Computação Gráfica

Relatório

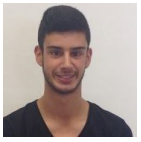
3º Fase - Curves, Cubic Surfaces and VBOs

Ana Paula Carvalho
a61855

Bruno Manuel Arieira
a70565

João Miguel Palmeira
a73864

Pedro Manuel Almeida
a74301



29 de Abril de 2018

Conteúdo

Lista de Figuras

1 Introdução

Foi-nos proposto, no âmbito da UC de **Computação Gráfica**, a criação de um mecanismo 3D baseado num cenário gráfico sendo que, para isso, teríamos de utilizar várias ferramentas apresentadas nas aulas práticas entre as quais C++ e *OpenGL*.

Este trabalho foi dividido em quatro partes, sendo esta a terceira fase que tem como objetivo a inclusão de curvas e superfícies cúbicas ao trabalho anteriormente desenvolvido, tendo como finalidade a criação de um modelo dinâmico do Sistema Solar com um cometa incluído, representado por um *teapot*.

1.1 Resumo

Tendo por base o trabalho realizando durante a primeira e segunda fases do projeto prático da Unidade Curricular de Computação Gráfica, adaptou-se e desenvolveu-se a implementação de modo a poder-se gerar um novo tipo de modelo baseado em *Bezier Patches*.

Esta fase traz consigo várias novidades que irão levar a diversas alterações, tanto ao nível do motor como do gerador.

Começando pelo gerador, este passará a conseguir ser capaz de criar um novo tipo de modelo baseado em *Bezier patches*. Desta forma, o gerador passará a poder receber como parâmetros o nome de um ficheiro, no qual se encontram definidos os pontos de controlo dos vários *patches*, assim como, um nível de tecelagem e, a partir destes, retornará um ficheiro contendo uma lista de triângulos que definem essa mesma superfície.

Passando para o motor, este sofrerá várias modificações e receberá também novas funcionalidades. Os elementos translate e rotate, presentes nos ficheiros XML, sofrerão algumas modificações. O elemento translation será agora acompanhado por um conjunto de pontos, que no seu todo, definirão uma *Catmull-Rom curve*, assim como o número de segundos para percorrer toda essa curva. Isto surge com o objetivo de passar a ser possível criar animações baseadas nessas mesmas curvas.

Passando para o elemento rotation, o ângulo anteriormente definido poderá agora ser substituído pelo tempo, correspondente este ao número de segundos que o objeto demora para completar uma rotação de 360 graus em torno do eixo definido. Deste modo, terá que ser alterado não só o *parser* responsável

por ler esses mesmos ficheiros, assim como, o modo como é processada toda a informação recebida com o intuito de gerar todo o cenário pretendido.

A última modificação que o motor irá sofrer, está relacionada com os modelos gráficos, uma vez que estes agora passarão a ser desenhados com o auxílio de VBOs, ao contrário do que acontecia na fase anterior, na qual estes eram desenhados de forma imediata.

Após realizar todas estas alterações esperamos alcançar um modelo do Sistema Solar mais realista do que aquele que geramos na fase anterior, visto que o trabalho passar de um modelo estático para um modelo dinâmico devido a implementação de todos estes novos requisitos.

2 Objetivos

Pretende-se, na presente fase de projeto, realizar uma evolução relativamente à fase antecedente que será viável e realizável através da criação da lista de triângulos correspondentes a superfícies de *Bezier*.

Ambiciona-se ainda que o software retorne uma lista dos triângulos gerados por intermédio de uma lista de vértices correspondentes aos pontos de controlo e ao grau de tesselação.

Nesta fase, irá continuar a incidir no modelo do Sistema Solar previamente desenvolvido nas fases anteriores onde se ambiciona o melhoramento do mesmo através de diferentes meios.

Por último pretende-se realizar a implementação de curvas como translações, rotações e a noção de tempo.

3 Contextualização

3.1 VBO's

Nesta fase é requisitado um melhoramento da eficiência do *software*, sendo este obtido a partir de *"vertex buffer object"* - basicamente é criado um *buffer* onde são guardados todos os pontos para desenhar um objeto.

Desta forma é necessário que se saiba quantos pontos são necessários para o processo de renderização de um objeto. A partir do momento em que se quantos pontos serão necessários, é indispensável abrir o *buffer* com o espaço corretamente alocado e introduzir toda a informação sobre os pontos.

De seguida, no *render* será invocado um conjunto de funções predefinidas que tratarão de o processar.

Para realizar estas alterações, foram realizadas várias modificações na função **GRload** na classe **estruturas.h**, que foi elaborada aquando da segunda fase, conseguindo que fosse possível a abertura da estrutura e a inserção da informação na mesma (neste caso, a inserção das coordenadas de cada um dos pontos).

```
verticeB = (float*) malloc(nPontos*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

O que é uma curva de Bezier?

Uma curva de Bézier é uma curva polinomial expressa como a interpolação linear entre alguns pontos representativos, chamados de pontos de controlo. Para contruir uma curva de Bezier são necessários, no mínimo, 3 pontos de controlo, podendo chegar a **n** pontos. No entanto, a sua forma mais amplamente utilizada é a de terceira ordem - a curva cúbica de Bezier - que é definida

por quatro pontos de controlo. Tais pontos são: 2 *endpoints* (também conhecidos como pontos âncoras) e dois *control points* (pontos de controlo) que não passam pela curva mas definem a sua forma. A linha que une um ponto âncora ao seu ponto de controlo corresponde à reta tangente à curva no ponto âncora e, por isso mesmo, é ela que determina o declive da curva neste ponto.

Esta forma de construção encontra-se ilustrada na figura seguinte.

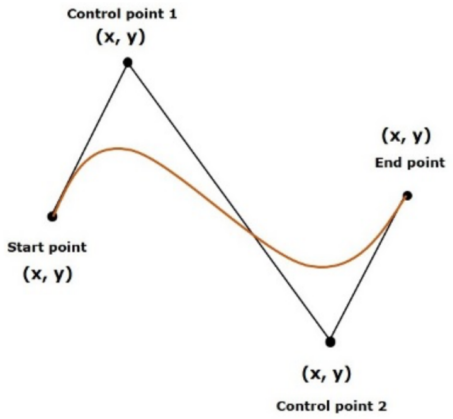


Figura 1: Exemplo de curva Bezier com 4 pontos

3.3 Superfícies de Bezier

Após um melhor entendimento de como se formam as curvas de Bezier foi possível concluir que também se podem formar superfícies utilizando o mesmo método, isto é, seguindo o mesmo processo embora utilizando a deslocação em dois eixos.

Tal como demonstrado, uma curva definida por pequenos segmentos de reta e é, portanto, seguro deduzir que quantos mais segmentos de reta uma curva tiver, mais definida esta se encontra.

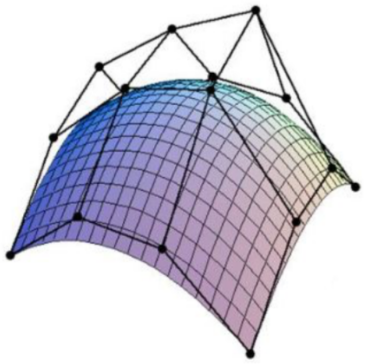


Figura 2: Exemplo de superficie Bezier

4 Estruturas

Na segunda fase do nosso projeto já se tinha definido algumas classes fundamentais que serviriam de base para a presente fase, tais como as designadas como *Ponto*, *Orbita*, *Cor*, *Modelo*, *Specs*, *Grupo* e *Figura*. No entanto, para esta terceira fase foi necessário acrescentar mais alguns dados às classes anteriores, como irá ser demonstrado de seguida.

Decidiu-se que a classe *Specs* definida na fase anterior - que representa os três tipos de transformações - fosse dividida, declarando as novas classes:

- Translação

  As primeiras três variáveis declaradas (coordx, coordy e coordz) servem de auxílio à translacão que se relaciona em mover os eixos para determinada posicão. Adicionamos a variável tempo que representa o tempo que um determinado objeto demora a executar uma translação.

```
class Translacao{
public:
    float coordx,coordy,coordz,tempo;
    list<Ponto> pontos;
};
```

- Rotação

  Neste tipo de transformação, é nos permitido movimentar um objeto com um determinado ângulo e direção. Acrescentou-se o float tempo que refere o tempo que um objeto demora a realizar uma rotação.

```
class Rotacao{
public:
    float angulo,eixox,eixoy,eixoz,tempo;
};
```

- Escala

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

```
    class Escala{
    public:
        float escala_x,escala_y,escala_z;
};
```

De seguida, apresenta-se o que foi modificado nas estruturas de dados da fase anterior:

- Orbita

  Esta classe representa os raios de uma órbita - que são necessários dois por se tratar de uma elipse - e a *nControl*, que menciona os pontos de controlo da curva de Bezier.

  ```
  class Orbita{
  public:
      float raioX,raioZ;
      int nControl;
  };
  ```

- Modelo

  Relativamente a esta estrutura adicionou-se a variável *buffers* que guarda inteiros do *OpenGL*.

  ```
  class Modelo{
  public:
      string ficheiro;
      int pontos;
      GLuint buffers[2];
  };
  ```

- Grupo

  Nesta classe, destacam-se as três estruturas de dados que foram divididas da classe *Specs* da fase última fase.

```
class Grupo{
public:
    string tipo; // planeta ou orbita
    Translacao translacao;
    Rotacao rotacao;
    Escala escala;
    Cor cor;
    Orbita orbita;
    list<Modelo> modelos;
    list<Grupo> grupos;
};
```

5 Gerador

5.1 Geração de figuras geométricas

Foram efetuadas alterações nas funções que geram as figuras geométricas elaboradas anteriormente. A cada uma destas funções foi adicionado uma linha que contém a informação para a utilização do modo *Vertex Buffer Object* (*VBO*), ou seja, contém o número que corresponde ao número de triângulos multiplicado pelo número de vértices que, por sua vez, é multiplicado pelo número de componentes dos vértices. Ou seja, este inteiro é gerado pela multiplicação de três fatores, sendo os dois últimos sempre iguais a 3.

$$nPontos = nT * nV * nCV$$

De seguida irão ser apresentadas as linhas de código que foram adicionadas a cada uma das funções que geram as figuras geométricas.

5.1.1 Plano

```
int nPontos = 2*2*3*3;
fprintf(f,"\%d\n",nPontos);
```

5.1.2 Caixa

```
int nPontos = 2*6*3*3;
fprintf(f,"\%d\n",nPontos);
```

5.1.3 Esfera

```
int pontos = 2*slices*stacks*3*3;
fprintf(f,"\%d\n",nPontos);
```

5.1.4 Cone

```
int pontos = slices*3*3 + 2*slices*stacks*3*3;
fprintf(f,"\%d\n",nPontos);
```

5.1.5 Anel

```
int pontos = 2*slices*3*3;
```

```
fprintf(f,"\%d\n",nPontos);
```

5.2 Patch

Nesta fase adicionou-se uma nova função designada por **drawPatch** que tem como objetivo ler os ficheiros *Patch*, transformar a informação que está contida nos ficheiros e guardar essa nova informação resultante no ficheiro que é passado como argumento da função (*savefilename*).

O passo intermédio consiste essencialmente na interpretação dos pontos de controlo e através do algoritmo Bezier (encontra os pontos finais através dos pontos de controlo) é realizado o cálculo de uma superfície.

5.2.1 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 índices;
* 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**, como já evidenciado acima, está fragmentada em duas partes essenciais: 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.

5.2.2 Parsing

O *parsing* é feito na primeira parte da função **drawPatch**. Inicialmente é aberto o ficheiro que é recebido como argumento na função e de seguida é recolhida a informação (consoante a ordem da estrutura que mencionamos na secção anterior) onde, em primeiro lugar, é guardado quer o número de *patches* quer o número de pontos composto por dois inteiros. Os índices de cada *patch* são guardados numa estrutura do tipo $vector < vector < int >>$, que tem como dimensões a multiplicação do número de *patches* com o número de índices, que neste caso são 16.

$$Dimensão = nP * nI$$

Além disso, cada ponto de controlo é guardado numa célula da estrutura, $vector < Ponto >$, 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 ---//
    fstream f;
    f.open(patchfilename,ios::in);
    int patches=0;
    int vertices=0;
    vector<vector<int> > indicesPatch;
    vector<Ponto> vertixs;
    if(f.is_open()){
        string line;
        if(getline(f,line)) sscanf(line.c_str(),"%d\n",&patches);
        for(int i=0; i<patches ;i++){
            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);
        Ponto p;
        p.x = x; p.y = y; p.z = z;
        vertixs.push_back(p);
    }
    f.close();
}
else {
    printf("Não foi possível abrir o ficheiro patch!\n"); exit(0);
    }
```

5.3 Cálculo de Pontos

Neste caso, após ser feito o *parsing*, é calculado, segundo o algoritmo de Bezier, o número de pontos que irão definir as quatro curvas de Bezier onde cada uma delas é constítuida por quatro vértices.

Após este cálculo ser efetuado, a informação gerada é guardada no ficheiro .3d, que foi fornecido para a escrita desta informação, seguindo a nova estrutura: em primeiro lugar é guardado a o número total de pontos elaborados e em segundo lugar 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/((float)tesselation-1);
float pontos = patches*(tesselation)*2*(tesselation)*3*3;
fstream g;
g.open(savefilename,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("Erro a abrir o ficheiro\n"); exit(0); }

}
```

6 Motor

6.1 XML

Como mencionado anteriormente, a classe designada *Spec* da estrutura de dados sofreu alterações, sendo que deu origem a *tags* diferentes no ficheiro *XML*. Assim separámos a *tag spec* em duas *tags* distintas:

- Relativamente á órbita, fragmentamos em rotação e translação;
- Em relação aos planetas e luas, dividimos em rotação e escala;

Com isto, houve a necessidade de acrescentar a *tag* ponto, à qual ainda se inseriu o atributo *tempo* às *tags* rotação e translação, e o atributo nControl à *tag* orbita, atributos estes que já evidenciados na estrutura de dados.

```
<ponto X=35.000000 Z=0.000000 />                <- tag Ponto
<translacao tempo=4.82 >                        <- tag Translação
<rotacao tempo=12.69 eixoY=1 />                 <- tag Rotação
<orbita raioX=45 raioZ=44.99 nControl=16 />     <- tag Orbita
```

6.2 Parsing

Como já citado anteriormente, foi adicionada a variável tempo às classes rotacao e translacao, tendo em atenção que este refere o tempo que demora uma rotação/translação completa. Na classe translação contém uma lista de pontos que refere os respetivos pontos de controlo numa curva cúbica, curva esta que contém uma lista de pontos de controlo associada á orbita que é declarada como nControl.

Por conseguinte, na função abaixo descrita, foram realizadas as alterações necessárias de a modo a implementar estas novas funcionalidades.

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

```
Translacao t;
TiXmlNode* tNode = gNode->FirstChild("translacao");
if(tNode){
    TiXmlElement* tElem = tNode->ToElement();
    const char *tTempo = tElem->Attribute("tempo");
    if(tTempo) t.tempo = atof(tTempo); else t.tempo = -1;

    if(t.tempo == -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("ponto");
    if(testNode){
        for(TiXmlElement* ptElem = tElem->FirstChild("ponto")->ToElement() ;
        ptElem; ptElem = ptElem->NextSiblingElement()){
            Ponto ponto;
            const char *tX = ptElem->Attribute("X"), *tY = ptElem->Attribute("Y"),
            *tZ = ptElem->Attribute("Z");
            if(tX) ponto.coordx = atof(tX); else ponto.coordx = 0;
            if(tY) ponto.coordy = atof(tY); else ponto.coordy = 0;
            if(tZ) ponto.coordz = atof(tZ); else ponto.coordz = 0;
            t.pontos.push_back(ponto);
        }
    }
    else t.tempo = -1;
    }
}
else { t.tempo = -1; t.coordx = 0; t.coordy = 0; t.coordz = 0; }
g.translacao = t;
//rotacao
Rotacao r;
TiXmlNode* rNode = gNode->FirstChild("rotacao");
if(rNode){
```

```
    TiXmlElement* rElem = rNode->ToElement();
    const char *angulo = rElem->Attribute("angulo") ,
    *eixoX = rElem->Attribute("eixoX"), *eixoY = rElem->Attribute("eixoY"),
    *eixoZ = rElem->Attribute("eixoZ"), *rTempo = rElem->Attribute("tempo");
    if(angulo) r.angulo = atof(angulo); else r.angulo = 0;
    if(eixoX) r.eixox = atof(eixoX); else r.eixox = 0;
    if(eixoY) r.eixoy = atof(eixoY); else r.eixoy = 0;
    if(eixoZ) r.eixoz = atof(eixoZ); else r.eixoz = 0;
    if(rTempo) r.tempo = atof(rTempo); else r.tempo = -1;
}
else { r.angulo = 0; r.eixox = 0; r.eixoy = 0; r.eixoz = 0; r.tempo = -1; }
g.rotacao = r;
//escala
Escala e;
TiXmlNode* eNode = gNode->FirstChild("escala");
if(eNode){
    TiXmlElement* eElem = eNode->ToElement();
    const char *eX = eElem->Attribute("X"), *eY = eElem->Attribute("Y"),
    *eZ = eElem->Attribute("Z");
    if(eX) e.escala_x = atof(eX); else e.escala_x = 1;
    if(eY) e.escala_y = atof(eY); else e.escala_y = 1;
    if(eZ) e.escala_z = atof(eZ); else e.escala_z = 1;
}
else { e.escala_x = 1; e.escala_y = 1; e.escala_z = 1; }
g.escala = e;

//cor
Cor c;
TiXmlNode* cNode = gNode->FirstChild("cor");
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.cor = c;

//orbita
Orbita o;
TiXmlNode* oNode = gNode->FirstChild("orbita");
if(oNode){
    TiXmlElement* oElem = oNode->ToElement();
    const char *raioX = oElem->Attribute("raioX"),
    *raioZ = oElem->Attribute("raioZ"), *nControl = oElem->Attribute("nControl");
    if(raioX) o.raioX = atof(raioX); else o.raioX = 0;
    if(raioZ) o.raioZ = atof(raioZ); else o.raioZ = 0;
    if(nControl) o.nControl = atoi(nControl); else o.nControl = 4;

    if(o.nControl<4) o.nControl = 4;
    g.tipo = "orbita";
}
else { o.raioX = 0; o.raioZ = 0; o.nControl = 4; g.tipo = "planeta"; }
g.orbita = o;

//modelos
TiXmlNode* mNode = gNode->FirstChild("modelos");
if(mNode){
    TiXmlElement* mElem = mNode->ToElement();
    TiXmlNode* testNode = mElem->FirstChild("modelo");
    if(testNode){

        for(TiXmlElement* modElem = mElem->FirstChild("modelo")->ToElement();
        modElem; modElem = modElem->NextSiblingElement()){
            Modelo m;
            const char *ficheiro = modElem->Attribute("ficheiro");
            if(ficheiro) m.ficheiro = ficheiro;
            fstream f;
            //f.open(m.ficheiro,ios::in);
            f.open(m.ficheiro.c_str());
            if(f.is_open()){
                string line;
```

```
                        int nPontos=0, j=0;
                        float *verticeB = NULL;

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

                        verticeB = (float*) malloc(nPontos*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.pontos*sizeof(float),verticeB,
                        GL_STATIC_DRAW);
                        free(verticeB);

                        f.close();
                    }

                    g.modelos.push_back(m);
                }
            }
        }
        //recursividade sobre grupos
        if(gNode) GRload(gNode,&g.grupos);
        grupos->push_back(g);
    }
}
```

6.3 Rendering

Nesta fase, era também pedido que as órbitas fossem desenhadas através das curvas *Catmull-Rom*, sendo que, estas curvas são definidas por um conjunto de pontos.

Como tal, primeiramente, calcula-se uma matriz (nControl*3), que divide a órbita em n segmentos iguais (n correspode à variável *nControl*). Optou-se por utilizar 16 pontos de controlo no Sistema Solar, decisão esta que possiblitou um grau de precisão que vai de acordo com as nossas expetativas.

Desta forma, os elementos da matriz são utilizados como pontos de controlo, e, cada curva, será desenhada com 100 segmentos, em consequencia de, no ciclo *for*, em modo *GL_LINE_LOOP*, a variável a ser incrementada 0.01, até o seu valor ser igual a 1.

```
void drawOrbit(float raioX, float raioZ, 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)*raioX;
        p[i][1] = 0;
        p[i][2] = sin(i*slice)*raioZ;
    }
    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] = (...)
        res[1] = (...)
        res[2] = (...)

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

    glEnd();
}
```

Para a implementação da nova forma de rotação, foi necessário adicionar uma variável *time* à classe **getRotation**. A nova variável indica o tempo, em segundos, necessários para uma rotação de 360 graus. Desta forma, para aplicar o movimento de rotação dos objetos, usamos as fórmulas:

$$angle = 360/(time * 1000)$$

$$intt = glutGet(GLUT\_ELAPSED\_TIME)$$

$$t * angle$$

Dado que o valor obtido pela função *glutGet(GLUT_ELAPSED_TIME)* é o tempo decorrido desde a execução do *glutInit*, este vai aumentar durante a execução do projecto.

Ao dividir o tempo multiplicado por 1000, obtem-se um valor decimal, que vai ser o coeficiente de 360 (graus) dando a amplitude de rotação que o objecto vai executar no momento. Desta forma, o valor de *angle* dá-nos o ângulo a aplicar a função *glRotatef*, fazendo o objecto rodar uma certa porção num instante de tempo.

Caso se aumente o valor do tempo (*time*) a rotação torna-se mais lenta e torna-se mais rápida caso contrário.

```
void getRotation(float time , float eixox , float eixoy , float eixoz){
float angle = 360/(time*1000);
int t = glutGet(GLUT_ELAPSED_TIME);
glRotatef(t*angle, eixox, eixoy, eixoz);
}
```

Para além da rotação, também foi necessário implementar uma nova forma de translação. Para tal, foi necessário adicionar uma variável time e um

array res[3] à classe *getTranslation*, à semelhança da função anterior. A variável *time* corresponde ao tempo dado no ficheiro .xml, que representa o tempo, em segundos, necessário para o objeto completar uma volta na sua trajetória. O *array* é necessário para colocar e alinhar o objeto com a curva (trajetória).

O valor de a é conseguido como o valor *angle* no método da rotação, isto é, utilizando as seguintes fórmulas:

$$tempo = glutGet(GLUT\_ELAPSED\_TIME)\%(int)(time * 1000)$$

$$a = 360/(time * 1000)$$

Da mesma forma que na rotação, limitamos o valor recebido pela função *glutGet(GLUT_ELAPSED_TIME)* e, por fim, obtemos o valor de a dividindo os 360 graus por time*1000, obtendo um coeficiente que corresponde a uma porção da translação na curva.

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

for(list<Ponto>::iterator itp = pontos.begin() ; itp!=pontos.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] = (...)
res[1] = (...)
    res[2] = (...)

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

De modo a ser possivel o uso de *VBOs* e a permitir as alterações enunciadas para esta terceira fase, foi necessário alterar a função *renderGrupo* de modo a implementar as animações/"movimentos".

```
    void renderGrupo(list<Grupo>::iterator g){
glPushMatrix();

    if(!strcmp(g->tipo.c_str(),"orbita")){
        glRotatef(g->rotacao.angulo, g->rotacao.eixox, g->rotacao.eixoy, g->rotacao.eixoz);
        glTranslatef(g->translacao.coordx, g->translacao.coordy, g->translacao.coordz);
    }
    else{
        if(g->translacao.tempo == -1) glTranslatef(g->translacao.coordx,
        g->translacao.coordy, g->translacao.coordz);
        else getTranslation(g->translacao.tempo,g->translacao.pontos);

        if(g->rotacao.tempo == -1) glRotatef(g->rotacao.angulo, g->rotacao.eixox,
        g->rotacao.eixoy, g->rotacao.eixoz);
        else getRotation(g->rotacao.tempo,g->rotacao.eixox,
        g->rotacao.eixoy,g->rotacao.eixoz);
    }

    glScalef(g->escala.escala_x, g->escala.escala_y, g->escala.escala_z);
    glColor3f(g->cor.r,g->cor.g,g->cor.b);
    drawOrbit(g->orbita.raioX,g->orbita.raioZ , g->orbita.nControl);
```

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

    }
    for(list<Grupo>::iterator itg = g->grupos.begin(); itg != g->grupos.end(); itg++)
        renderGrupo(itg);

glPopMatrix();}
```

7 Resultados

Depois de efetuadas todas as alterações necessárias na implementação de modo ao modelo gerado corresponder ao proposto na tarefa, anexa-se na presente secção, os resultados do trabalho desenvolvido - as perspectivas da representação do Sistema Solar.

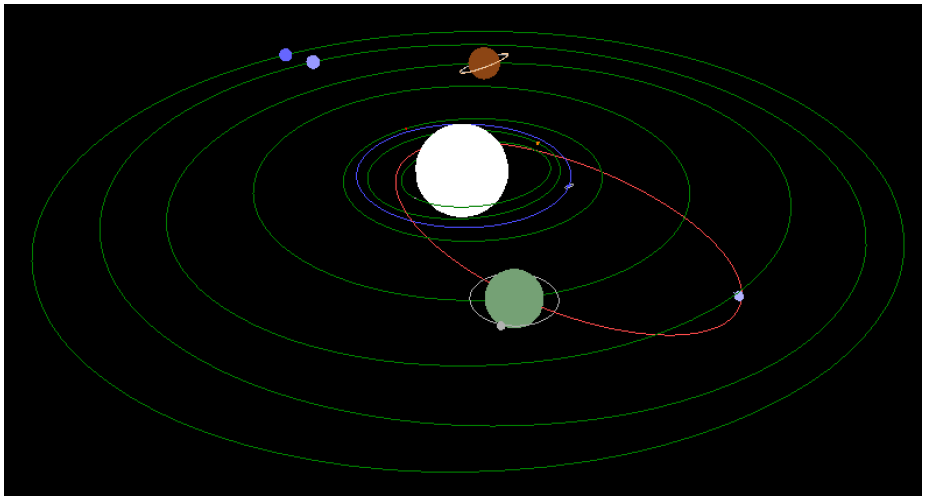


Figura 3: Perspectiva geral do modelo

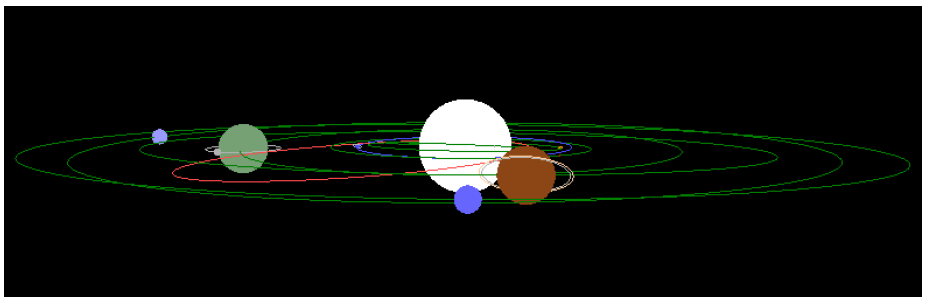


Figura 4: Vista lateral do modelo

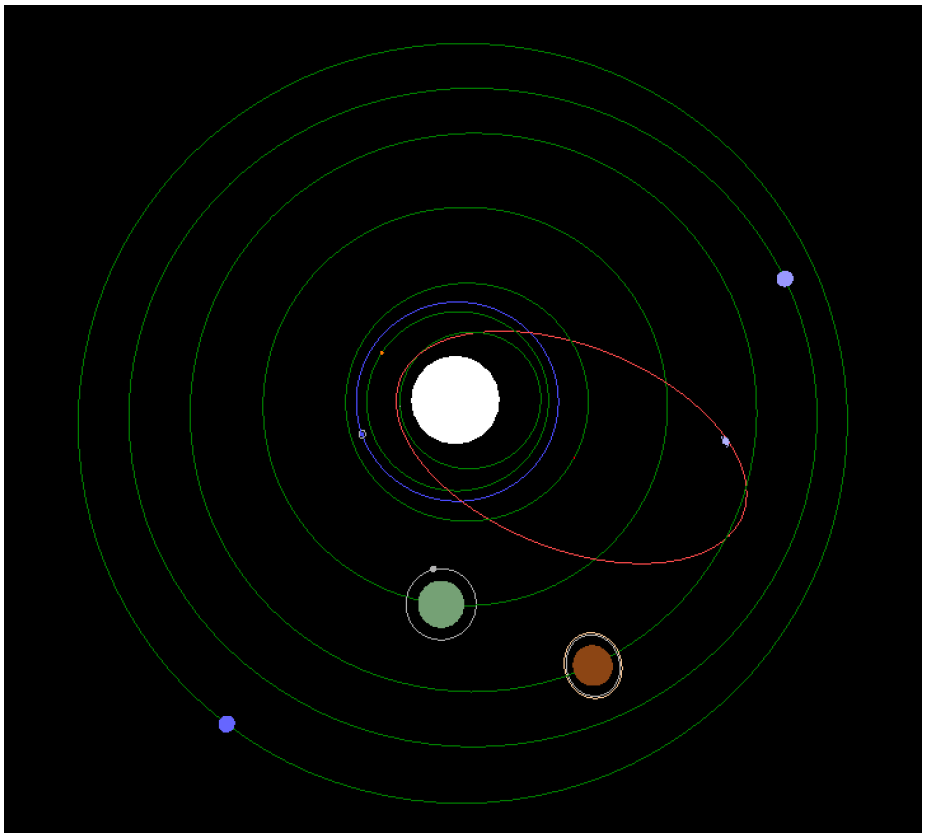


Figura 5: Vista superior do modelo

Como pode ser verificado nas imagens apresentadas, optou-se por distinguir a órbita da Terra das restantes pela cor - representada em azul. Foi também incorporado um cometa, gerado através da utilização do ficheiro teapot.patch que, tal como a Terra, apresenta uma órbita diferenciada das restantes pela cor vermelha.

As imagens a seguir apresentadas salientam o bule de chã, de modo demonstrar o nosso resultado final com as devidas alterações.

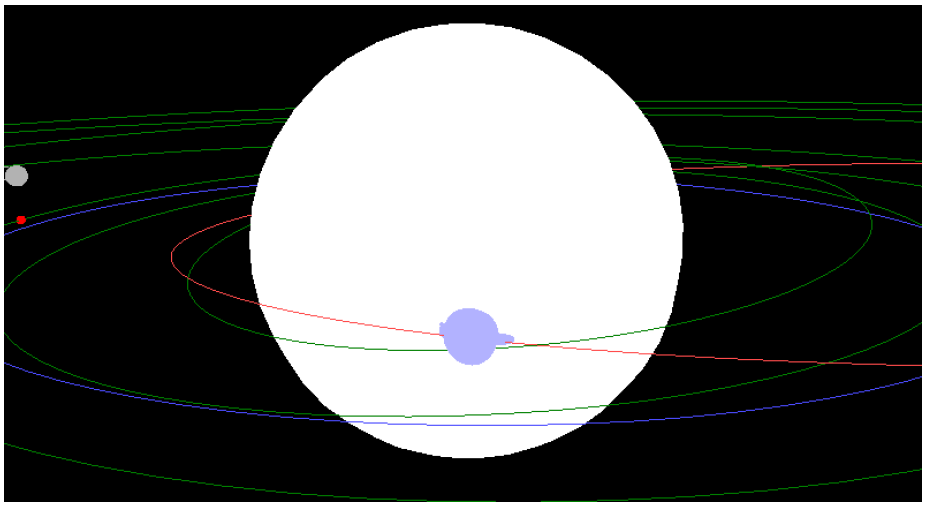


Figura 6: Vista diagonal do modelo

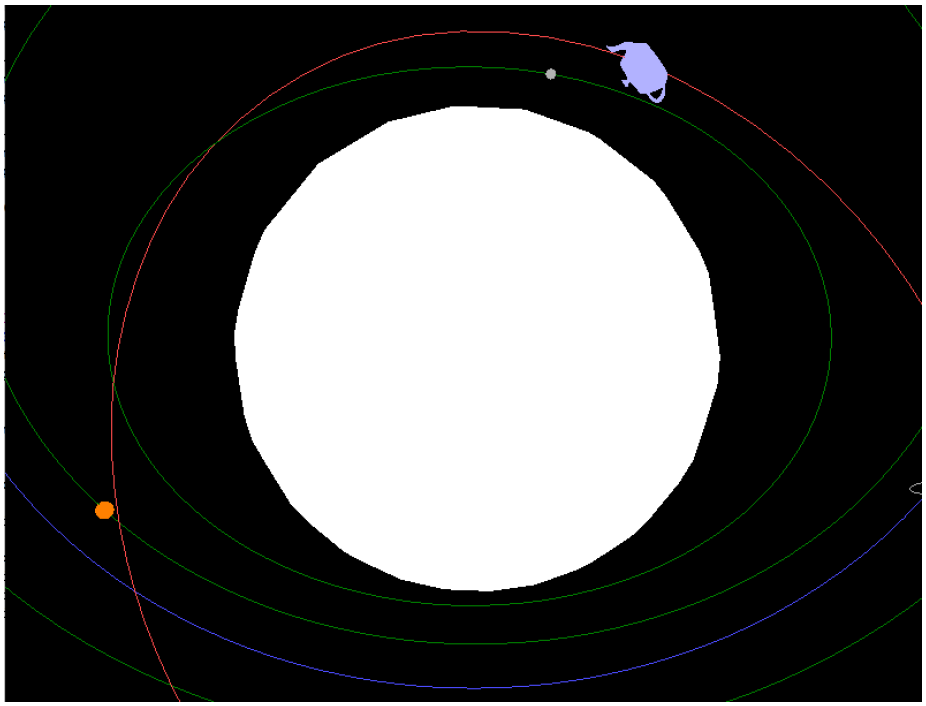


Figura 7: Vista superior do modelo

8 Conclusão

Dada por concluida esta terceira fase do projecto prático o grupo conseguiu, com sucesso, pôr à prova os mais recentes conhecimentos lecionados nas aulas da Unidade Curricular de Computação Gráfica aplicando-os de modo a satisfazerem o que era pedido na tarefa - implementaram-se curvas de Bézier, rotações e translações cíclicas dos constituintes do Sistema Solar e utilização de funcionalidades VBO.

Sentimos algumas dificuldades no decorrer desta fase o que causou uma maior demora na chegada ao resultado esperado. Entre essas dificuldades sentidas pelo grupo temos os Bezier patches e na melhor forma de projetar o modelo correspondente através dos mesmos. Foi necessário um maior cuidado na sua implementação mas tal dificuldade acabou por ser ultrapassada após algumas pesquizas e duvidas dissipadas. A implementação dos VBOs também mereceram atenção por parte do grupo mas, graças ao trabalho desenvolvido na UC, acabaram por ser resolvidas com maior facilidade.

Por fim, o grupo considera-se bem sucedido na realização e conclusão desta tarefa prática e admite que se revelou benéfica para um conhecimento mais cimentado e mais aprofundado nesta área de estudos. Pretendemos agora continuar na proxima fase atingindo com sucesso os requisitos da mesma.