Universidade do Minho
Escola de Engenharia

Computação Gráfica

Licenciatura em Engenharia Informática

Grupo 41

2022/2023

3ª Fase

Alexandra Santos (A94523)
Inês Ferreira (A97040)
José Rafael Ferreira (A97642)
Marta Sá (A97158)

ÍNDICE

INTRODUÇÃO

Esta 3º fase do trabalho prático teve como objetivo a implementação de um Sistema Solar mais eficiente e dinâmico. Assim, foi necessário realizar alterações no *engine* e no *generator*.

O *generator* passa a ser capaz de gerar novos modelos .3d baseados em superfícies de *Bezier* através de um nível de tesselação e de pontos de controle definidos num ficheiro *.patch*.

Quanto ao *engine*, implementou-se funcionalidades que permitem criar animações manipulando os elementos *translate* e *rotate* das transformações. Por um lado, o *translate* passa a receber pontos de controle a serem utilizados para a construção de curvas de *Catmull-Rom*, um parâmetro *time* que define o tempo necessário para percorrer a curva e um parâmetro *align* que define se o objeto está ou não alinhado com a curva. Por outro lado, o *rotate* passa a ser a capaz de receber um parâmetro *time* que especifica o tempo para realizar uma rotação de 360º em torno de um eixo específico.

Para além disso, o *engine* começa a utilizar *VBOs* para desenhar as figuras de forma a aumentar a performance do programa.

GENERATOR

No programa *generator*, de modo a ser possível implementar as superfícies de *Bezier* para desenhar os triângulos a serem desenhados num ficheiro .3d, foi necessário utilizar o seguinte comando:

patch patchFile level saveFile

No referido comando, *patchFile* corresponde ao nome do ficheiro onde se encontram definidos os pontos de controle e os *patches* para calcular os pontos de *Bezier*.

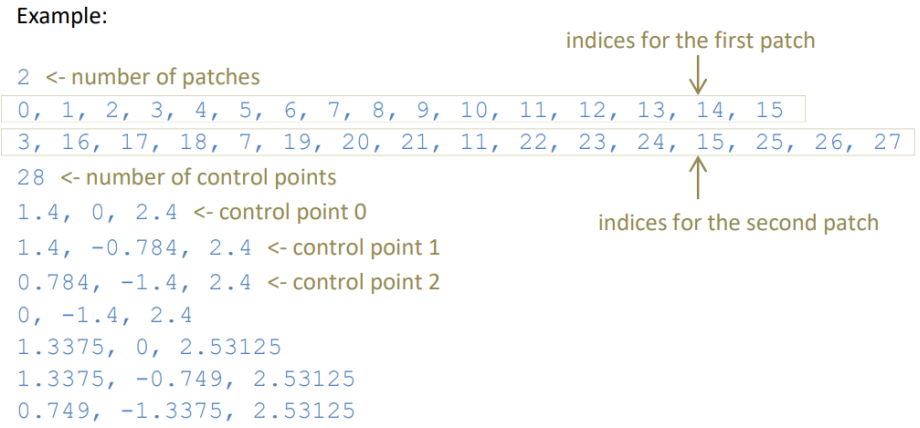

Figura 1 - Formato do ficheiro *.patch*.

O *level* corresponde ao nível de tesselação a ser utilizado e *saveFile* corresponde ao ficheiro .3d de *output* responsável por armazenar o número de vértices a serem desenhados e as especificações desses mesmos vértices.

BEZIER PATCHES

Para o desenvolvimento de modelos 3D a partir de *patches* de *Bezier* utilizou-se como *input* o objeto *teapot* definido no ficheiro *teapot.patch* e um nível de tesselação.

Desta forma, começamos por definir a função *patch* responsável por ler cada linha do ficheiro *teapot.patch* e por armazenar em variáveis do tipo *int* o número de *patches* e de pontos especificados no ficheiro. Para além disso, a função armazena na variável *vector<vector<int>> patches* os 16 índices de cada *patch* e na variável *vector<vector<float>> cpoints* as 3 coordenadas de cada ponto.

De seguida, calcula-se os pontos de *Bezier* a partir da função *getBezierPoint* que recebe como parâmetros as variáveis *u* e *v* do tipo *float*, uma variável *float points[16][3]* para armazenar os pontos correspondentes a cada índice de uma determinada *patch* e uma variável *pos[3]* para armazenar o ponto calculado. Desta forma, a função é chamada dentro de dois ciclos: um, mais externo, que faz variar a variável *u* o valor de *delta_u* e outro, mais interno, que faz variar a variável *v* o valor de *delta_v*. Ambas as variáveis *delta_u* e *delta_v* assumem a seguinte definição:

$$\frac{1}{tesselation}$$

Quanto maior for o valor de *tesselation*, mais quadrados serão desenhados e maior será o detalhe da figura. Assim, em cada iteração calculamos os vértices do quadrado formado por dois triângulos com 3 vértices cada.

```
// Calcula os pontos dos quadrados
for (float u = 0; u < 1; u += delta_u) {
    for (float v = 0; v < 1; v += delta_v) {
        // Triangulo 1
        getBezierPoint(u, v, points, pos);
        write << pos[0] << " " << pos[1] << " " << pos[2] << endl;
        getBezierPoint(u, v + delta_v, points, pos);
        write << pos[0] << " " << pos[1] << " " << pos[2] << endl;
        getBezierPoint(u + delta_u, v, points, pos);
        write << pos[0] << " " << pos[1] << " " << pos[2] << endl;

        // Triangulo 2
        getBezierPoint(u + delta_u, v + delta_v, points, pos);
        write << pos[0] << " " << pos[1] << " " << pos[2] << endl;
        getBezierPoint(u + delta_u, v, points, pos);
        write << pos[0] << " " << pos[1] << " " << pos[2] << endl;
        getBezierPoint(u, v + delta_v, points, pos);
        write << pos[0] << " " << pos[1] << " " << pos[2] << endl;
    }
}
```

Figura 2 - Cálculo dos pontos de *Bezier*.

Como tal, na função *getBezierPoint* utilizamos a seguinte expressão para calcular cada ponto de *Bezier, p(u,v)*:

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

Por sua vez, a matriz central será alterada a cada iteração durante a travessia do vetor que armazena os *patches*. Desta forma, cada linha da matriz representa uma curva de *Bezier* constituída por 4 pontos de determinada componente x, y ou z.

Para além disso, tanto a matriz M como a sua transposta (M^T) são dadas pela seguinte expressão:

$$M = M^T \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

```
void getBezierPoint(float u, float v, float (*points)[3], float* pos) {
    float m[4][4] = { {-1.0f, 3.0f, -3.0f, 1.0f},
                      {3.0f, -6.0f, 3.0f, 0.0f},
                      {-3.0f, 3.0f, 0.0f, 0.0f},
                      {1.0f,  0.0f, 0.0f, 0.0f} };

    // Vetores U e V
    float U[4] = { u * u * u, u * u, u, 1 };
    float V[4] = { v * v * v, v * v, v, 1 };

    // Armazena cada componente dos 16 pontos
    float p[3][16];
    for (int i = 0; i < 16; i++) {
        p[0][i] = points[i][0];
        p[1][i] = points[i][1];
        p[2][i] = points[i][2];
    }

    // Cálculo para cada componente x, y e z do ponto
    for (int i = 0; i < 3; i++) { // x, y, z
        float A1[4];
        float A2[4];

        // Compute A1 = U * M
        multMatrixVector((float*)m, U, A1);

        // Compute A2 = A1 * P
        multMatrixVector(p[i], A1, A2);

        // Compute A1 = A2 * MT (where MT = M)
        multMatrixVector((float*)m, A2, A1);

        // Compute pos[i] = A1 * V  (i.e. U * M * P * MT * V)
        pos[i] = V[0] * A1[0] + V[1] * A1[1] + V[2] * A1[2] + V[3] * A1[3];
    }
}
```

Figura 3 - Função *getBezierPoint*.

Como podemos verificar, utilizamos a função *multMatrixVector* as vezes necessárias para calcular o valor de *p(u,v)* para cada componente x, y e z. De seguida estes valores são armazenados na variável *pos* e escritos, posteriormente, no ficheiro *saveFile*.

ENGINE

No *engine*, como foi referido, foi necessário adaptá-lo de forma a ser mais eficiente e a possibilitar a integração de animações a partir de translações e de rotações com tempo.

PARSE XML

Para possibilitar a extensão das transformações *rotate* e *translate* foi necessário alterar o modo de leitura dos ficheiros XML, de forma a suportar transformações com tempo e com pontos de controle:

```
<transform>
    <translate time = "10" align="true">
        <point x = "0" y = "0" z = "4" />
        <point x = "4" y = "0" z = "0" />
        <point x = "0" y = "0" z = "-4" />
        <point x = "-4" y = "10" z = "0" />
    </translate>
    <rotate time="5" x="1" y="0" z="0" />
    <scale x="0.5" y="0.5" z="0.5" />
</transform>
```

Figura 4 - Extensão dos elementos *translate* e *rotate*.

Assim, começamos por alterar a classe *Transformation* adicionando os vetores *px*, *py* e *pz* para armazenar as componentes dos pontos de controle definidos no XML. Para além disso, adicionamos também a variável *align* para definir se o objeto está ou não alinhado com a sua trajetória. Por fim, o valor do atributo *time* é armazenado na variável *params* já existente na segunda fase do projeto.

De seguida, adaptamos a função *loadGroup* para que possa armazenar elementos estáticos (*time=0*) e de elementos dinâmicos (*time!=0*) com a mesma lógica definida na segunda fase do trabalho.

```
// Parse do elemneto translate
if (strcmp(node->name(), "translate") == 0) {
    xml_attribute<>* x = node->first_attribute("x");
    if (x) { // Elemento estático
        params.push_back(0.0f); // time = 0.0f
        params.push_back(atof(x->value()));
        params.push_back(atof(node->first_attribute("y")->value()));
        params.push_back(atof(node->first_attribute("z")->value()));
    }
    else { // Elemento com movimento
        params.push_back(atof(node->first_attribute("time")->value()));
        char* align_prev = node->first_attribute("align")->value();
        int align = 1;

        if (!strcmp(align_prev, "false")) {
            align = 0;
        }

        xml_node<>* point = node->first_node("point");
        while (point) {
            px.push_back(atof(point->first_attribute("x")->value()));
            py.push_back(atof(point->first_attribute("y")->value()));
            pz.push_back(atof(point->first_attribute("z")->value()));

            point = point->next_sibling("point");
        }

        transform.setPx(px);
        transform.setPy(py);
        transform.setPz(pz);
        transform.setAlign(align);
    }

    transform.setTipo("translate");
    transform.setParams(params);
    transformations.push_back(transform);
}
```

Figura 5 - *Parse* de um elemento *translate*.

```
// Parse do elemento rotate
if (strcmp(node->name(), "rotate") == 0) {
    xml_attribute<>* rotate = node->first_attribute("angle");
    if (rotate){
        params.push_back(atof(rotate->value()));
    }
    else {
        params.push_back(0.0f);
    }

    xml_attribute<>* time = node->first_attribute("time");
    if (time) {
        params.push_back(atof(time->value()));
    }
    else {
        params.push_back(0.0f);
    }

    params.push_back(atof(node->first_attribute("x")->value()));
    params.push_back(atof(node->first_attribute("y")->value()));
    params.push_back(atof(node->first_attribute("z")->value()));
    transform.setTipo("rotate");
    transform.setParams(params);
    transformations.push_back(transform);
}
```

Figura 6 - *Parse* de um elemento *rotate*.

DESENHO DOS OBJETOS

Após a leitura e o armazenamento da informação contida no ficheiro XML, passamos a tratar da sua renderização. Para isso, modificamos a função *drawFigures* para que possa aplicar as novas translações e rotações.

TRANSLAÇÃO

De modo a aplicar as translações com tempo, foi necessário recorrer ao cálculo de curvas de *Catmull-Rom* a partir da função *getGlobalCatmullRomPoint,* chamada quando a função *drawFigures* encontra um *translate* com tempo associado.

```
void getGlobalCatmullRomPoint(float time, vector<float> px, vector<float> py, vector<float> pz, float* pos, float* deriv) {
    int POINT_COUNT = (int)px.size();

    // Tempo desde o início do programa / tempo de duração da animação (em milissegundos)
    float gt = (float)glutGet(GLUT_ELAPSED_TIME) / (time * 1000);

    // Tempo local (real)
    float t = gt * POINT_COUNT;
    int index = floor(t);
    t = t - index;

    // Determina os índices dos pontos de controlo
    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;

    // Determina os pontos de controlo a serem utilizados
    float p[4][3];
    p[0][0] = px[indices[0]];
    p[0][1] = py[indices[0]];
    p[0][2] = pz[indices[0]];

    p[1][0] = px[indices[1]];
    p[1][1] = py[indices[1]];
    p[1][2] = pz[indices[1]];

    p[2][0] = px[indices[2]];
    p[2][1] = py[indices[2]];
    p[2][2] = pz[indices[2]];

    p[3][0] = px[indices[3]];
    p[3][1] = py[indices[3]];
    p[3][2] = pz[indices[3]];

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

Figura 7 - Função *getGlobalCatmullRomPoint*.

De acordo com esta função, é preciso existir pelo menos 4 pontos de controle para definir a curva. Estes pontos são armazenados durante a leitura do ficheiro XML e encontram-se associados à respetiva transformação a partir das variáveis *px*, *py* e *pz*.

Assim, caso existam mais do que 4 pontos, a função trata de encontrar quais os 4 a serem usados para desenhar o segmento da curva num determinado instante calculado com o auxílio da função *glutGet(GLUT_ELAPSED_TIME)*. De seguida, através da função *getCatmullRomPoint*, calculamos a posição atual e a derivada de acordo com as seguintes fórmulas:

$$p(t) = [t^3 \;\; t^2 \;\; t \;\; 1] \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

$$p'(t) = [3t^2 \;\; 2t \;\; 1 \;\; 0] \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

Por fim, caso a figura tenha que estar alinhada com a sua trajetória (*align=true*), calcula-se os eixos normais do objeto:

```
// Tempo desde o inicio do programa / tempo de duração da animação (em milissegundos)
float gt = (float)glutGet(GLUT_ELAPSED_TIME) / (time * 1000);
getGlobalCatmullRomPoint(gt, px, py, pz, Pos, Deriv);
glTranslatef(Pos[0], Pos[1], Pos[2]);

if (align) {
    float rot[16];
    float* X = Deriv;
    float Y[3] = { 0,1,0 }; // Variável que define a normal do objeto
    float Z[3];

    // Z = X x Y
    cross(X, Y, Z);
    // Y = Z x X
    cross(Z, X, Y);

    // Normaliza os vetores
    normalize(X);
    normalize(Y);
    normalize(Z);

    buildRotMatrix(X, Y, Z, rot);

    glMultMatrixf(rot);
}
```

Figura 8 - Normalização e cálculo vetorial.

ROTAÇÃO

De modo a implementar dinamicamente a rotação, foi necessário calcular o ângulo de rotação em cada instante a partir do valor de *time*, associado à transformação dinâmica, e do valor dado pela função *glutGet(GLUT_ELAPSED_TIME)*.

```
else if (transf.getTipo() == "rotate") {
    float angle = params[0];
    float time = params[1];
    float x = params[2];
    float y = params[3];
    float z = params[4];

    // Objeto estático
    if (time == 0) {
        glRotatef(angle, x, y, z);
    }
    else{ // Objeto com movimento
        float t = (float)glutGet(GLUT_ELAPSED_TIME) / (time * 1000);
        glRotatef(t * 360, x, y, z);
    }
}
```

Figura 9 - Cálculo das rotações.

VBOs

De modo a aumentar a performance do programa, implementou-se, nesta terceira fase, VBOs responsáveis pelo desenho das primitivas. Assim, definiu-se a variável *vector<float> vertexB* para armazenar as coordenadas de cada ponto lido na função *readFile*. Para além disso, definimos também a variável *ptr* para controlar e registar a zona do VBO a ser desenhada em cada momento.

Desta forma, o VBO é criado através da função *glGenBuffers* definida em *init* e preenchido, como foi referido, aquando a leitura do ficheiro .3d, na função *readFile*. O processo de desenho do conteúdo do VBO dá-se na função *drawFigures* que, ao percorrer um vetor de figuras, desenha cada uma delas a partir das funções evidenciadas na figura abaixo:

```
// Percorremos e desenhamos as figuras com VBOs
for (int i = 0; i < g.figures.size(); i++) {
    Figure fig = g.figures[i];
    int n_vertices = fig.getNrVertices();

    // Ativa e define o tipo de buffer (array)
    glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);
    // Forma como organizamos os vértices
    glVertexPointer(3, GL_FLOAT, 0, 0);
    // Desenha os triângulos (conjuntos de 3 vértices)
    glDrawArrays(GL_TRIANGLES, ptr, n_vertices);

    // O pointer passa para o próximo conjunto de vértices
    ptr = ptr + n_vertices;
}
```

Figura 10 - Desenho das primitivas utilizando VBOs.

SISTEMA SOLAR

De modo a cumprir o objetivo desta terceira fase, adicionamos ao ficheiro XML as transformações necessárias para animar o Sistema Solar.

Assim, começamos por criar o programa *controlPointsGenerator.py* que, para um determinado raio, é capaz definir 12 pontos de controle de forma a definir a órbita de cada planeta a partir das curvas de *Catmull-Rom*.

```python
import math

def dividir_circunferencia(r):
    # loop para dividir a circunferência em 12 fatias iguais
    for i in range(1,360,30):
        # ângulo da fatia em relação ao eixo x
        angulo = i * math.pi/180

        # coordenadas do ponto na circunferência no formato do xml
        print(f"<point x=\"{round(r * math.cos(angulo),3)}\" y=\"{round(r * math.sin(angulo),3)}\" z=\"0\"/>")

dividir_circunferencia(21)
```

Figura 11 - Programa *controlPointsGenerator*.py.

```
<!-- TERRA -->
<group>
    <transform>
        <color x="0.058" y="0.156" z="0.423"/>
        <translate time="21" align="true">
            <point x="-9" y="0" z="0"/>
            <point x="-7.76" y="0" z="4.51"/>
            <point x="-4.51" y="0" z="7.76"/>
            <point x="0" y="0" z="9"/>
            <point x="4.51" y="0" z="7.76"/>
            <point x="7.76" y="0" z="4.51"/>
            <point x="9" y="0" z="0"/>
            <point x="7.76" y="0" z="-4.51"/>
            <point x="4.51" y="0" z="-7.76"/>
            <point x="0" y="0" z="-9"/>
            <point x="-4.51" y="0" z="-7.76"/>
            <point x="-7.76" y="0" z="-4.51"/>
        </translate>
        <rotate time="5" x="0" y="1" z="0"/>
        <scale x="0.05" y="0.05" z="0.05"/>
    </transform>
    <models>
        <model file="spherePrototype_4_80_80.3d" />
    </models>
```

```
<!-- LUA -->
<group>
    <transform>
        <color x="1" y="1" z="1"/>
        <rotate angle="18" x="0" y="0" z="1"/>
        <translate time="27" align="true">
            <point x="-7" y="0" z="0"/>
            <point x="-6.12" y="0" z="3.54"/>
            <point x="-3.54" y="0" z="6.12"/>
            <point x="0" y="0" z="7"/>
            <point x="3.54" y="0" z="6.12"/>
            <point x="6.12" y="0" z="3.54"/>
            <point x="7" y="0" z="0"/>
            <point x="6.12" y="0" z="-3.54"/>
            <point x="3.54" y="0" z="-6.12"/>
            <point x="0" y="0" z="-7"/>
            <point x="-3.54" y="0" z="-6.12"/>
            <point x="-6.12" y="0" z="-3.54"/>
        </translate>
        <scale x="0.25" y="0.25" z="0.25"/>
    </transform>
    <models>
        <model file="spherePrototype_4_80_80.3d" />
    </models>
</group>
</group>
```

Figura 12 - Excerto do ficheiro XML.

De modo a tornar o modelo mais realista, utilizou-se os tempos reais para definir a escala de translação dos planetas à volta do Sol com valores normalizados entre 20 e 300, a escala de rotação dos planetas sobre si próprios com valores entre 5 e 10 . Para além disso, tivemos em conta os sentidos reais de cada planeta para definir a ordem correta dos pontos de controle dos mesmos.

Planetas	Período Translação (dias)	Período Translação (normalizado)	Período Rotação (dias)	Período Rotação (normalizado)	Sentido
Mercúrio	87,97	20	58,6	6	Anti-horário
Vénus	224,7	21	243	10	Horário
Terra	365,26	21	1	5	Anti-horário
Marte	686,67	23	1,03	5	Anti-horário
Júpiter	4331,86	40	0,41	5	Anti-horário
Saturno	10.760,26	70	0,45	5	Anti-horário
Urano	30.684,65	163	0,72	5	Horário
Neptuno	60.189,55	300	0,67	5	Anti-horário

Tabela 1 - Informações dos planetas.

OUTPUT

De seguida apresenta-se o *output* do programa finalizado nesta fase:

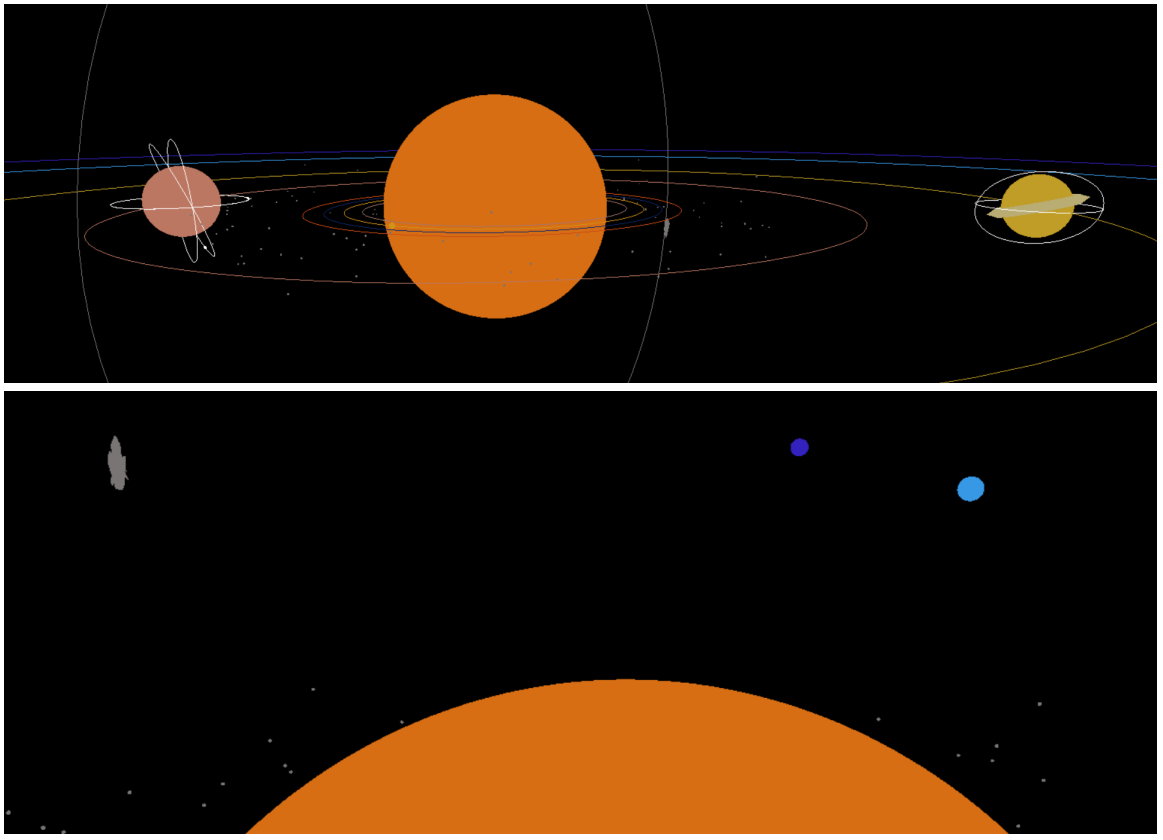

Figura 13 - Sistema Solar.

COMETA

Ainda nesta fase, foi-nos proposto a criação de um cometa no Sistema Solar a partir dos pontos de controle utilizados no *teapot*.

Assim, começamos por redefinir e eliminar algumas *patches* utilizadas no *teapot* de forma a aplicar no ficheiro *comet.patch* apenas a parte esférica do *teapot*. Depois, com o objetivo de deformar a superfície esférica, criamos um *script*, *cometGenerator.py*, que recebe uma *string* de pontos de controle e soma a cada um deles um valor aleatório entre -1 e 1. Por fim, após a utilização desta configuração para gerar o cometa, aplicamos uma escala assimétrica no eixo do x ao mesmo.

```python
pontos = []

for line in pontos_originais.strip().split('\n'):
    ponto = []
    for coord in line.strip().split(', '):
        newCoord = random.uniform(-1, 1)
        newCoord += float(coord)
        ponto.append(round(newCoord,5))

    pontos.append(ponto)

for ponto in pontos:
   print(f"{ponto[0]}, {ponto[1]}, {ponto[2]}")
```

Figura 14 - *Script* em *python*.

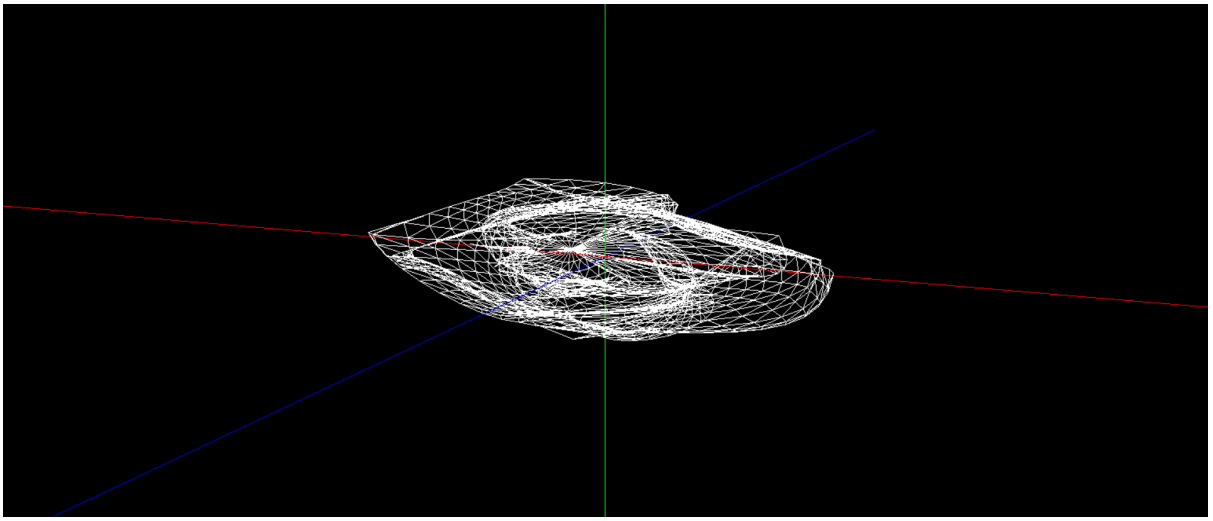

Figura 15 - Cometa.

DEMO SCENES

Com o objetivo de visualizar melhor o trabalho realizado, apresentamos de seguida um conjunto de cenas resultantes dos testes realizados ao longo desta terceira fase do projeto.

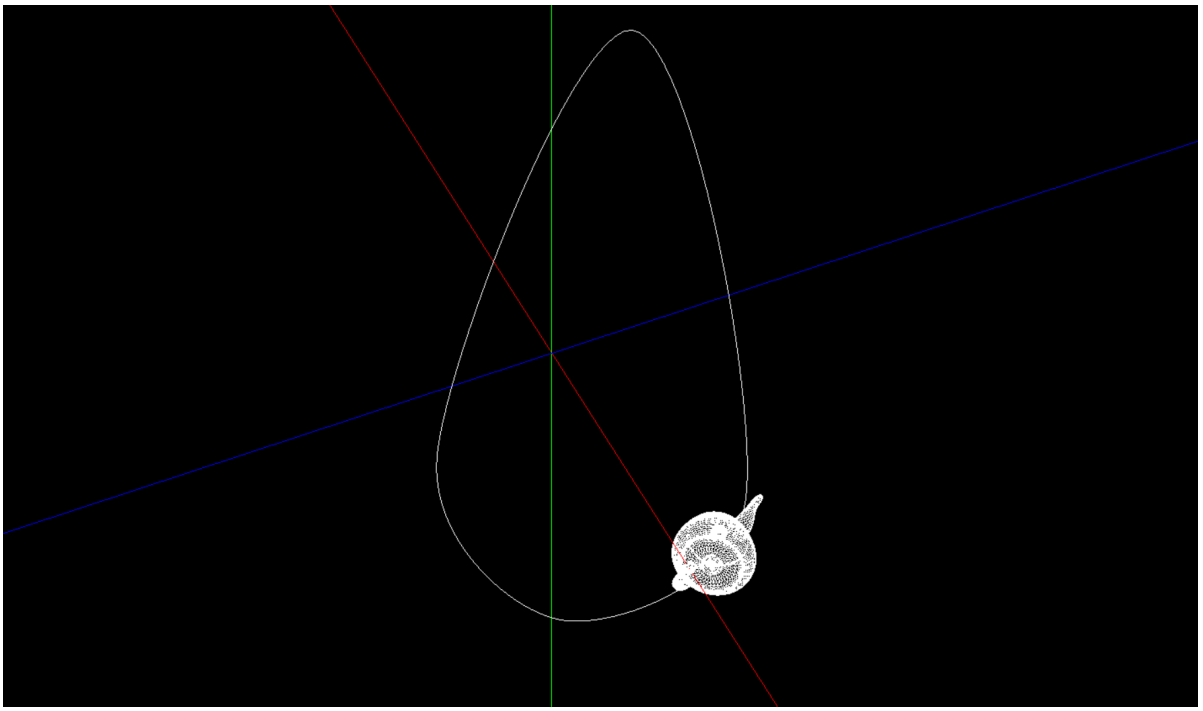

Figura 16 - Resultado do teste *test_3_1*.

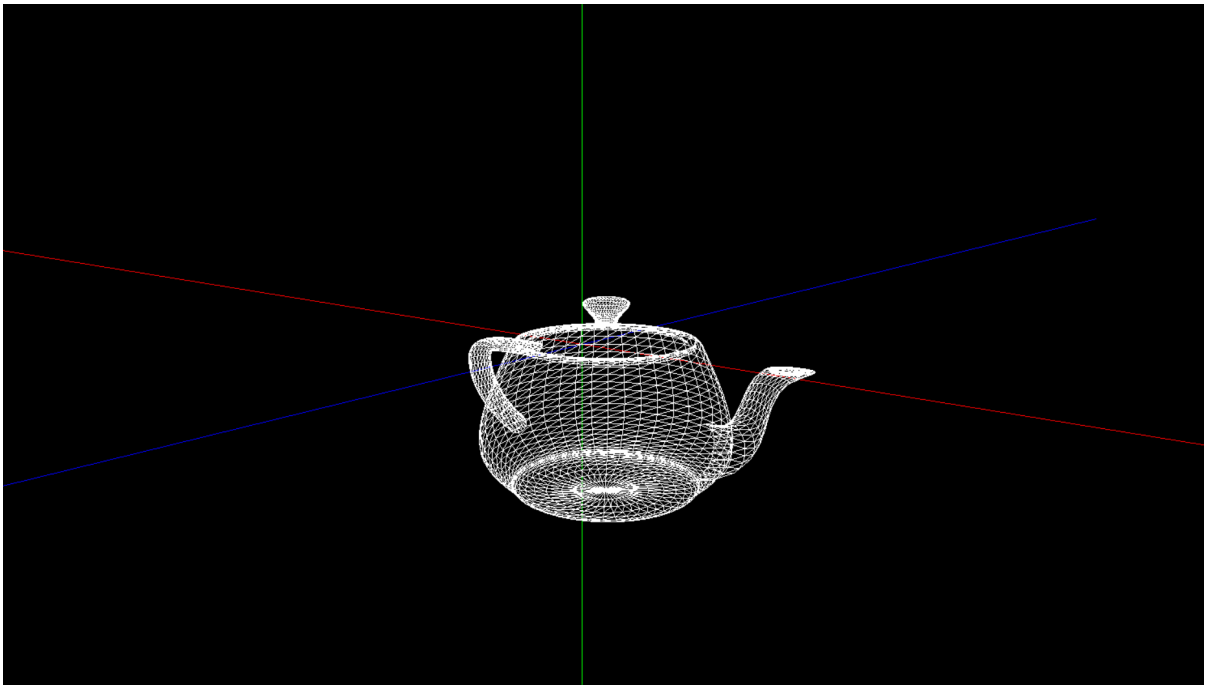

Figura 17 - Resultado do teste *test_3_2*.

CONCLUSÃO

Nesta terceira fase do projeto consideramos que apesar de ter sido mais trabalhosa, foi produtiva, sendo possível assimilar os conteúdos lecionados na Unidade Curricular. Ao longo do projeto, foram implementadas curvas de *Bezier*, rotações e translações cíclicas dos constituintes do Sistema Solar e utilizadas funcionalidades VBO.

Apesar de considerarmos que houve um balanço positivo do trabalho, sentimos que há aspetos a melhorar, como a implementação da cintura de asteroides com VBOs, assim como as animações e a sua especificação num ficheiro XML.