Universidade do Minho

Licenciatura em Engenharia Informática

Computação Gráfica

Phase 3 - Curves, Cubic Surfaces and VBOs

Grupo 12

Ana Murta (A93284)

Ana Henriques (A93268)

Leonardo Freitas (A93281)

Rui Coelho (A58898)

maio, 2022

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

A presente fase do trabalho prático, realizado no âmbito da disciplina de Computação Gráfica, procura dar continuidade ao desenvolvimento da cena anteriormente construída para o Sistema Solar. O modelo construído integra primitivas desenvolvidas anteriormente e adiciona, agora, uma nova primitiva que possibilita a leitura de *patches*.

Este modelo do Sistema Solar, representado sob um ficheiro *XML*, integra agora componentes como a movimentação dos corpos celestes. Para tal, foram efetuadas algumas alterações às componentes já desenvolvidas, de modo a acomodar estas novas funcionalidades.

Ao longo do presente relatório, é dado a conhecer o trabalho desenvolvido, assim como as estratégias adotadas para o efeito.

Capítulo 2

Generator

Um dos objetivos propostos para a terceira fase de entrega do trabalho prático centrava-se na geração de superfícies de Bézier. A criação destas superfícies foi efetuada com recurso a um ficheiro que continha os dados necessários para o efeito. Para acomodar esta funcionalidade, surgiu a necessidade de estender as operações fornecidas pelo *generator*.

O desafio colocado consiste na geração de um bule de chá (*teapot*) a fim de este ser usado como um cometa, no modelo do sistema solar. A criação deste objeto é efetuada com recurso a um ficheiro, *teapot.patch* que apresenta: o conjunto de *pacthes* para gerar o *teapot* e os pontos de controlo necessários. Assim sendo, foi adicionada a primitiva *generatePatch* que permite processar o ficheiro e, para um determinado valor de tesselagem, calcular os pontos necessários para desenhar o bule de chá – armazenando o conjunto de pontos num ficheiro *.3d* que recebe também como argumento.

A leitura do ficheiro de *patch* é efetuada numa travessia, sendo armazenada a informação de cada linha na variável adequada para o efeito – sendo a primeira linha correspondente ao número total de *pacthes* presentes no ficheiro, seguindo-se os *pacthes*, o número total de pontos de controlo e, por fim, os pontos de controlo. Após a leitura completa do ficheiro, as diversas superfícies cúbicas, vulgo *patches*, são tratadas iterativamente.

De um modo geral, um *patch* é descrito por 16 pontos de controlo, que permitem, através de cálculo matricial, determinar os pontos de interesse para determinar os triângulos necessários para desenhar a superfície. A informação retirada do ficheiro indica, para uma dada superfície, os índices dos pontos de controlo necessários para desenhar a superfície. Em suma, um *patch* possui 16 pontos, que representam 4 curvas de Bézier, cada uma composta por 4 pontos de controlo.

A função *triangulacao* encontra-se encarregue de calcular as coordenadas dos pontos da superfície e recebe como argumentos o conjunto dos pontos de controlo relevantes para o cálculo e o valor de tesselação. A fórmula usada para determinar os pontos encontra-se presente na Figura 2.1.

$$P(u,v) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \times M \times \begin{bmatrix} P_{00} & P_{10} & P_{20} & P_{30} \\ P_{01} & P_{11} & P_{21} & P_{31} \\ P_{02} & P_{12} & P_{22} & P_{32} \\ P_{03} & P_{13} & P_{23} & P_{33} \end{bmatrix} \times M^T \times \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix}$$

Figura 2.1: Cálculo dos pontos para a superfície de Bézier.

O cálculo das coordenadas dos pontos é elaborado em dois "níveis": por um lado, é efetuada uma pré-computação da multiplicação das matrizes que estas são constantes para o cálculo de todos os pontos de uma dada superfície[1]; por outro lado, executa-se o cálculo iterativo dos pontos, com base nos valores das variáveis v e u – que variam entre 0 e 1, sendo incrementadas de acordo com o fator apresentado na Figura 2.2.

$$x_i = x_{i-1} + \frac{1}{tesselação}, i \in [0, tesselagem]$$

Figura 2.2: Incremento das variáveis u e v a cada iteração.

Para a computação do cálculo matricial são usadas as seguintes funções auxiliares:

- *multMatrixPoints*: permite multiplicar a matriz M^2 pela matriz de pontos do *patch*, gerando uma matriz de pontos auxiliar *aux*;
- *multPointsMatrix*: permite multiplicar a matriz *aux* pela matriz M^{T} [3]
- *multVectorMatrix*: permite multiplicar um vetor por uma matriz[4], gerando um vetor *res*.
- *multVectors*: permite multiplicar dois vetores[5].

Os pontos calculados configuram uma grelha de pontos, que é usada para determinar os triângulos necessários para a reprodução gráfica da imagem. A Figura 2.3 demonstra um quadrado da grelha de pontos. Em cada quadrado da grelha podem ser definidos dois triângulos, tal como pode ser observado. Respeitando a regra da mão direita, e tendo em conta o exemplo gráfico apresentado na Figura 2.3, os pontos serão armazenados de acordo com a seguinte ordem: P_1, P_3 e P_2 (para formarem o triângulo superior) e P_3, P_4 e P_2 (para o triângulo inferior).

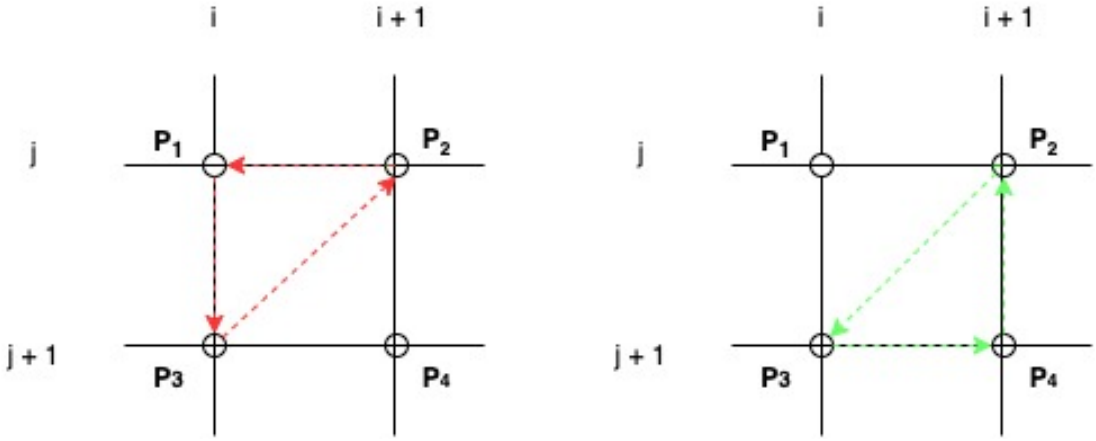

Figura 2.3: Racional de triangulação dos pontos.

[1] M * Patch * M^T
[2] Matriz de Bézier
[3] Transposta da matriz de Bézier: uma vez que a matriz de Bézier é simétrica, tem-se que $M = M^T$
[4] sendo usada para multiplicar o vetor $\begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix}$ pela matriz pré-calculada
[5] Usada para calcular o produto de *res* pelo vetor $\begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix}$

A Figura 2.4 apresenta a representação gráfica obtida com o módulo *ENGINE* e os pontos anteriormente calculados.

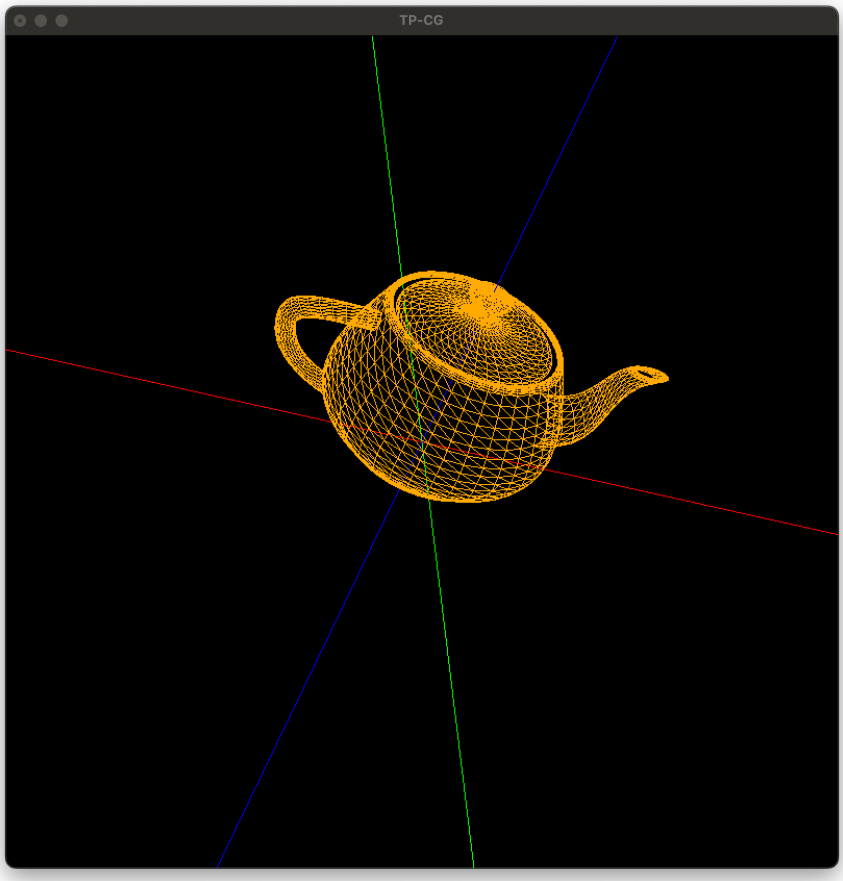

Figura 2.4: Representação do *teapot*.

Capítulo 3

Engine

Para conseguir cumprir o objetivo estabelecido para esta fase, efetuaram-se algumas alterações na leitura do ficheiro XML, permitindo translações e rotações com tempo, e no desenho das primitivas, que agora passarão a ser desenhadas com recurso a VBOs.

3.1 Leitura do ficheiro XML

As atualizações previamente mencionadas à função *parseGroup* resumem-se, agora, à capacidade de ler translações e rotações com tempo e sem tempo.

Na figura 3.1, temos a representação de um grupo destinado ao planeta Mercúrio, no qual podemos observar uma translação com tempo – <`translate time="17"align="True"`> e, ainda, uma rotação também com tempo – <`rotate time="10"x="0"y="1"z="0"`>. De facto, se um grupo contiver um *translate* com o atributo *time*, os pontos dentro de *translate* serão guardados num `vector<Point>`, variável adicionada à classe *Group* que guarda estes valores, para depois desenharem a curva de *CatMullRom*. Estes pontos foram calculados através de um *script* em *Python*, ilustrado na Figura 3.2. Para além disto, foram acrescentadas duas variáveis, *time* e *align*, à classe `trans`, que permitem armazenar os valores para as transformações. No que toca ao *rotate*, este pode conter o atributo *time*, e nesse caso a variável *angle* toma o valor de 0, ou, então, o atributo *angle* e, nesse caso a variável *time* toma o valor de 0.

```
<!-- Mercúrio -->
<group>
    <transform>
        <translate time="17" align="True">
            <point x="1.3" y="0" z="0" />
            <point x="0.9192388155425119" y="0" z="0.9192388155425117" />
            <point x="0" y="0" z="1.3" />
            <point x="-0.9192388155425117" y="0" z="0.9192388155425119" />
            <point x="-1.3" y="0" z="0" />
            <point x="-0.9192388155425512" y="0" z="-0.9192388155425117" />
            <point x="0" y="0" z="-1.3" />
            <point x="0.9192388155425116" y="0" z="-0.9192388155425512" />
        </translate>
        <rotate time="10" x="0" y="1" z="0"/>
        <scale x="0.05" y="0.05" z="0.05"/>
        <color x="238" y="154" z="73"/>
    </transform>
    <models>
        <model file="sphere.3d"/>
    </models >
</group>
```

Figura 3.1: Ficheiro *solarSystem.xml*

```
from cmath import cos, sin
import math
import xml.etree.cElementTree as Tree
world = Tree.Element('world')

raio = 5.5

def calculate_circle(raio):

    points = []

    nrPoints = 8

    i = 0
    for i in range(nrPoints):
        angle = math.pi / 4
        point = (math.cos(angle*i)*raio,math.sin(angle*i)*raio)
        points.append(point)

    return points

points = calculate_circle(raio)

for p in points:
    point = Tree.SubElement(world, "point")
    point.attrib["x"]=str(p[0].real)
    point.attrib["y"]="0"
    point.attrib["z"]=str(p[1].real)

tree = Tree.ElementTree(world)

tree.write(f'curvas{raio}.xml')
```

Figura 3.2: *Script* em *Python*

3.2 Desenho do ficheiro XML

Primeiramente, criou-se o ficheiro *CatmullRom* onde se encontram todas as funções necessárias para escrever as transformações associadas às curvas de *CatmullRoom*. Deste modo, consegue-se tornar o código mais legível com uma melhor organização ao retirar o código respetivo às curvas do ficheiro principal do projeto.

Uma vez que, tanto as translações, como as rotações, podem acontecer com tempo ou sem tempo, teve-se o cuidado de distinguir essas duas situações ao alterar a função *drawPrimitives*:

- **sem tempo**, a função desenha as primitivas exatamente da mesma forma como desenhou nas fases anteriores, não revelando qualquer diferença;
- **com tempo**, a função desenha as curvas de *CatmullRom* e, também, constrói as primitivas com base nas regras de rotação e translação implementadas.

```
void drawPrimitives(Group g) {
    (...)
    for (int j = 0; j < g.getNrTrans(); j++) {
        Trans t = g.getTrans(j);
        if (translate.compare(t.getName()) == 0) {
            if (t.getTime() != 0) {
                catmullPoints = g.getCatmullPoints();
                glPushMatrix();
                float p[3], d[3];
                float t1 = 100.0f;
                glBegin(GL_LINE_LOOP);
                for (int i = 0; i < t1; i += 1) {
                    CatmullRom::getGlobalCatmullRomPoint(catmullPoints, i / t1, p, d);
                    glVertex3fv(p);
                }
                glEnd();
                glPopMatrix();
                float pos[3];
                float deriv[3];
                float timeT = -(glutGet(GLUT_ELAPSED_TIME) / 1000.f) / (float)(t.getTime());
                CatmullRom::getGlobalCatmullRomPoint(g.getCatmullPoints(), timeT, (float*)pos,
                                                     (float*)deriv);
                glTranslatef(pos[0], pos[1], pos[2]);
                if (isAlign.compare(t.getAlign()) == 0) {
                    float m[4][4];
                    float x[3], z[3];
                    CatmullRom::cross(deriv, prev_y, z);
                    CatmullRom::cross(z, deriv, prev_y);
                    CatmullRom::normalize(deriv);
                    CatmullRom::normalize(prev_y);
                    CatmullRom::normalize(z);
                    CatmullRom::buildRotMatrix(deriv, prev_y, z, *m);
                    glMultMatrixf(*m);
                }
            }
            else if (t.getTime() == 0) {
                glTranslatef(t.getX(), t.getY(), t.getZ());
```

```
            }
        }
        else if (rotate.compare(t.getName()) == 0) {
            if (t.getAngle() == 0 && t.getTime() != 0) {
                float time = glutGet(GLUT_ELAPSED_TIME) / 1000.f;
                float angle = t.getAngle() + time * (t.getTime() ? 360.f / t.getTime() : 0);
                glRotatef(angle, t.getX(), t.getY(), t.getZ());
            }
            else if (t.getAngle() != 0 && t.getTime() == 0) {
                glRotatef(t.getAngle(), t.getX(), t.getY(), t.getZ());
            }
        }
        else if (color.compare(t.getName()) == 0){
            (...)
        }
        else if (scale.compare(t.getName()) == 0) {
            (...)
        }
    }
    for (int z = 0; z < g.getNrPrimitives(); z++) {
        Primitive p = g.getPrimitives(z);
        int nrVertices = p.getNrVertices();

        if (g.getNameFile().compare("asteroids.3d") == 0) {
            (...)
        }
        else {
            glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);
            glVertexPointer(3, GL_FLOAT, 0, 0);
            glDrawArrays(GL_TRIANGLES, vboZone, nrVertices);

            vboZone = vboZone + nrVertices;
        }
    }
    for (int z = 0; z < g.getNrGroups(); z++) {
        (...)
    }
}
```

3.3 Implementação VBOs

Um dos outros requisitos impostos para esta terceira fase foi atualizar o desenho das primitivas anteriormente feito nas outras fases com o uso de VBOs, tornando o programa mais eficiente.

Deste modo, foram criadas as variáveis *vertexB* (`vector<float>`) e *vboZone* (`GLuint`). A variável *vboZone* verifica, em cada momento, qual a zona do VBO a desenhar, funcionando, por isso, como uma variável de controlo já que a leitura de todas as primitivas é efetuada para um único VBO. Já o vetor *vertexB* é ultilizado para armazenar as coordenadas de cada ponto do ficheiro .3d, sendo preenchido pela função *readFile*, que faz o push_back de cada coordenada para o dito *vector*. A função *initGlut*, recorrendo às funções *glGenBuffers*, *glBindBuffer* e *glBufferData*, encarrega-se de formar o VBO de *vertexB*, que será copiado para a memória gráfica.

Posteriormente, para construir o VBO e o seu conteúdo, recorreu-se às seguintes funções:

- *glBindBuffer*, que é usada identificar o VBO ativo;
- *glVertexPointer*, que é usada para definir um vértice com 3 *floats*;
- *glDrawArrays*, que é usada para desenhar os triângulos que constituem o VBO a partir do modo GL_TRIANGLES.

Com a variável *vboZone* e com as primitivas guardadas nas estruturas de dados, conhecemos a zona do VBO que será desenhada, tal como foi dito anteriormente.

3.4 Transformações com tempo

Assim como supramencionado, no ficheiro XML, um grupo pode conter translação e rotação com ou sem tempo. A função responsável por determinar se uma transformação acontece com tempo ou sem tempo é a *gluGet(GLUT_ELAPSED_TIME)*.

3.4.1 Rotação

Com o intuito de implementar as rotações com tempo, foi necessário modificar o código da função *drawPrimitives*, desenvolvido na segunda fase deste projeto, e a criação de uma nova variável chamada *time*, como já mencionado anteriormente. Esta variável guarda o valor do atributo *time*, associado ao *rotate*, lido do ficheiro XML.

Durante a *parseGroup*, que faz o *parse* dos <group>(...)</group> do ficheiro XML, são armazenados valores nas variáveis *time* e *angle* da classe `Trans`, que dependem do facto da transformação acontecer com ou sem tempo. De facto, são estas variáveis que nos permitem implementar ambas as transformações através de condições, i.e. na função *drawPrimitive*, são tratadas as transformações com e sem tempo através do uso de condições.

```
else if (rotate.compare(t.getName()) == 0) {
            if (t.getAngle() == 0 && t.getTime() != 0) {
                float time = glutGet(GLUT_ELAPSED_TIME) / 1000.f;
                float angle = t.getAngle() + time * (t.getTime()
                              ? 360.f / t.getTime() : 0);
                glRotatef(angle, t.getX(), t.getY(), t.getZ());
            }
            else if (t.getAngle() != 0 && t.getTime() == 0) {
                glRotatef(t.getAngle(), t.getX(), t.getY(), t.getZ());
            }
        }
```

Como podemos observar, quando temos uma rotação com o tempo, são utilizadas as fórmulas para atualizar o ângulo:

```
float time = glutGet(GLUT_ELAPSED_TIME) / 1000.f

float angle = t.getAngle() + time * (t.getTime() ?  360.f / t.getTime() :  0)
```

Assim sendo, conquista-se, assim, um dos objetivos iniciais desta fase, nomeadamente traçar os movimentos de rotação em torno de si para os planetas.

3.4.2 Translação

A utilização de primitivas de translação com tempo recorreu às curvas de Catmull-Rom para delinear as trajetórias dos corpos celestes. Cada corpo possui um tempo definido que é usado para determinar a sua movimentação em torno do sol.

Com vista a organizar de uma forma metódica a componente *ENGINE*, foi criado um módulo referente ao trabalho desenvolvido com estas curvas. As funções presentes no ficheiro *Catmull-Rom.cpp* permitem implementar a movimentação dos astros representados no modelo do sistema solar, uma vez que contem um conjunto de funções que permite operar sobre os pontos e vetores de interesse para o efeito[1].

Para determinar as curvas de Calmull-Rom são utilizados os pontos que se encontram dentro da *tag* de *translate* no ficheiro *XML* que descreve o modelo do sistema solar. Tal como foi anteriormente mencionado, os pontos foram gerados com recurso a um *scrpit* escrito em *Python* – cf. 3.2.

O excerto de código apresentado representa uma porção da função *drawPrimitives*, presente no módulo *main.cpp* do *ENGINE* – em particular, focando-se no desenho de transformações de *translate* com tempo associado. Aquando do desenho de uma primitiva é, inicialmente, averiguado se esta possui uma variável de tempo associada – indicando assim a noção de movimento sobre uma curva. Quando tal se verifica, é determinada a curva que descreve o movimento de translação do astro.

```
if (translate.compare(t.getName()) == 0) {
    if (t.getTime() != 0) {
        (...)
        glBegin(GL_LINE_LOOP);
        for (int i = 0; i < t1; i += 1) {
            CatmullRom::getGlobalCatmullRomPoint(catmullPoints, i / t1, p, d);
            glVertex3fv(p);
        }
        glEnd();
        glPopMatrix();
        (...)
        float timeT = -(glutGet(GLUT_ELAPSED_TIME) / 1000.f) / (float)(t.getTime());
        CatmullRom::getGlobalCatmullRomPoint(g.getCatmullPoints(), timeT,
            (float*)pos,(float*)deriv);
        glTranslatef(pos[0], pos[1], pos[2]);
        (...)
    }
    (...)
}
```

A função *getGlobalCatmullRomPoint* é usada para determinar os pontos e derivadas, a partir do valor de tempo ($timeT$) – permitindo que a posição do corpo celeste se vá atualizando. A atualização do tempo é efetuada com recurso à seguinte fórmula:

[1] Como o caso de funções para efetuar a normalização de vetores, determinar a matriz de rotação e o cálculo dos pontos usados para determinar as curvas

```
loat timeT = -(glutGet(GLUT_ELAPSED_TIME) / 1000.f) / (float)(t.getTime())
```

Uma *tag* adicional que se pode encontrar dentro de uma transformação *translate* refere-se com o alinhamento do objeto com a direção do movimento. O excerto de código abaixo apresentado demonstra os cálculos matriciais conduzidos para que tal seja executado[2].

```
if (translate.compare(t.getName()) == 0) {
    if (t.getTime() != 0) {
        (...)
        if (isAlign.compare(t.getAlign()) == 0) {
            (...)
            CatmullRom::cross(deriv, prev_y, z);
            CatmullRom::cross(z, deriv, prev_y);
            CatmullRom::normalize(deriv);
            CatmullRom::normalize(prev_y);
            CatmullRom::normalize(z);
            CatmullRom::buildRotMatrix(deriv, prev_y, z, *m);
            glMultMatrixf(*m);
        }
    }
```

A Figura 3.3 apresenta as curvas de Catmull-Rom definidas para o movimento de translação dos planetas, e restantes corpos celestes, em torno do Sol[3].

[2]A sequência de cálculos executados assenta nos princípios teóricos discutidos nas aulas, pelo que envolvem operações como o produto vetorial e a normalização de vetores

[3]Nesta fase do trabalho as curvas que designam as trajetórias de translação dos planetas foram propositadamente desenhadas, com o objetivo de confirmar a correta movimentação e alinhamento dos corpos celestes.

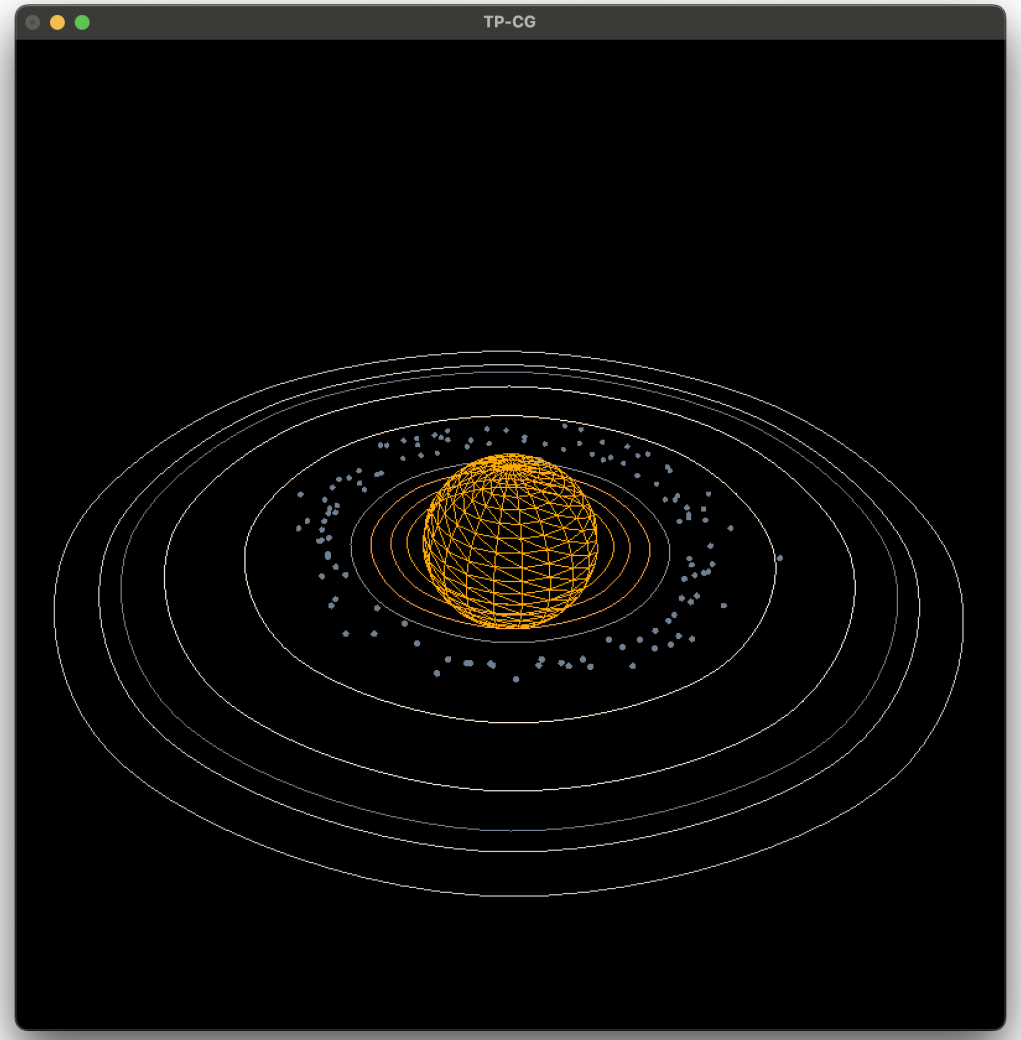

Figura 3.3: Curvas de *CatmullRom*.

3.5 Modelo do Sistema Solar

A Figura 3.4 apresenta o estado atual do Sistema Solar, tendo sido capturada aquando do movimento de rotação e translação dos corpos celestes.

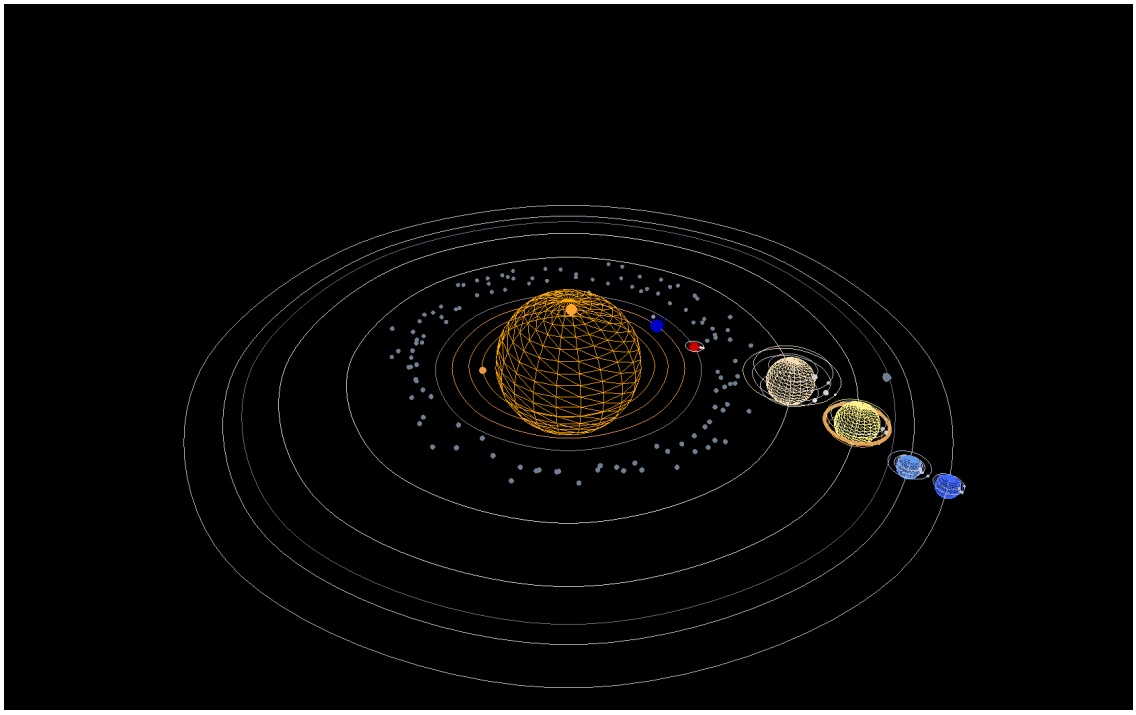

Figura 3.4: Modelo do Sistema Solar atual.

Capítulo 4

Conclusão

A terceira fase do trabalho prático procurou melhorar o modelo do sistema solar desenvolvido até ao momento, introduzindo novas funcionalidades e melhorando algumas das funcionalidades previamente desenvolvidas. No que diz respeito a novas funcionalidades, em particular, esta fase centrou-se na determinação das curvas que descrevem a movimentação dos corpos celestes, no processamento e desenho de um novo tipo de superfície e na implementação de VBOs.

De um modo geral, os objetivos estipulados foram alcançados com sucesso. O grupo foi capaz de introduzir movimento no sistema solar já criado, através da alteração das componentes *XML* e *ENGINE*. Este movimento consiste na translação dos astros em torno do sol e na rotação dos corpos celestes sobre si mesmos. Face ao modelo do sistema solar previamente desenvolvido, foi ainda adicionado o cometa, com recurso ao desenho de um *teapot* com os pontos gerados pelo módulo *GENERATOR*.

A determinação das superfícies de bézier foi também um dos objetivos estipulados para a corrente fase. O grupo foi capaz de implementar estas superfícies, apesar de se terem sentido algumas dificuldades no decurso do processo. A leitura do ficheiro de *patch* revelou-se um trabalho simples e bastante *straightforward* – algo que o grupo considerou que não aconteceu com o processamento da informação contida no ficheiro. O trabalho sobre estes dados revelou-se um pouco moroso, pois envolveu a criação de funções auxiliares para o processamento dos dados e a pré-computação de cálculo matricial. Em parte esta dificuldade sentida surgiu da falta de familiaridade de ferramentas da linguagem *C++* que pudessem auxiliar na tarefa. Uma vez enfrentados estes desafios, a aplicação dos conhecimentos teóricos revelou-se uma tarefa mais natural.

Posto isto, o grupo sente que ainda há espaço para melhorias nas componentes já desenvolvidas. Em particular, o grupo gostaria de afinar a computação das rotas de translação dos planetas e os tempos de translação e rotação dos corpos celestes. Adicionalmente, o grupo pretende diferenciar o nível de detalhe com que os diversos corpos celestes se encontram a ser desenhados, isto é, de momento os pontos usados para desenhar, por exemplo, o Sol e a Lua são exatemente os mesmos, o que parece ser pouco adequado, tendo em conta a proporção do tamanho da Lua em relação ao Sol. Por fim, pretende-se que, futuramente, a geração da cintura de asteroides seja feito com recurso a VBOs – algo que não foi implementado até à corrente fase.