. Sendo assim, quando é executado o *Parser* sobre o ficheiro XML, é conveniente preencher o `buffer` com os pontos da figura.

Desta forma, desenvolveu-se o método `fillBuffer()`. Neste método, é colocado no `vertexArray` os vértices a desenhar e é preparado o `buffer` para que no *engine* seja apenas necessário invocar um método auxiliar (método `draw()`), que permita o desenho da figura através de triângulos constituidos pelos pontos que já se encontram preparados.

```
void Struct::fillBuffer(){
        Point p;
        int index = 0;

        //existirá tantos floats quanto o número de pontos vezes 3
        //pois cada ponto é constituido por um X, um Y e um Z
        vertex_array = (float*) malloc(sizeof(float) * points.size() * 3);

        //preencher o vertex_array com os pontos do ficheiro 3d
        for (vector<Point *>::const_iterator i = points.begin();
            i != points.end(); ++i) {
                p = **i;
                vertex_array[index] = p.getX();
                vertex_array[index+1] = p.getY();
                vertex_array[index+2] = p.getZ();
                index+=3;
        }

        //geração de 1 buffer
        glGenBuffers(1, buffer);
        //ativação do buffer
        glBindBuffer(GL_ARRAY_BUFFER, buffer[0]);
        //preenchimento do buffer com os pontos do vertex_array
        //e escolha do padrão de desenho
        glBufferData(GL_ARRAY_BUFFER, sizeof(float) * points.size() * 3,
    vertex_array, GL_STATIC_DRAW);
        glEnableClientState(GL_VERTEX_ARRAY);
}
```

```
void Struct::draw(){
        //ativação do buffer.
        glBindBuffer(GL_ARRAY_BUFFER, buffer[0]);
        //especificação do formato dos pontos a ler do buffer,
        //neste caso tratam-se de 3 floats por vértice
        glVertexPointer(3, GL_FLOAT, 0, 0);
        //especificação do método de desenho e do inicio e fim do buffer
        //(triângulos, tal como nos foi proposto no enunciado)
        glDrawArrays(GL_TRIANGLES, 0, points.size() * 3);
}
```

Com estas duas novas variáveis e métodos, o desenho das figuras será optimizado devido ao facto de não ser necessário estar constantemente a aceder aos pontos no *engine*.

Desta forma, a classe *Struct* possui a seguinte estrutura:

```
class Struct{

    string file;
    vector<Transform*> refit;
    vector<Point*> points;
    GLuint buffer[1];
    float* vertexArray;

public:
        Struct();
        Struct(string,vector<Transform*>,vector<Point*>, GLuint, float*);
        string getFile();
        vector<Transform*> getRefit();
        vector<Point*> getPoints();
        GLuint getBuffer();
        float* getVertexArray();
        void setFile(string);
        void setRefit(vector<Transform*>);
        void setPoints(vector<Point*>);
        void setBuffer(GLuint);
        void setVertexArray(float*);
        void addTransform(Transform*);
```

```
        void addTransform(vector<Transform*>);
        void clear();
        int size();
        void fillBuffer();
        void draw();
};
```

### 2.3.4 Patch

Tal como mencionado anteriormente, implementou-se uma nova classe designada *Patch* que tem como objetivo armazenar os pontos de controlo de um *patch*. Para tal, e como se pode observar no excerto de código abaixo, fez-se uso de um `vector<Point*>`.

```
class Patch{

    vector<Point*> controlPoints;

public:
    Patch();
    Patch(vector<Point*>);
    vector<Point*> getControlPoints();
    void setCtrlPoints(vector<Point*>);
};
```

### 2.3.5 Vertex

A classe *Vertex* realiza a geração de vértices que serão guardados no ficheiro com extensão 3d.

Nesta fase, foi necessário calcular os pontos obtidos através de um ficheiro patch. Para tal foram desenvolvidas três funções: `bezierCurve`, `bezierPatch` e `bezierPatchTriangles`.

```
class Vertex{

    vector<Point*> pointsList;

public:
    Vertex();
    Vertex(vector<Point*>);
    vector<Point*> getPointsList();
    void setPointsList(vector<Point*>);
    void makePlane(float);
    void makeBox(float, float, float, int);
    void makeCone(float, float, int, int);
    void makeSphere(float, int, int);
    void makeTorus(float, float, int, int);
    Point* bezierCurve(float, Point*, Point*, Point*, Point*);
    Point* bezierPatch(float, float, vector<Point*>);
    vector<Point*> bezierPatchTriangles(int, vector<Patch*>);
};
```

Com a função `bezierCurve`, que é apresentada de seguida, é possível obter-se qualquer ponto da curva de acordo com o respetivo `t`, sendo este um valor no intervalo de 0 a 1.

```
Point* Vertex::bezierCurve(float t, Point* p0, Point* p1, Point* p2, Point* p3){
    float x, y, z;
    //valores aprendidos para o cálculo da curva de Bezier
    float b0 = (1.0 - t) * (1.0 - t) * (1.0 - t);
    float b1 = 3 * t * (1.0 - t) * (1.0 - t);
    float b2 = 3 * t * t * (1.0 - t);
    float b3 = t * t * t;

    //obter 1 ponto através da multiplicação de matrizes p e b
    x = b0*p0->getX() + b1*p1->getX() + b2*p2->getX() + b3*p3->getX();
    y = b0*p0->getY() + b1*p1->getY() + b2*p2->getY() + b3*p3->getY();
    z = b0*p0->getZ() + b1*p1->getZ() + b2*p2->getZ() + b3*p3->getZ();

    Point* new_point = new Point(x, y, z);
    return new_point;
}
```

A função `bezierPatch`, que é apresentada de seguida, permite lidar com dois parâmetros (U e V) de forma a obter-se qualquer ponto do *patch* de *Bezier*. Como se pode observar na seguinte imagem, cada linha do *patch* irá representar uma curva de *Bezier*.

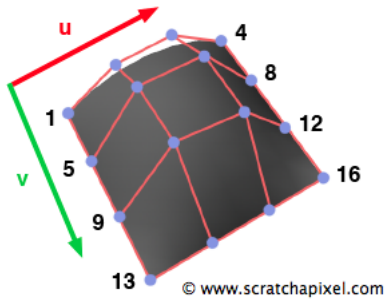


Figura 1: Superfície de Bezier

```
Point* Vertex::bezierPatch(float u, float v, vector<Point*> control_points){
    float matU[4][3], matUV[4][3];
    int i, j=0, k=0;

    //obter os pontos de controlo do patch
    for (i = 0; i < 16; i ++) {
            matU[j][0] = control_points[i]->getX();
            matU[j][1] = control_points[i]->getY();
            matU[j][2] = control_points[i]->getZ();

            j++;

            //quando se obtem 4 pontos de controlo,
            //calcular a sua curva de bezier
            if (j % 4 == 0) {
                    Point* pointU = bezierCurve(u,
                        new Point(matU[0][0],matU[0][1],matU[0][2]),
                        new Point(matU[1][0],matU[1][1],matU[1][2]),
                        new Point(matU[2][0],matU[2][1],matU[2][2]),
                        new Point(matU[3][0],matU[3][1],matU[3][2]));
```

```
                    //adicionar o ponto obtido
                    //à lista de pontos de controlo do patch
                    matUV[k][0] = pointU->getX();
                    matUV[k][1] = pointU->getY();
                    matUV[k][2] = pointU->getZ();

                    k++;
                    j = 0;
                }
            }

            //calcular a curva de bezier com os pontos de controlo do patch
            Point* pointUV = bezierCurve(v,
                        new Point(matUV[0][0],matUV[0][1],matUV[0][2]),
                        new Point(matUV[1][0],matUV[1][1],matUV[1][2]),
                        new Point(matUV[2][0],matUV[2][1],matUV[2][2]),
                        new Point(matUV[3][0],matUV[3][1],matUV[3][2]));

            return pointUV;
}
```

Por fim, foi desenvolvida a função `bezierPatchTriangles` que permite obter os vértices dos triângulos que constituem o *patch* de *Bezier*. Desta forma, esta função poderá ser invocada pelo *generator* para colocar no ficheiro com extensão 3d todos os vértices encontrados.

```
vector<Point*> Vertex::bezierPatchTriangles(int divs, vector<Patch*> patch_list){
        vector<Point*> point_list;
        float u, uu, v, vv;
        float inc = 1.0 / divs; //tesselation

        //percorrer todos os patches e obter os seus vértices
        for(int n_patches = 0; n_patches < patch_list.size(); n_patches++){
            vector<Point*> control_points =
                    patch_list[n_patches]->getControlPoints();

            for(int j=0; j <= divs ; j++){
                    for(int i=0; i <= divs; i++){
                            u = i * inc;
                            v = j * inc;
                            uu = (i+1) * inc;
                            vv = (j+1) * inc;

                            //calculo de cada patch para cada valor de u e v
                            //obtidos através da tesselation
                            Point* p0 = bezierPatch(u, v, control_points);
                            Point* p1 = bezierPatch(u, vv, control_points);
                            Point* p2 = bezierPatch(uu, v, control_points);
                            Point* p3 = bezierPatch(uu, vv, control_points);

                            //adicionar os vértices do triângulo
                            //regra da mão direita
                            point_list.push_back(p0);
                            point_list.push_back(p2);
                            point_list.push_back(p3);

                            point_list.push_back(p0);
                            point_list.push_back(p3);
                            point_list.push_back(p1);
                    }
            }
        }
        return point_list;
}
```

## 2.4 Outros Ficheiros

### 2.4.1 tinyxml2

O *tinyxml2* é uma ferramenta que processa ficheiros XML, sendo de extrema importância para o funcionamento do *Parser*.

```
namespace tinyxml2
{
class XMLDocument;
class XMLElement;
class XMLAttribute;
class XMLComment;
class XMLText;
class XMLDeclaration;
class XMLUnknown;
class XMLPrinter;
...
}
```

### 2.4.2 Parser

Este ficheiro é crucial para o bom funcionamento do *engine* devido ao facto de ser este que efetua o parsing do ficheiro XML. Desta forma, o *Parser* é responsável por inserir toda a informação encontrada no documento XML num vetor de *Struct*.

Como o XML sofreu alterações, este ficheiro também se encontra modificado relativamente à fase anterior. No entanto, a explicação realizada na segunda fase para as funções `parserXML`, `readFile` e `lookFiles` mantem-se. Sendo assim, alterou-se apenas a função `lookAux` e `lookupT` e criou-se a função `lookupTranslate`.

A função `lookAux` continua com o objetivo de percorrer um grupo do XML e os grupos filhos correspondentes, extraindo a informação associada. Sendo assim, esta função retorna um vector de *Struct*. Caso a informação a ler seja uma transformação ou uma cor, na fase anterior apenas se invocava a função `lookupT`, no entanto, nesta fase, tal não se sucede se a transformação for um `translate` devido ao facto de esta transformação poder vir acompanhada de um conjunto de pontos que define a curva de *Catmull-Rom*. Desta forma, é

invocada a nova função `lookupTranslate` que está encarregue de processar essa informação.

Além disso, como nesta fase é necessário o preenchimento do `buffer` de cada *Struct* criada, através da função `fillBuffer()` explicada na secção *2.3.3*, considerou-se conveniente fazê-lo na função `lookAux` antes da adição da *Struct* ao vector.

# 3 Gerador

Tal como mencionado anteriormente, o *generator* é responsável pela criação dos ficheiros 3d que contêm os vértices das figuras, não só para as figuras primitivas das primeiras fases, mas também para superfícies cúbicas. Para tal, foi necessário desenvolver uma função que extraisse a informação relevante deste ficheiro, `makePatch`.

```
vector<Point*> makePatch(int tesselation, string inputFile){
        vector<Patch*> patchList;
        int i, j, k, ind, patches, pos, ctrl;
        float number;
        string l, ctrLine;

        ifstream file;
        //abertura e leitura do ficheiro
        file.open(inputFile.c_str());
        if(file.is_open()){
          getline(file,l);
          //obter número de patches do ficheiro
          patches = atoi(l.c_str());

          //percorrer todos os patches
          for(i=0; i<patches; i++){
            getline(file,l);
            vector<Point*> v;
            Patch* pat = new Patch();

            //percorrer os 16 pontos de controlo
            for(j=0; j<16; j++){
```

```
            //remover informação desnecessaria do ficheiro
            pos = l.find(", ");
            ind = atoi(l.substr(0, pos).c_str());
            l.erase(0, pos+2);
            //obter ponto
            ctrl = 3 + patches + ind;
            ctrLine = getPointsOfLine(inputFile,ctrl);
            Point* p = new Point();
            for(k=0; k<3; k++){
              pos = ctrLine.find(", ");
              number = atof(ctrLine.substr(0, pos).c_str());
              ctrLine.erase(0, pos+2);
              //guardar os valores X, Y e Z para obter um ponto
              if(k==0){
                p->setX(number);
              }
              if(k==1){
                p->setY(number);
              }
              if(k==2){
                p->setZ(number);
              }
            }
            //adicionar ponto à lista v
            v.push_back(p);
          }
          //adicionar patch à lista patchList
          pat->setCtrlPoints(v);
          patchList.push_back(pat);
        }
        //obter os vertices do triangulo através da função
        //bezierPatchTriangles(explicada mais à frente), usando
        //a patchList obtida, e a tesselation passada como argumento
        Vertex* vx = new Vertex();
        vector<Point*> points =
    vx->bezierPatchTriangles(tesselation,patchList);
        return points;
}
```

# 4 Engine

O ficheiro *engine*, tal como referido anteriormente, deve processar a informação de um documento XML e desenhar o seu conteúdo.

Com o auxílio do *parser* essa ação é realizada obtendo-se assim um vetor de *Struct*.

```
vector<Struct> estruturas = lookFiles(argv[1]);
```

Como o *engine* nesta fase deve ser capaz de elaborar cenários de forma estática e dinâmica, este foi alterado para assim conseguir lidar com essa informação.

Desta forma, a função `sistemaSolar`, descrita na fase anterior, é eliminada sendo apenas utilizada a função `figuraPrimitiva` que consegue, nesta fase, lidar com ambos os casos. Assim, essa função é invocada na `RenderScene` para cada *Struct* presente no vector.

Para um melhor entendimento de como esta função elabora o cenário, esta será descrita de seguida.

## 4.1 figuraPrimitiva

Tal como o nome sugere, esta função está encarregue de processar uma figura primitiva de forma estática ou dinâmica.

É importante referir que esta se encontra rodeada por um *glPushMatrix()* e um *glPopMatrix()* para assim, sempre que for aplicada uma transformação aos eixos para o desenho da figura em questão, ser mantido o eixo de origem como a posição de referência para o desenho da próxima figura.

Antes do desenho da figura em si, é necessário aplicar a cor, `color`, e as transformações dos eixos, mais especificamente `translate`, `rotate` e `scale`. Posteriormente, é gerada uma cor aleatória caso não tenha sido definida uma para a figura. No entanto, caso o `rotate` ou o `translate` venham acompanhados com um tempo é necessário utilizar a função `glutGet(GLUT_ELAPSED_TIME)` que calcula o número de milisegundos que passaram desde que o programa foi inicializado. Com esse valor pode-se multiplicar por 360 (equivalente a realizar o movimento completo sobre uma curva de *Catmull-Rom*) para obter-se a velocidade do objeto. Sendo assim, ao aumentar a variável `time`, maior será o tempo necessário para realizar uma rotação completa.

```
float te = glutGet(GLUT_ELAPSED_TIME)/100.f;
float gr = (te*360) / (timeT * 1000);
```

Desta forma, é possível obter um cenário dinâmico utilizando `glRotatef(gr,x,y,z)` no caso do `rotate` ou `orbitaCatmullRom((*t)->Transform::getPoints(), gr)` no caso do `translate`.

Estando os eixos preparados, é possível efetuar o desenho da figura invocando a função `draw()` explicada na secção *2.3.3*.

De seguida é apresentado o código desta função.

```
void figuraPrimitiva(Struct s){
    const char* nameTransf;
    float timeT, angle, x, y, z, te, gr;
    bool cor=false;
    //obter as várias transformações associadas à figura
    vector<Transform*> vt = s.getRefit();

    glPushMatrix();

    //aplicar todas as transformações da figura
    for (vector<Transform *>::const_iterator t = vt.begin();
    t != vt.end(); t++) {
        //verificar qual a transformação em questão
        nameTransf = (*t)->Transform::getName().c_str();

        //se não for um translate então a transformação só contem um ponto
        if(strcmp(nameTransf,"translate")){
            x = (*t)->Transform::getPoint()->Point::getX();
            y = (*t)->Transform::getPoint()->Point::getY();
            z = (*t)->Transform::getPoint()->Point::getZ();

            if (!strcmp(nameTransf,"rotate")) {
                timeT = (*t)->Transform::getTime();
                angle = (*t)->Transform::getAngle();
                if(angle!=0){ //rotação normal do eixo
                    glRotatef(angle,x,y,z);
                }
                else { //rotação dinâmica do eixo
```

```
                te = glutGet(GLUT_ELAPSED_TIME)/100.f;
                gr = (te*360) / (timeT * 1000);
                glRotatef(gr,x,y,z);
            }
        }
        else if (!strcmp(nameTransf,"scale")) {
            glScalef(x,y,z);
        }
        else if (!strcmp(nameTransf,"color")) {
            glColor3f(x,y,z);
            cor=true;
        }
    }
    else{ //caso seja um translate
        timeT = (*t)->Transform::getTime();
        if(timeT==0){ //translate normal do eixo
            x = (*t)->Transform::getPoint()->Point::getX();
            y = (*t)->Transform::getPoint()->Point::getY();
            z = (*t)->Transform::getPoint()->Point::getZ();

            glTranslatef(x,y,z);
        }
        else{ //translate dinâmico do eixo com curvas de Catmull-Rom
            te = glutGet(GLUT_ELAPSED_TIME) % (int)(timeT * 1000);
            gr = te / (timeT * 1000);
            glColor3f(1,1,1);
            orbitaCatmullRom((*t)->Transform::getPoints(), gr);
        }
    }
}

float a, b, c;

//geração de uma cor aleatória para a figura
//caso esta não tenha uma cor associada
if(cor==false) {
    srand(1024);
```

```
            a = (float) rand() / (float) RAND_MAX;
            b = (float) rand() / (float) RAND_MAX;
            c = (float) rand() / (float) RAND_MAX;

            if (a <= 0.1 && b <= 0.1 && c <= 0.1) a = 1;

            glColor3f(a, b, c);
        }

        //desenho da figura
        s.draw();

        glPopMatrix();
    }
```

## 4.2 orbitaCatmullRom

Quando a função `figuraPrimitiva` se encontra no processo de aplicar as transformações associadas a uma figura, esta invoca a função `orbitaCatmullRom` caso identifique um `translate` com `time`.

A função `orbitaCatmullRom` possui como objetivo desenhar a curva de *Catmull-Rom* e realizar o movimento da figura ao longo desta. Sendo assim, esta recebe como parâmetros os pontos de controlo da curva de *Catmull-Rom* associados à figura e a velocidade de rotação da mesma. Esta função requer assim várias funções auxiliares, nomeadamente:

- `getCatmullRomPoint`;
- `getGlobalCatmullRomPoint`;
- `renderCatmullRomCurve`.

Começando por descrever a função `getCatmullRomPoint`, esta tem como objetivo obter qualquer ponto da curva, e a sua derivada, de acordo com um respetivo `t` e os pontos de controlo da curva.

```
void getCatmullRomPoint(float t, float *p0, float *p1, float *p2,
        float *p3, float *pos, float *deriv) {
    //Matriz catmull-rom leccionada nas aulas
    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}};

    // Compute A = M * P
    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]};

    float Ax[4], Ay[4], Az[4];
    multMatrixVector(*m, X, Ax);
    multMatrixVector(*m, Y, Ay);
    multMatrixVector(*m, Z, Az);

    //  pos = T * A
        float T[4] = {t*t*t, t*t, t, 1};

    pos[0] = T[0] * Ax[0] + T[1] * Ax[1]
           + T[2] * Ax[2] + T[3] * Ax[3];
    pos[1] = T[0] * Ay[0] + T[1] * Ay[1]
           + T[2] * Ay[2] + T[3] * Ay[3];
    pos[2] = T[0] * Az[0] + T[1] * Az[1]
           + T[2] * Az[2] + T[3] * Az[3];

    // derivada = T' * A
    float T_d[4] = {3*t*t, 2*t, 1, 0 };

    deriv[0] = T_d[0] * Ax[0] + T_d[1] * Ax[1]
           + T_d[2] * Ax[2] + T_d[3] * Ax[3];
    deriv[1] = T_d[0] * Ay[0] + T_d[1] * Ay[1]
           + T_d[2] * Ay[2] + T_d[3] * Ay[3];
    deriv[2] = T_d[0] * Az[0] + T_d[1] * Az[1]
           + T_d[2] * Az[2] + T_d[3] * Az[3];
}
```

Utilizando a função `getGlobalCatmullRomPoint` consegue-se obter qualquer ponto da curva dado um valor *gt*, comprimido entre 0 e 1.

```
void getGlobalCatmullRomPoint(float gt, float *pos, float *deriv,
    float* p, int POINT_COUNT) {
    float t = gt * POINT_COUNT; //calculo de t
    int index = floor(t); //identificar o segmento que se pretende
    t = t - index; //a posição no segment

    //indices dos pontos de controlo a utilizar
    int indices[4];
    indices[0] = (index + POINT_COUNT-1)%POINT_COUNT;
    indices[1] = (indices[0]+1)%POINT_COUNT;
    indices[2] = (indices[1]+1)%POINT_COUNT;
    indices[3] = (indices[2]+1)%POINT_COUNT;

    //obtenção do ponto da curva
    //consoante o t e os pontos de controlo calculados
    getCatmullRomPoint(t, p+indices[0]*3, p+indices[1]*3,
    p+indices[2]*3, p+indices[3]*3, pos, deriv);
}
```

Por fim, a função `renderCatmullRomCurve` realiza o desenho da curva através das duas funções referidas anteriormente.

```
void renderCatmullRomCurve(float* p, int POINT_COUNT) {
    float pos[3];
    float deriv[3];

    //para cada valor de gt, obter o próximo vertice a desenhar
    glBegin(GL_LINE_LOOP);
    for (float gt = 0; gt <= 1; gt += 0.01) {
        getGlobalCatmullRomPoint(gt, pos, deriv, p, POINT_COUNT);
        glVertex3f(pos[0], pos[1], pos[2]);
    }
    glEnd();
}
```

Desta forma, a função `orbitaCatmullRom` apresenta a seguinte estrutura:

```
void orbitaCatmullRom(vector<Point*> vp, float gr){
    int POINT_COUNT = vp.size();
    float p[POINT_COUNT][3];
    float Y[3] = { 0, 1, 0 }, Z[3], M[16], pos[3], deriv[3];

    //obter todos os pontos de controlo da figura
    for (int i = 0; i < POINT_COUNT; ++i) {
        p[i][0] = vp.at(i)->getX();
        p[i][1] = vp.at(i)->getY();
        p[i][2] = vp.at(i)->getZ();
    }

    //desenho da curva
    renderCatmullRomCurve(*p,POINT_COUNT);

    //calculo e aplicação do movimento da figura
    getGlobalCatmullRomPoint(gr, pos, deriv, *p, POINT_COUNT);
    glTranslatef(pos[0], pos[1], pos[2]);

    //ajuste de matrizes para melhorar o movimento
    cross(deriv, Y, Z);
    cross(Z, deriv, Y);

    normalize(deriv);
    normalize(Z);
    normalize(Y);

    buildRotMatrix(deriv, Y, Z, M);
    glMultMatrixf(M);
}
```

## 4.3 Exemplos de XML

Como exemplo de utilização foram desenvolvidos quatro cenários sendo dois deles estáticos (sistema solar e boneco de neve) e os outros dois dinâmicos (sistema solar e figuras primitivas).

Relativamente aos dois cenários estáticos, estes encontram-se descritos na fase anterior.

Quanto aos cenários dinâmicos, estes serão explicados de seguida.

### 4.3.1 Figuras Primitivas

Julgou-se conveniente a apresentação de um cenário que contivesse todas as figuras desenvolvidas (`plane`, `box`, `cone`, `sphere`, `torus` e o teapot obtido através de `patches` de *Bezier*). Além disso, considerou-se interessante a apresentação de uma curva de *Catmull-Rom* para uma simples observação de como a figura se movimenta sobre a mesma.

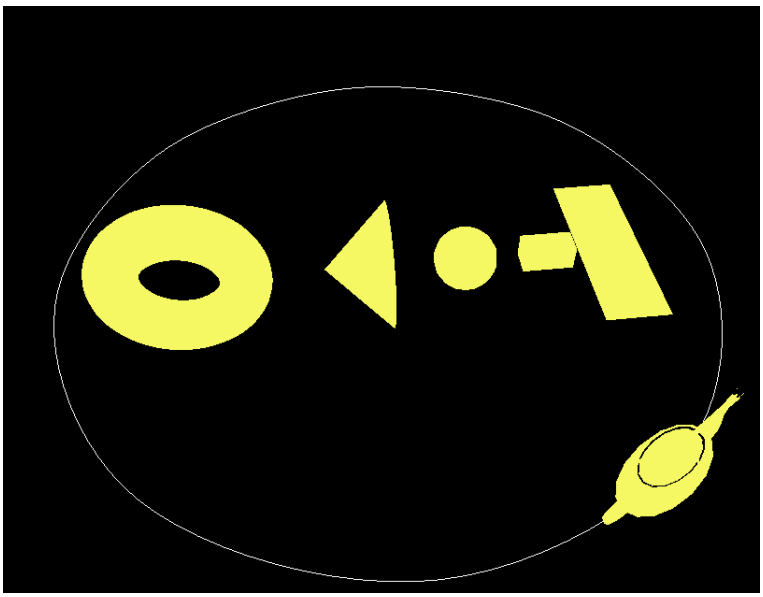

Figura 2: Figuras primitivas e a *teapot*

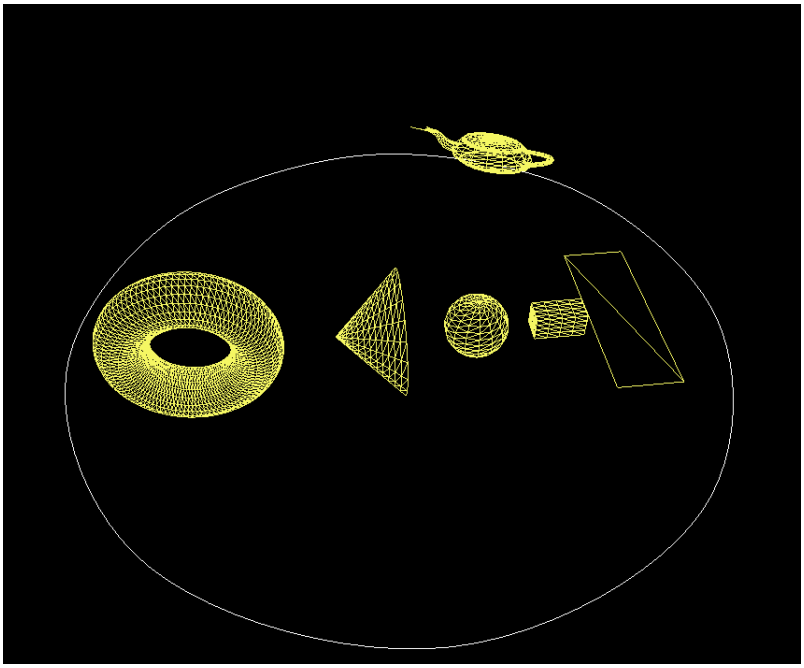

Figura 3: Figuras primitivas e *teapot* - Linhas

De seguida, segue-se o ficheiro XML que produz este cenário. De notar a utilização de todos os tipos de transformações, `translate` (normal e de curva de *Catmull-Rom*), `rotate` e `scale` e a geração, por parte do *engine*, de uma cor aleatória para o cenário.

```
<scene>
    <group>
        <translate x="35" y="0" z="0" />
        <scale x="5" y="10" z="15" />
        <models>
                <model file = "plane.3d" />
        </models>
    </group>
    <group>
        <translate x="25" y="0" z="0" />
        <scale x="2" y="1" z="1" />
        <models>
                <model file = "box.3d" />
```

```
    </models>
</group>
<group>
    <translate x="12" y="0" z="0" />
    <scale x="5" y="5" z="5" />
    <models>
            <model file = "sphere.3d" />
    </models>
</group>
<group>
    <rotate angle="90" x="0" y="0" z="1" />
    <models>
            <model file = "cone.3d" />
    </models>
</group>
<group>
    <translate x="-32" y="0" z="0" />
    <models>
            <model file = "torus.3d" />
    </models>
</group>
<group>
    <translate time="10" >
         <point x="0" y="0" z="50" />
         <point x="35.355" y="0" z="35.355" />
         <point x="50" y="0" z="0" />
         <point x="35.355" y="0" z="-35.355" />
         <point x="0" y="0" z="-50" />
         <point x="-35.3557" y="0" z="-35.355" />
         <point x="-50" y="0" z="0" />
         <point x="-35.355" y="0" z="35.355" />
    </translate>
    <scale x="4" y="2" z="2" />
    <rotate angle="-90" x="1" y="0" z="0" />
    <models>
            <model file = "teapot.3d" />
    </models>
</group>
```

```
</scene>
```

### 4.3.2 Sistema Solar

Para demonstrar o projeto em funcionamento, foi desenvolvido um Sistema Solar dinâmico, tendo por base o Sistema Solar estático demonstrado na fase anterior. De forma a obter as curvas correspondentes às orbitas, foi aplicada a seguinte formula:

```
for{int i = 0; i<360; i = i+45}{
    //considerando r a distância do corpo celeste ao sol
    x=r*sin(i);
    y=0;
    z=r*cos(i);
}
```

Desta forma, conseguiu-se obter os valores para a curva de *Catmull-Rom*, passando-os para o ficheiro XML. O tempo para percorrer a curva foi baseado em tempos reais (aplicados a uma escala muito menor) de forma a obter uma demonstração realista do Sistema Solar. O mesmo se aplicou para a rotação do corpo celestre sobre si próprio.

Além disso, foi adicionado um cometa, construido com *Patches* de *Bezier*, que possui um `rotate`, visto que a sua orbita não está situada no mesmo plano que a os outros planetas, e um `translate`, para alterar o centro da sua orbita visto que não é o Sol, ao contrário dos planetas. A escolha da orbita tem por base o cometa *Halley*.

É de notar que a ordem pela qual se aplica cada transformação é relevante. Primeiro é necessário preparar os eixos para se desenhar a figura no local pretendido, como por exemplo no caso do cometa. De seguida, aplica-se o `translate` associado à curva de *Catmull-Rom* e só depois é que se aplica o `rotate` relativo à rotação do corpo celeste sobre si próprio. Para finalizar, aplica-se a cor para assim a órbita (curva de *Catmull-Rom*) não ficar com a mesma cor que a do corpo celeste.

O XML deste cenário é o apresentado de seguida.

```
<scene>
    <group>
        <!--Sol-->
            <rotate time="1" x="0" y="1" z="0" />
```

```
        <color x="0.8" y="0.2" z="0.0" />
        <models>
            <model file="sol.3d" />
        </models>
</group>
<group>
    <!--Cometa Halley-->
        <rotate angle="-45" x="1" y="0" z="0" />
        <translate x="40" y="-35" z="-40" />
        <translate time="100">
         <point x="0" y="0" z="70" />
         <point x="49.497" y="0" z="49.497" />
         <point x="70" y="0" z="0" />
         <point x="49.497" y="0" z="-49.497" />
         <point x="0" y="0" z="-70" />
         <point x="-49.497" y="0" z="-49.497" />
         <point x="-70" y="0" z="0" />
         <point x="-49.497" y="0" z="49.497" />
        </translate>
        <translate x="0" y="-1" z="0" />
        <rotate angle="270" x="1" y="0" z="0"/>
        <color x="0.6" y="1.0" z="0.5" />
        <models>
            <model file="teapot.3d" />
        </models>
</group>
<group>
    <!--Mercurio-->
        <translate time="10" >
         <point x="0" y="0" z="35" />
         <point x="24.7487" y="0" z="24.7487" />
         <point x="35" y="0" z="0" />
         <point x="24.7487" y="0" z="-24.7487" />
         <point x="0" y="0" z="-35" />
         <point x="-24.7487" y="0" z="-24.7487" />
         <point x="-35" y="0" z="0" />
         <point x="-24.7487" y="0" z="24.7487" />
        </translate>
```

```
        <rotate time="1" x="0" y="1" z="0" />
        <color x="0.2" y="0.2" z="0.2" />
        <models>
            <model file="mercurio.3d" />
        </models>
</group>
<group>
    <!--Venus-->
        <translate time="20" >
         <point x="0" y="0" z="44" />
         <point x="31.112" y="0" z="31.112" />
         <point x="44" y="0" z="0" />
         <point x="31.112" y="0" z="-31.112" />
         <point x="0" y="0" z="-44" />
         <point x="-31.112" y="0" z="-31.112" />
         <point x="-44" y="0" z="0" />
         <point x="-31.112" y="0" z="31.112" />
        </translate>
        <rotate time="1" x="0" y="1" z="0" />
        <color x="0.8" y="0.5" z="0.0" />
        <models>
            <model file="venus.3d" />
        </models>
</group>
<group>
    <!--Terra-->
        <translate time="30" >
         <point x="0" y="0" z="52" />
         <point x="36.775" y="0" z="36.775" />
         <point x="52" y="0" z="0" />
         <point x="36.775" y="0" z="-36.775" />
         <point x="0" y="0" z="-52" />
         <point x="-36.775" y="0" z="-36.775" />
         <point x="-52" y="0" z="0" />
         <point x="-36.775" y="0" z="36.775" />
        </translate>
        <rotate time="1" x="0" y="1" z="0" />
        <color x="0.0" y="0.0" z="0.9" />
```

```
            <models>
                <model file="terra.3d" />
            </models>
        <group>
            <translate time="20" >
             <point x="0" y="0" z="6" />
             <point x="4.242" y="0" z="4.242" />
             <point x="6" y="0" z="0" />
             <point x="4.242" y="0" z="-4.242" />
             <point x="0" y="0" z="-6" />
             <point x="-4.242" y="0" z="-4.242" />
             <point x="-6" y="0" z="0" />
             <point x="-4.242" y="0" z="4.242" />
            </translate>
            <rotate time="1" x="0" y="1" z="0" />
            <color x="0.5" y="0.8" z="0.8" />
            <models>
                <model file="lua.3d" />
            </models>
        </group>
    </group>
    <group>
        <!--Marte-->
            <translate time="40" >
             <point x="0" y="0" z="70" />
             <point x="49.497" y="0" z="49.497" />
             <point x="70" y="0" z="0" />
             <point x="49.497" y="0" z="-49.497" />
             <point x="0" y="0" z="-70" />
             <point x="-49.497" y="0" z="-49.497" />
             <point x="-70" y="0" z="0" />
             <point x="-49.497" y="0" z="49.497" />
            </translate>
            <rotate time="1" x="0" y="1" z="0" />
            <color x="1.0" y="0.0" z="0.0" />
             <models>
                <model file="marte.3d" />
             </models>
```

```
</group>
<group>
    <!--Jupiter-->
        <translate time="50" >
         <point x="0" y="0" z="159" />
         <point x="112.43" y="0" z="112.43" />
         <point x="159" y="0" z="0" />
         <point x="112.43" y="0" z="-112.437" />
         <point x="0" y="0" z="-159" />
         <point x="-112.43" y="0" z="-112.43" />
         <point x="-159" y="0" z="0" />
         <point x="-112.43" y="0" z="112.43" />
        </translate>
        <rotate time="1" x="0" y="1" z="0" />
        <color x="0.8" y="0.5" z="0.2" />
        <models>
            <model file="jupiter.3d" />
        </models>
    <group>
        <translate time="15" >
         <point x="0" y="0" z="16.8" />
         <point x="11.879" y="0" z="11.879" />
         <point x="16.8" y="0" z="0" />
         <point x="11.879" y="0" z="-11.879" />
         <point x="0" y="0" z="-16.8" />
         <point x="-11.879" y="0" z="-11.879" />
         <point x="-16.8" y="0" z="0" />
         <point x="-11.879" y="0" z="11.879" />
        </translate>
        <rotate time="1" x="0" y="1" z="0" />
        <color x="0.5" y="0.8" z="0.8" />
        <models>
                <model file="io.3d" />
        </models>
    </group>
    <group>
        <translate time="20" >
         <point x="0" y="0" z="21.3" />
```

```
        <point x="15.061" y="0" z="15.061" />
        <point x="21.3" y="0" z="0" />
        <point x="15.061" y="0" z="-15.061" />
        <point x="0" y="0" z="-21.3" />
        <point x="-15.061" y="0" z="-15.061" />
        <point x="-21.3" y="0" z="0" />
        <point x="-15.061" y="0" z="15.061" />
    </translate>
    <rotate time="1" x="0" y="1" z="0" />
    <color x="0.5" y="0.8" z="0.8" />
    <models>
            <model file="europa.3d" />
    </models>
</group>
<group>
    <translate time="25" >
        <point x="0" y="0" z="25.3" />
        <point x="17.889" y="0" z="17.889" />
        <point x="25.3" y="0" z="0" />
        <point x="17.889" y="0" z="-17.889" />
        <point x="0" y="0" z="-25.3" />
        <point x="-17.8891" y="0" z="-17.889" />
        <point x="-25.3" y="0" z="0" />
        <point x="-17.889" y="0" z="17.889" />
    </translate>
    <color x="0.5" y="0.8" z="0.8" />
    <models>
            <model file="ganymede.3d" />
    </models>
</group>
<group>
    <translate time="30" >
        <point x="0" y="0" z="30.3" />
        <point x="21.425" y="0" z="21.425" />
        <point x="30.3" y="0" z="0" />
        <point x="21.425" y="0" z="-21.425" />
        <point x="0" y="0" z="-30.3" />
        <point x="-21.425" y="0" z="-21.425" />
```

```
            <point x="-30.3" y="0" z="0" />
            <point x="-21.425" y="0" z="21.425" />
        </translate>
        <rotate time="1" x="0" y="1" z="0" />
        <color x="0.5" y="0.8" z="0.8" />
        <models>
                <model file="callisto.3d" />
        </models>
    </group>
</group>
<group>
    <!--Saturno-->
        <translate time="60" >
            <point x="0" y="0" z="294" />
            <point x="207.889" y="0" z="207.889" />
            <point x="294" y="0" z="0" />
            <point x="207.889" y="0" z="-207.889" />
            <point x="0" y="0" z="-294" />
            <point x="-207.889" y="0" z="-207.889" />
            <point x="-294" y="0" z="0" />
            <point x="-207.889" y="0" z="207.889" />
        </translate>
        <rotate time="1" x="0" y="1" z="0" />
        <color x="0.8" y="0.6" z="0.4" />
        <models>
                <model file="saturno.3d" />
        </models>
    <group>
        <color x="0.8" y="0.6" z="0.0" />
        <rotate angle="-70" x="1" y="0" z="0" />
        <models>
                <model file="anel.3d" />
        </models>
    </group>
    <group>
        <rotate angle="-70" x="1" y="0" z="0" />
        <rotate angle="90" x="1" y="0" z="0" />
        <translate time="15" >
```

```
            <point x="0" y="3" z="16.8" />
            <point x="11.879" y="3" z="11.879" />
            <point x="16.8" y="3" z="0" />
            <point x="11.879" y="3" z="-11.879" />
            <point x="0" y="3" z="-16.8" />
            <point x="-11.879" y="3" z="-11.879" />
            <point x="-16.8" y="3" z="0" />
            <point x="-11.879" y="3" z="11.879" />
          </translate>
          <color x="0.5" y="0.8" z="0.8" />
          <rotate time="1" x="0" y="1" z="0" />
          <models>
                  <model file="titan.3d" />
          </models>
      </group>
  </group>
  <group>
      <!--Urano-->
          <translate time="80" >
            <point x="0" y="0" z="354" />
            <point x="250.316" y="0" z="250.316" />
            <point x="354" y="0" z="0" />
            <point x="250.316" y="0" z="-250.316" />
            <point x="0" y="0" z="-354" />
            <point x="-250.316" y="0" z="-250.316" />
            <point x="-354" y="0" z="0" />
            <point x="-250.316" y="0" z="250.316" />
          </translate>
          <rotate time="1" x="0" y="1" z="0" />
          <color x="0.5" y="0.5" z="1.0" />
          <models>
                  <model file="urano.3d" />
          </models>
  </group>
  <group>
      <!--Neptuno-->
          <translate time="90" >
            <point x="0" y="0" z="462" />
```

```
            <point x="326.683" y="0" z="326.683" />
            <point x="462" y="0" z="0" />
            <point x="326.683" y="0" z="-326.683" />
            <point x="0" y="0" z="-462" />
            <point x="-326.683" y="0" z="-326.683" />
            <point x="-462" y="0" z="0" />
            <point x="-326.683" y="0" z="326.683" />
        </translate>
        <rotate time="1" x="0" y="1" z="0" />
        <color x="0.2" y="0.2" z="1.0" />
        <models>
                <model file="neptuno.3d" />
        </models>
    <group>
        <translate time="20" >
            <point x="0" y="0" z="12" />
            <point x="8.485" y="0" z="8.485" />
            <point x="12" y="0" z="0" />
            <point x="8.485" y="0" z="-8.485" />
            <point x="0" y="0" z="-12" />
            <point x="-8.485"   y="0" z="-8.485" />
            <point x="-12" y="0" z="0" />
            <point x="-8.485" y="0" z="8.485" />
        </translate>
        <rotate time="1" x="0" y="1" z="0" />
        <color x="0.5" y="0.8" z="0.8" />
        <models>
                <model file="triton.3d" />
        </models>
    </group>
</group>
<group>
    <!--Plutão-->
        <translate time="100" >
            <point x="0" y="0" z="602" />
            <point x="425.678" y="0" z="425.678" />
            <point x="602" y="0" z="0" />
            <point x="425.678" y="0" z="-425.678" />
```

```
            <point x="0" y="0" z="-602" />
            <point x="-425.678" y="0" z="-425.678" />
            <point x="-602" y="0" z="0" />
            <point x="-425.678" y="0" z="425.678" />
          </translate>
          <rotate time="1" x="0" y="1" z="0" />
          <color x="0.7" y="0.7" z="0.7" />
          <models>
                  <model file="plutao.3d" />
          </models>
    </group>
</scene>
```

Desta forma, obteu-se um protótipo do Sistema Solar dinâmico com o seguinte aspeto:

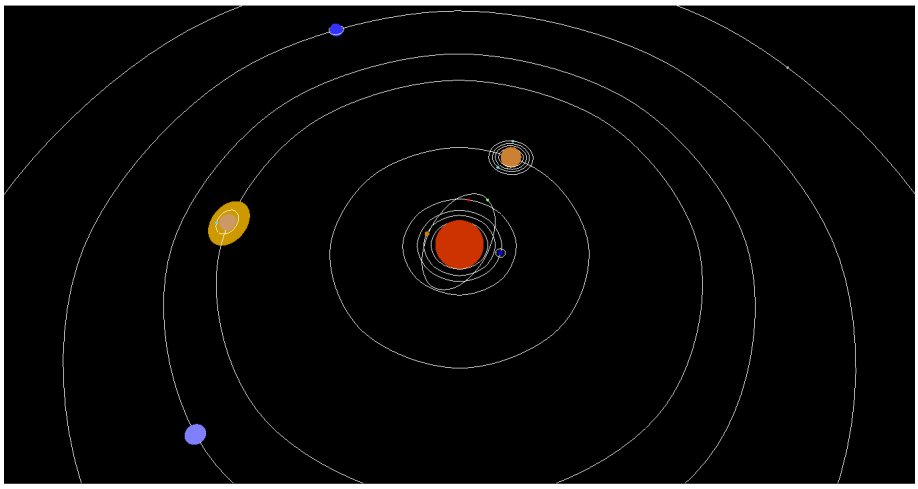

Figura 4: Sistema Solar

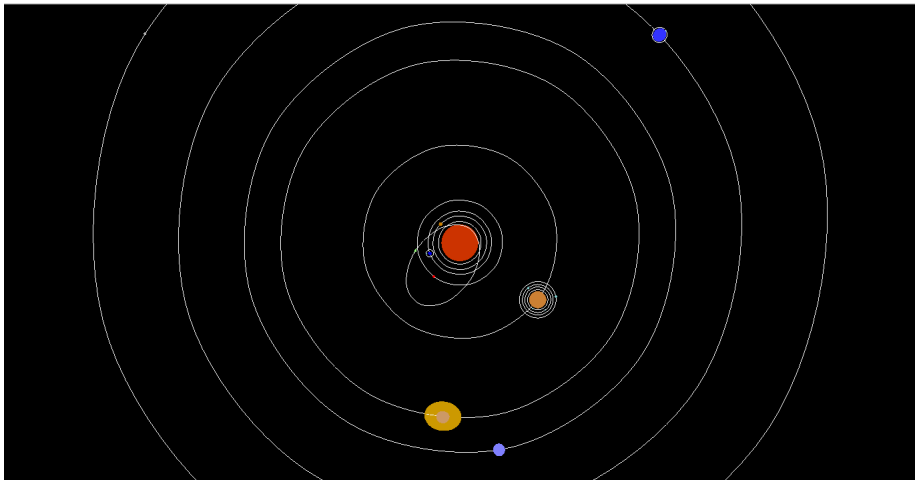

Figura 5: Sistema Solar - Uma outra perspectiva

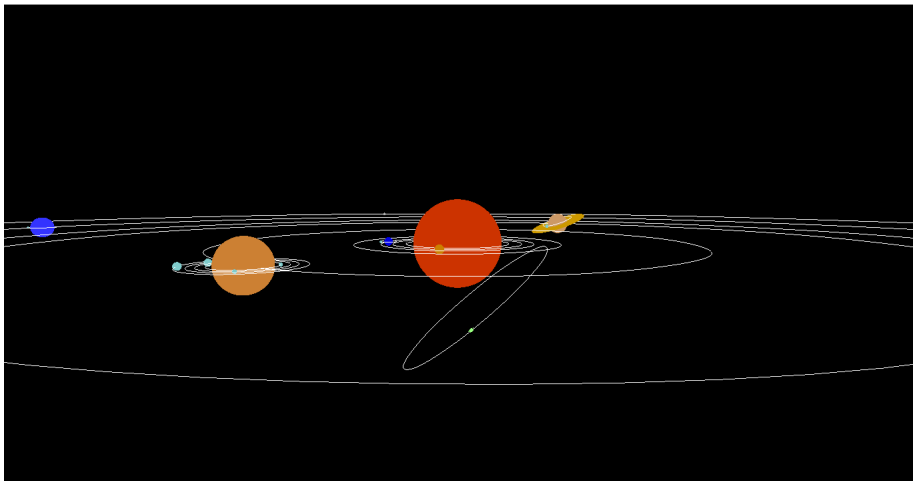

Figura 6: Sistema Solar - Mais uma perspectiva

## 4.4 Câmera

Tal como na fase anterior, continuou-se a aplicar a câmera no modo explorador, isto é, a câmara encontra-se direcionada para um ponto de referência, inicialmente o centro, e movimenta-se numa superficie esfêrica. Desta forma, é possível visualizar cada cenário criado com melhor exatidão.

Para o movimento da câmera utiliza-se o teclado, estando as teclas implementadas para cada funcionalidade descritas no relatório anterior.

# 5 Conclusões

Nesta terceira fase continuou-se a desenvolver o conhecimento de OpenGL, especialmente a manipulação de *VBOs*, aplicação de curvas de *Catmull-Rom* e utilização de *patches* de *Bezier*.

Mais uma vez, cumpriu-se todos os requisitos propostos, apesar de todas as dificuldades encontradas, principalmente na geração de pontos através de *patches* de *Bezier*. Com a utilzação de *VBOs* notou-se uma significativa melhoria no processamento do Sistema Solar dinâmico.

Para além daquilo que foi proposto, também foi explorado o Sistema Solar mais detalhadamente, isto é, atribuindo-lhe órbitas, desenhando as luas e anel dos respetivos planetas.

Nas restantes fases do projeto espera-se melhorar o projeto base até agora desenvolvido, de forma a melhorar a sua apresentação.