



**Universidade do Minho**  
Escola de Engenharia  
Licenciatura em Engenharia Informática

## **Computação Gráfica**

Ano Letivo 2022/2023

# **Trabalho prático**

## **Parte 3 - Curvas, superfícies e VBO's**

### **Grupo 14**

Ana Rita Santos Poças, A97284  
Miguel Silva Pinto, A96106  
Orlando José da Cunha Palmeira, A97755  
Pedro Miguel Castilho Martins, A97613

3 de maio de 2023

# **Trabalho prático**

## **Parte 3 - Curvas, superfícies e VBO's**

### **Grupo 14**

Ana Rita Santos Poças, A97284

Miguel Silva Pinto, A96106

Orlando José da Cunha Palmeira, A97755

Pedro Miguel Castilho Martins, A97613

3 de maio de 2023

# Índice

<b>1</b>	<b>Introdução</b>	<b>1</b>
<b>2</b>	<b>Alterações às estruturas de dados</b>	<b>2</b>
<b>3</b>	<b>Funcionalidades</b>	<b>3</b>
3.1	<i>Generator</i> . . . . .	3
3.1.1	Construção de modelos baseados em <i>patches</i> de Bezier . . . . .	3
3.2	<i>Engine</i> . . . . .	6
3.2.1	Rotações dependentes do tempo . . . . .	6
3.2.2	Translações dirigidas por curvas de Catmull-Rom . . . . .	7
3.2.3	Utilização de VBO's . . . . .	9
3.2.4	Movimentação livre da câmara . . . . .	11
3.3	Cometa . . . . .	12
<b>4</b>	<b>Animação do sistema solar</b>	<b>13</b>
<b>5</b>	<b>Conclusão</b>	<b>15</b>
	<b>Anexos</b>	<b>16</b>
	Anexo 1 - Como utilizar o <i>Generator</i> . . . . .	16
	Anexo 2 - Como iniciar o <i>Engine</i> . . . . .	16
	Anexo 3 - Comandos do <i>Engine</i> . . . . .	17
	Anexo 4 - Significado das informações que aparecem no título da janela do <i>Engine</i> . . . . .	17
	Anexo 5 - Estrutura do projeto . . . . .	18

# Índice de figuras

2.1	Antiga implementação da <i>struct Transform</i> . . . . .	2
2.2	Nova implementação da <i>struct Transform</i> . . . . .	2
3.1	Obtenção do número de <i>patches</i> . . . . .	3
3.2	Obtenção dos índices dos pontos de cada <i>patch</i> . . . . .	3
3.3	Obtenção dos pontos presentes no ficheiro . . . . .	4
3.4	Construção dos <i>patches</i> . . . . .	4
3.5	Função <i>generateSurface</i> . . . . .	5
3.6	Função <i>surfacePoint</i> . . . . .	5
3.7	<i>Teapot</i> gerado pelo <i>generator</i> apresentado no <i>engine</i> . . . . .	6
3.8	Excerto de código da função <i>executeTranformations</i> . . . . .	7
3.9	Função <i>getCatmullRomPoint</i> . . . . .	8
3.10	Função <i>getGlobalCatmullRomPoint</i> . . . . .	8
3.11	Implementação das translações . . . . .	9
3.12	Função <i>loadBuffersData</i> . . . . .	10
3.13	Função <i>drawGroups</i> . . . . .	10
3.14	Ângulos $\alpha$ e $\beta$ . . . . .	11
3.15	Grelha que compõe um <i>patch</i> . . . . .	12
3.16	Cometa gerado pelo programa em <i>Python</i> . . . . .	12
4.1	Sistema solar . . . . .	13
4.2	Sistema solar com as trajetórias dos cometas . . . . .	14

# 1 Introdução

A terceira fase do projeto da cadeira de **Computação Gráfica** consistiu na progressão do trabalho já realizado nas fases anteriores, acrescentando novas funcionalidades às duas aplicações desenvolvidas, o *Generator* e o *Engine*.

As atualizações contemplaram a capacidade do *Generator* produzir modelos baseados em *patches* de Bezier bem como a capacidade do *Engine* interpretar e aplicar novos tipos de translações e rotações. Para além disso, o *Engine* também utiliza **VBO's** de modo a permitir um melhor desempenho na visualização dos modelos produzidos.

Os novos tipos de translações e rotações permitem que estas transformações geométricas sejam animadas. As rotações, para além de estáticas, podem ser dependentes do tempo da animação e as translações podem ser dirigidas por curvas de Catmull-Rom.

Ao longo deste relatório, iremos descrever de forma detalhada as decisões e abordagens que foram adotadas e que permitiram a implementação das funcionalidades propostas.

## 2 Alterações às estruturas de dados

As funcionalidades exigidas para esta fase obrigaram a realizar alterações na estrutura relativa às transformações geométricas, a *Transform*.

Na segunda fase, a *struct transform* foi definida do seguinte modo:

```
struct transform{  
    char type;  
    float x,y,z;  
    float angle;  
};
```

Figura 2.1: Antiga implementação da *struct Transform*

Esta definição serviu enquanto as translações e rotações eram apenas estáticas. Para estas transformações poderem ser animadas, tivemos de acrescentar alguns campos, fazendo com que a estrutura esteja definida do seguinte modo:

```
struct transform{  
    char type;  
    float x,y,z;  
    float angle;  
    float time;  
    bool align;  
    vector<vector<float>>*> points;  
    float yAxis[3];  
};
```

Figura 2.2: Nova implementação da *struct Transform*

Dentro dos novos campos que foram adicionados à *Transform*, o *time* será utilizado tanto pela rotação como pela translação. Já os campos *align*, *points* e *yAxis* apenas serão utilizados pela translação.

## 3 Funcionalidades

Neste capítulo iremos descrever o processo de implementação das funcionalidades propostas para esta fase do projeto.

### 3.1 Generator

#### 3.1.1 Construção de modelos baseados em *patches* de Bezier

Para construir os modelos baseados em *patches* de Bezier, precisamos, em primeiro lugar, de ler os ficheiros `.patch` que contêm as informações dos *patches*. Para esse fim, implementámos a função `readPatchesFile`.

A implementação da função `readPatchesFile` foi feita do seguinte modo:

1. Começamos por obter o número de *patches* que o ficheiro contém.

```
vector<vector<vector<float>>> readPatchesFile(const char* filePath){
    FILE* file = fopen(filePath,"r");
    vector<vector<vector<float>>> result;
    if(file){
        char buffer[2048];
        // Obtenção do número de patches
        if(!fgets(buffer,2047,file)) return result;
        int numPatches = atoi(buffer);
```

Figura 3.1: Obtenção do número de *patches*

2. Obtemos os índices dos pontos de cada *patch*.

```
vector<vector<int>> indicesPerPatch;

// Obtenção dos índices de cada patch
for(int i = 0; i < numPatches; i++){
    if(!fgets(buffer,2047,file)) return result;
    vector<int> indices;
    for(char* token = strtok(buffer,","); token; token = strtok(NULL,",")){
        indices.push_back(atoi(token));
    }
    indicesPerPatch.push_back(indices);
}
```

Figura 3.2: Obtenção dos índices dos pontos de cada *patch*.

3. Obtemos todos os pontos presentes no ficheiro.

```
// Obtenção do número de pontos de controlo
if(!fgets(buffer,2047,file)) return result;
int numControlPoints = atoi(buffer);

// Obtenção dos pontos de controlo
vector<vector<float>> controlPoints;
for(int i = 0; i < numControlPoints; i++){
    if(!fgets(buffer,2047,file)) return result;
    vector<float> point;
    for(char* token = strtok(buffer,","); token; token = strtok(NULL,",")){
        point.push_back(atoi(token));
    }
    controlPoints.push_back(point);
}
```

Figura 3.3: Obtenção dos pontos presentes no ficheiro

4. Fazemos a construção dos *patches*.

A construção dos *patches* consiste em ir a cada `vector<int>` de índices recolher os índices dos pontos do *patch* que vai ser construído e, a partir de cada índice, recolher o respetivo ponto para o adicionar ao *patch*.

```
// Construção dos patches
for(vector<int> indices : indicesPerPatch){
    vector<vector<float>> patch;
    for(int indice : indices){
        vector<float> point;
        point.push_back(controlPoints[indice][0]);
        point.push_back(controlPoints[indice][1]);
        point.push_back(controlPoints[indice][2]);
        patch.push_back(point);
    }
    result.push_back(patch);
}
fclose(file);
```

Figura 3.4: Construção dos *patches*

Como o resultado da função é um conjunto de *patches* e um *patch* é um conjunto de pontos, então o resultado desta função é um valor do tipo `vector<vector<vector<float>>>` em que `vector<float>` representa um ponto.

Para construir o modelo a partir do *patch*, implementámos a função `generateSurface` que irá calcular os pontos bem como a triangulação conveniente para o modelo ser apresentado no ecrã.

A função `generateSurface` está implementada do seguinte modo:



```

Figura generateSurface(const char* filePath, int tessellation){
    Figura result = newEmptyFigura();
    float u = 0.0f, v = 0.0f, delta = 1.0f/tessellation;
    float A[3], B[3], C[3], D[3];
    vector<vector<vector<float>>> patches = readPatchesFile(filePath);
    for(vector<vector<float>> patch : patches){ // um patch tem 16 pontos
        for(int i = 0; i < tessellation; i++, u += delta){
            for(int j = 0; j < tessellation; j++, v += delta){
                // Cálculo dos pontos
                surfacePoint(u,v,patch,A);
                surfacePoint(u,v+delta,patch,B);
                surfacePoint(u+delta,v,patch,C);
                surfacePoint(u+delta,v+delta,patch,D);
                // Triangulação
                addPontoArr(result,A);
                addPontoArr(result,A);
                addPontoArr(result,B);
                addPontoArr(result,B);
                addPontoArr(result,D);
                addPontoArr(result,C);
            }
            v = 0.0f;
        }
        u = v = 0.0f;
    }
    return result;
}

```

Figura 3.5: Função generateSurface

Esta função começa por inicializar os parâmetros  $u = 0.0f$ ,  $v = 0.0f$  e  $\delta = 1.0f/tessellation$  e por ler o ficheiro `.patch` com a informação dos *patches*. Posteriormente, ela vai calcular os pontos de cada *patch* de acordo com a tesselação fornecida através da função `surfacePoint`.

A função `surfacePoint` calcula um ponto de uma superfície de Bezier dados os parâmetros  $u$ ,  $v$  e o respetivo *patch*.

```

void surfacePoint(float u, float v, vector<vector<float>> patch, float* res){
    float M[16] = {-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}; // 4x4
    float U[4] = {u*u*u,u*u,u,1.0f}, V[4] = {v*v*v,v*v,v,1.0f}; // U: 1x4; V: 4x1
    float UM[4]; multiplyMatrices(1,4,U,4,4,M,UM); // UM: 1x4
    float MV[4]; multiplyMatrices(4,4,M,4,1,V,MV); // MV: 4x1
    float P[3][16] = {{patch[0][0],patch[1][0],patch[2][0],patch[3][0],
                       patch[4][0],patch[5][0],patch[6][0],patch[7][0],
                       patch[8][0],patch[9][0],patch[10][0],patch[11][0],
                       patch[12][0],patch[13][0],patch[14][0],patch[15][0]},
                      {patch[0][1],patch[1][1],patch[2][1],patch[3][1],
                       patch[4][1],patch[5][1],patch[6][1],patch[7][1],
                       patch[8][1],patch[9][1],patch[10][1],patch[11][1],
                       patch[12][1],patch[13][1],patch[14][1],patch[15][1]},
                      {patch[0][2],patch[1][2],patch[2][2],patch[3][2],
                       patch[4][2],patch[5][2],patch[6][2],patch[7][2],
                       patch[8][2],patch[9][2],patch[10][2],patch[11][2],
                       patch[12][2],patch[13][2],patch[14][2],patch[15][2]}};

    for(int i = 0; i < 3; i++){
        float UMP[4]; // UMP: 1x4
        multiplyMatrices(1,4,UM,4,4,P[i],UMP);
        multiplyMatrices(1,4,UMP,4,1,MV,&res[i]);
    }
}

```

Figura 3.6: Função surfacePoint

O objetivo desta função é implementar a fórmula:

$$p(u, v) = U \cdot M \cdot P \cdot M^T \cdot V^T \quad (3.1)$$

$$\text{Com } U = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix}, V^T = \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix}, M = M^T = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \text{ e } P = \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix}$$

Na realidade, a função `surfacePoint` vai executar a fórmula 3.1 três vezes, uma para cada componente  $x$ ,  $y$  e  $z$  dos pontos da matriz  $P$  para obter cada componente  $x$ ,  $y$  e  $z$  do ponto resultante. É por esse motivo que temos um *array*  $P$  com três matrizes  $4 \times 4$  e um ciclo `for` que itera sobre esse *array* e calcula cada coordenada do ponto resultante em cada iteração do ciclo.

Para testar o funcionamento desta implementação, processámos o ficheiro `teapot.patch` fornecido. O ficheiro `.3d`, gerado pelo *Generator*, quando é aberto no *Engine* gera a seguinte imagem:

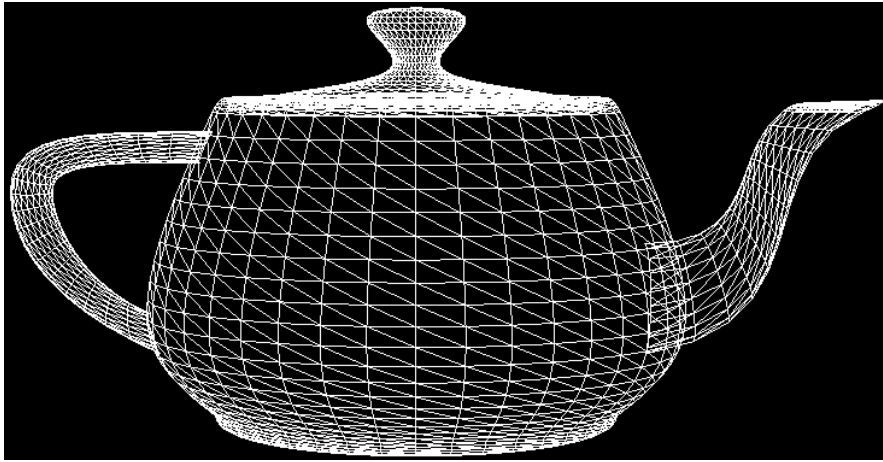


Figura 3.7: *Teapot* gerado pelo *generator* apresentado no *engine*

## 3.2 Engine

### 3.2.1 Rotações dependentes do tempo

Para termos uma rotação animada no *Engine*, precisamos de fornecer na configuração XML o tempo que o objeto demora a realizar uma rotação de 360 graus. Com essa informação, conseguimos deduzir o valor do ângulo de rotação em cada instante da animação.

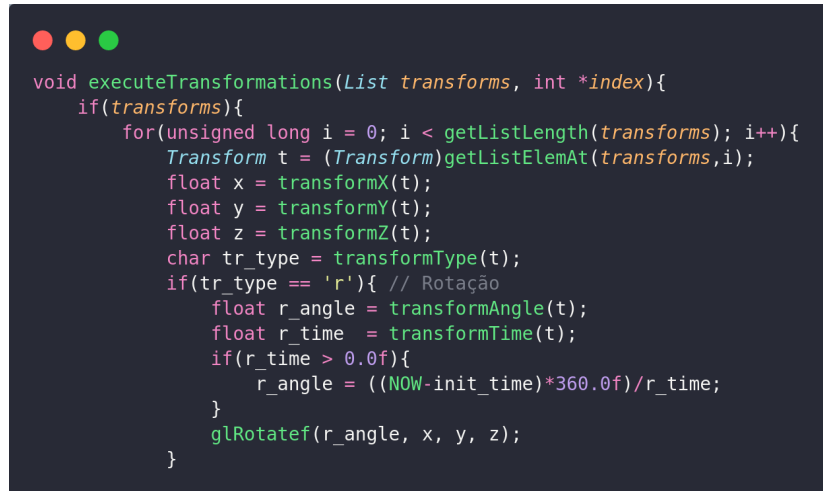
Em cada execução da função `renderScene`, calculamos o valor do ângulo de rotação do objeto em função do instante de tempo atual (obtido através de `glutGet(GLUT_ELAPSED_TIME)`) através da seguinte fórmula:

$$\alpha(t)^\circ = \frac{(t - t_0) \times 360^\circ}{t_r} \quad (3.2)$$

Em que  $t_0$  é o instante em que o *Engine* começou,  $t_r$  é o tempo definido na configuração XML e  $\alpha$  é o ângulo de rotação no instante  $t$ .

Uma vez calculado o valor de  $\alpha$ , executamos a função `glRotatef` com o valor  $\alpha$  e o eixo fornecido na configuração.

O modo como as rotações estão implementadas pode ser demonstrado na figura seguinte:



```
void executeTransformations(List transforms, int *index){
    if(transforms){
        for(unsigned long i = 0; i < getListLength(transforms); i++){
            Transform t = (Transform)getListElemAt(transforms,i);
            float x = transformX(t);
            float y = transformY(t);
            float z = transformZ(t);
            char tr_type = transformType(t);
            if(tr_type == 'r'){ // Rotação
                float r_angle = transformAngle(t);
                float r_time = transformTime(t);
                if(r_time > 0.0f){
                    r_angle = ((NOW-init_time)*360.0f)/r_time;
                }
                glRotatef(r_angle, x, y, z);
            }
        }
    }
}
```

Figura 3.8: Excerto de código da função `executeTranformations`

Neste excerto de código, a variável `t` é uma struct `transform` (figura 2.2) com as informações da rotação provenientes do ficheiro XML. Quando estamos a ler a informação do ficheiro XML, se se tratar de uma rotação estática, o valor do tempo na struct `transform` é igual a 0 (uma vez que tempo igual a 0 não faz sentido neste contexto).

Para saber se se trata de uma rotação dependente do tempo, basta verificar se o tempo na transformação é superior a 0. Se isso acontecer, aplica-se uma rotação segundo o ângulo calculado através da fórmula 3.2. Caso contrário, aplica-se uma rotação segundo o ângulo constante na configuração XML.

### 3.2.2 Translações dirigidas por curvas de Catmull-Rom

Para termos translações animadas no *Engine*, precisamos de fornecer na configuração XML alguns parâmetros acerca da trajetória que o objeto vai tomar. Estes parâmetros incluem os pontos de controlo da curva de Catmull-Rom que define a trajetória do objeto, o tempo que o objeto demora a percorrer toda a curva e se o objeto irá ficar alinhado com a curva.

Para efetuar o cálculo dos pontos da curva, temos duas funções: a `getCatmullRomPoint` e a `getGlobalCatmullRomPoint`, apresentadas a seguir.

```

void getCatmullRomPoint(float t, vector<float> p0, vector<float> p1, vector<float> p2, vector<float> p3, float *pos, float *deriv) {
    // Matriz catmull-rom
    float m[16] = {-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}; // 4x4

    // Matriz P -> contém os pontos de controlo da curva
    float P[12] = {p0[0], p0[1], p0[2],
                  p1[0], p1[1], p1[2],
                  p2[0], p2[1], p2[2],
                  p3[0], p3[1], p3[2]}; // 4x3

    float A[12]; // 4x3
    multiplyMatrices(4,4,m,4,3,P,A);
    float T[4] = {t*t*t, t*t, t, 1}, DERT[4] = {3*t*t, 2*t, 1, 0}; // T-> 1x4, DERT -> 1x4
    if(pos) multiplyMatrices(1,4,T,4,3,A,pos);
    if(deriv) multiplyMatrices(1,4,DERT,4,3,A,deriv);
}

```

Figura 3.9: Função getCatmullRomPoint

A função getCatmullRomPoint serve para calcular um ponto de uma curva de Catmull-Rom dados quatro pontos de controlo. Esta função implementa a seguinte fórmula:

$$p(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \cdot \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} \quad (3.3)$$

A função getGlobalCatmullRomPoint serve para calcular um ponto da curva dado um instante de tempo atual da animação. A função verifica o valor desse instante e recolhe os quatro pontos de controlo da curva onde o valor do instante se encaixa. Posteriormente, fornece esses pontos à função getCatmullRomPoint que retornará o ponto desejado.

```

void getGlobalCatmullRomPoint(float gt, vector<vector<float>> controlPoints, float *pos, float *deriv) {
    size_t POINT_COUNT = controlPoints.size();
    float t = gt * POINT_COUNT; // this is the real global t
    int index = floor(t); // which segment
    t = t - index; // where within the segment

    // indices store the points
    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;

    getCatmullRomPoint(t, controlPoints[indices[0]], controlPoints[indices[1]], controlPoints[indices[2]], controlPoints[indices[3]], pos, deriv);
}

```

Figura 3.10: Função getGlobalCatmullRomPoint

Com a obtenção do instante de tempo atual através de glutGet(GLUT\_ELAPSED\_TIME), conseguimos colocar o objeto a percorrer a curva e visualizar esse percurso na animação. No entanto, se utilizarmos apenas o instante de tempo atual, o objeto demorará apenas 1 segundo a percorrer toda a curva. Para se definir o quanto demora o objeto a percorrer a curva, basta dividir o instante de tempo atual pelo tempo definido no XML. Por exemplo, se quisermos fazer com que um objeto demore 10 segundos a percorrer a curva, dividimos o instante de tempo atual por 10 e calculamos o respetivo ponto da curva para saber a nova posição do objeto. Ao dividir os instantes de tempo, estamos a alterar a taxa de variação da posição do objeto na curva ao longo do tempo e, consequentemente, o tempo que ele demora a percorrer a curva.

A implementação das translações foi feita conforme consta na figura seguinte:

```

float t_time = transformTime(t);
if(t_time > 0.0f){
    float pos[3], deriv[3], y[3], z[3], rot[16];
    vector<vector<float>> points = translatePoints(t);
    getGlobalCatmullRomPoint(NOW/t_time,points,pos,deriv);
    glTranslatef(pos[0],pos[1],pos[2]);

    if(transformAlign(t)){
        normalize(deriv);
        cross(deriv,transformYAxis(t).data(),z);
        normalize(z);
        cross(z,deriv,y);
        setTransformYAxis(t,y);
        normalize(y);
        buildRotMatrix(deriv,y,z,rot);
        glMultMatrixf(rot);
    }

}else{
    glTranslatef(x,y,z);
}

```

Figura 3.11: Implementação das translações

Para alinhar o objeto com a curva, precisamos de definir a orientação dos eixos do seu sistema de coordenadas cada vez que calculamos a sua nova posição. Para isso, calculamos a orientação dos eixos do seguinte modo:

$$X_i = p'(t)$$

$$Y_i = Z_i \times X_i$$

$$Z_i = X_i \times Y_{i-1}$$

Em que "×" representa o produto vetorial,  $p'(t)$  a derivada da curva no instante  $t$  (obtida através da função `getGlobalCatmullRomPoint`) e  $Y_{i-1}$  é a orientação do eixo  $Y$  no instante imediatamente anterior ao atual. Na nossa implementação, consideramos que a orientação inicial do eixo  $Y$  dos sistemas de coordenadas dos objetos segue a orientação do vetor  $(0, 1, 0)$ .

Tal como nas rotações, se na configuração XML não for indicado o tempo, este será considerado igual a 0 e assim a translação executada será estática (figura 3.11).

### 3.2.3 Utilização de VBO's

Na inicialização do *Engine*, este começa por ler o ficheiro de configuração XML e construir a árvore de *groups*. Tendo a árvore *groups* construída, é feita uma travessia *pre-order* na qual carregamos para os *buffers* os pontos de cada modelo presente na árvore através da função `loadBuffersData`.

```

void loadBuffersData(Tree groups, int* index){
    if(groups){
        Group group = (Group)getRootValue(groups);
        List models = getGroupModels(group);

        for(unsigned long i = 0; i < getListLength(models); i++){
            Figura fig = (Figura)getListElemAt(models,i);
            vector<float> toBuffer = figuraToVector(fig);
            glBindBuffer(GL_ARRAY_BUFFER, buffers[*index]++);
            glBufferData(GL_ARRAY_BUFFER, sizeof(float)*toBuffer.size(), toBuffer.data(), GL_STATIC_DRAW);
            buffersSizes.push_back(toBuffer.size()/3);
        }

        List filhos = getChildren(groups);
        for(unsigned long i = 0; i < getListLength(filhos); i++){
            Tree next = (Tree)getListElemAt(filhos, i);
            loadBuffersData(next,index);
        }
    }
}

```

Figura 3.12: Função loadBuffersData

Quando o *Engine* estiver a correr a animação, a função `renderScene`, que por sua vez executa a função `drawGroups`, será executada em cada *frame*. Como a função `drawGroups` vai percorrer a árvore de `groups` pela mesma ordem que a função que carregou os modelos nos *buffers* (`loadBuffersData`) temos a garantia que as transformações geométricas serão aplicadas corretamente nos respetivos modelos.

```

void drawGroups(Tree groups, int* index){
    if(groups){
        glPushMatrix(); // guarda o estado dos eixos

        Group group = (Group)getRootValue(groups);
        List transforms = getGroupTransforms(group);
        unsigned long modelsCount = getListLength(getGroupModels(group));
        executeTransformations(transforms,index);

        // Desenha o conteúdo dos buffers
        for(unsigned long i = 0; i < modelsCount; i++, (*index)++){
            glBindBuffer(GL_ARRAY_BUFFER, buffers[*index]);
            glVertexAttribPointer(3, GL_FLOAT, 0, 0);
            glDrawArrays(GL_TRIANGLES, 0, buffersSizes[*index]);
        }

        // Procede para fazer o mesmo nos nodos filhos.
        List filhos = getChildren(groups);
        for(unsigned long i = 0; i < getListLength(filhos); i++){
            Tree next = (Tree)getListElemAt(filhos, i);
            drawGroups(next,index);
        }
        glPopMatrix(); // retorna ao respetivo estado anterior dos eixos.
    }
}

```

Figura 3.13: Função drawGroups

### 3.2.4 Movimentação livre da câmara

Até à segunda fase deste projeto, a câmara do *Engine* apenas se podia movimentar através de coordenadas esféricas em que o parâmetro *lookAt* era sempre o ponto  $(0, 0, 0)$ .

Para a câmara não ter uma movimentação tão limitada, decidimos que esta pode passar a ter dois modos: *SPHERICAL* e *FREE*. O utilizador pode alternar entre esses dois modos ao clicar na tecla **V**.

O modo *FREE* permite que a câmara se desloque para a frente, para trás, para a esquerda, para a direita, para cima e para baixo utilizando, respetivamente, as teclas **W, S, A, D, +** e **-**. Este modo permite também alterar o ponto para onde a câmara está a olhar (*lookAt*) em que as suas coordenadas são calculadas do seguinte modo ( $x_c$ ,  $y_c$  e  $z_c$  são as coordenadas da posição da câmara):

$$x = x_c + \cos(\beta) \times \sin(\alpha)$$

$$y = y_c + \sin(\beta)$$

$$z = z_c + \cos(\beta) \times \cos(\alpha)$$

Abaixo encontra-se uma figura que clarifica o que são os ângulos  $\alpha$  e  $\beta$  utilizados nas equações acima.

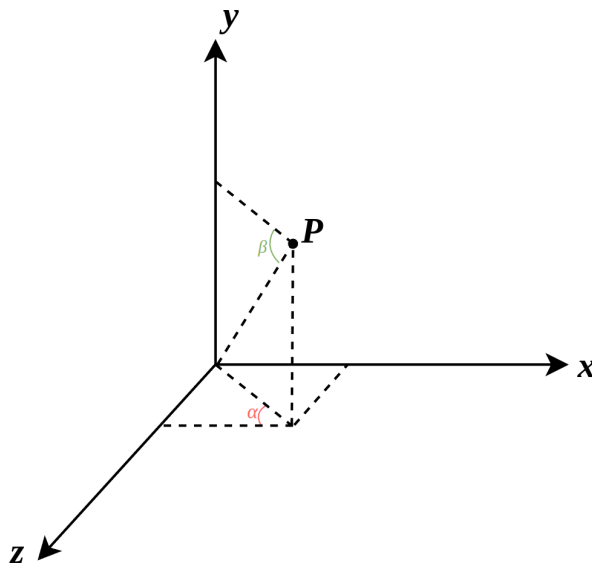


Figura 3.14: Ângulos  $\alpha$  e  $\beta$

### 3.3 Cometa

Para além de utilizarmos o *teapot* como cometa, decidimos criar um que tenta imitar um cometa da vida real. Para isso, implementámos um pequeno programa em *Python* que tenta construir o cometa da seguinte forma:

1. Começa por criar vários *patches* de Bezier que, em conjunto, formam uma esfera.
2. Em cada *patch* dessa esfera vai ser introduzida uma irregularidade nos seus pontos centrais, isto é, alterar-se-á a distância desses pontos à origem através de números aleatórios.

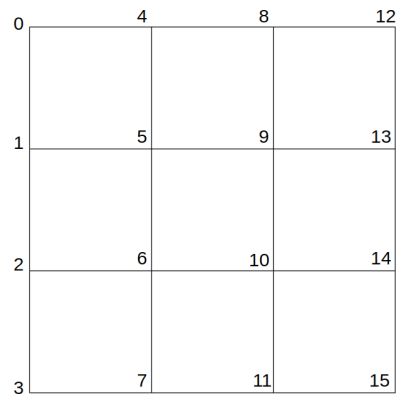


Figura 3.15: Grelha que compõe um *patch*

Imaginando que um *patch* da esfera se encontra no formato da figura acima (os números são os índices dos pontos do *patch*), nós vamos introduzir a irregularidade da distância à origem nos pontos cujos índices são 5, 6, 9 e 10. Desta forma obtemos uma figura irregular em que não existe uma grande discrepância nas fronteiras entre os vários *patches* que a compõem.

Na figura abaixo, encontra-se uma demonstração de uma figura gerada através desta estratégia.

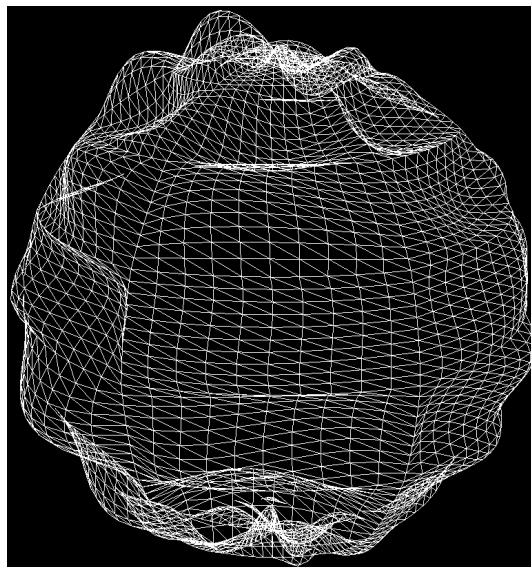


Figura 3.16: Cometa gerado pelo programa em *Python*



## 4 Animação do sistema solar

A cena de demonstração do sistema solar inclui todos os planetas, o sol, a lua do planeta terra e dois cometas.

O movimento de translação dos planetas e da lua é dirigido por rotações dependentes do tempo. Já as trajetórias dos dois cometas são feitas com translações dirigidas por curvas de Catmull-Rom. A trajetória do cometa *teapot* é elíptica e tem uma inclinação de  $15^\circ$  relativamente ao plano  $XZ$  em torno do vetor  $(0, 0, 1)$ . O cometa construído por nós tem uma trajetória circular ondulada.

Nas figuras seguintes apresentamos o nosso sistema solar com a presença dos dois cometas bem como os desenhos das suas trajetórias:

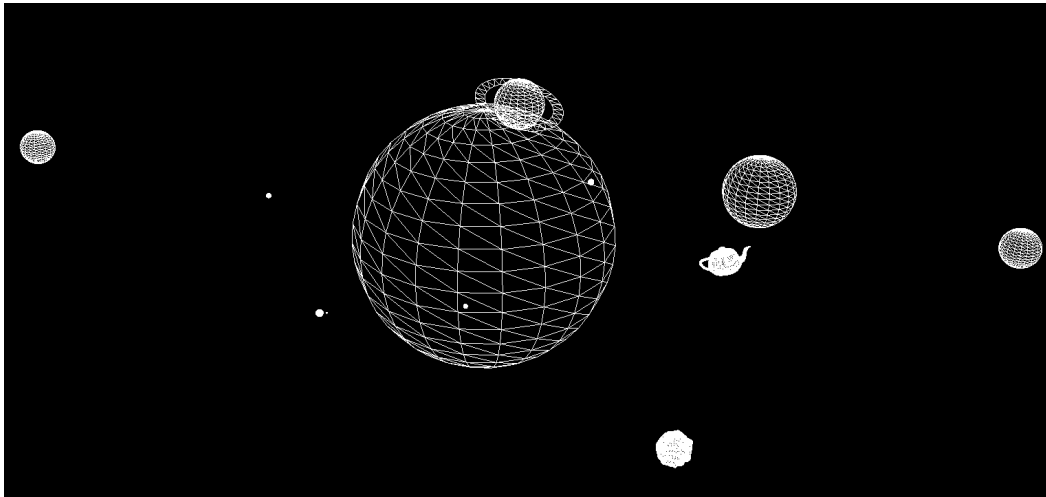


Figura 4.1: Sistema solar

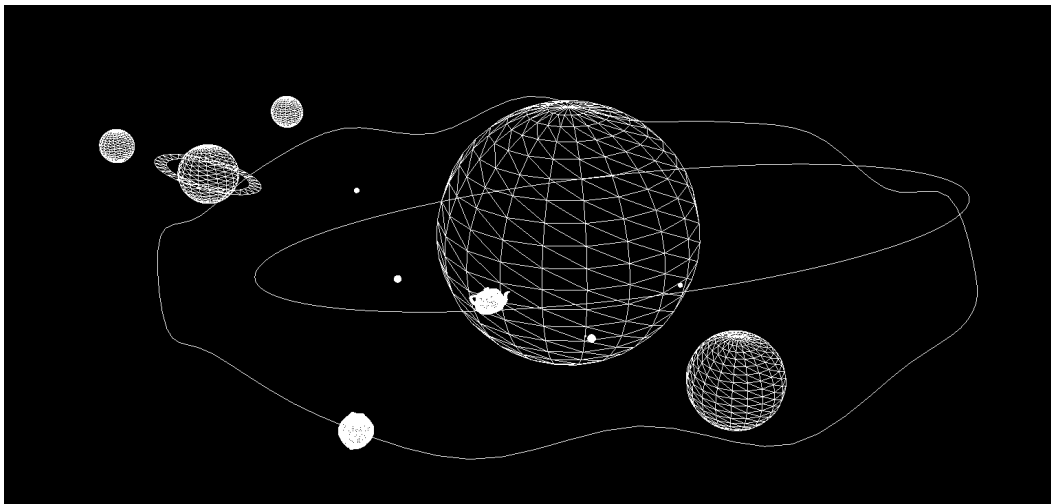


Figura 4.2: Sistema solar com as trajetórias dos cometas

## 5 Conclusão

Ao longo da elaboração desta terceira fase do projeto, foi possível aplicar na prática alguns conceitos de computação gráfica, nomeadamente superfícies de Bezier e curvas de Catmull-Rom, o que ajudou a consolidar os nossos conhecimentos acerca do funcionamento destas duas técnicas.

Acerca da concretização do trabalho realizado, encontramos-nos satisfeitos, uma vez que conseguimos implementar todas as funcionalidades pedidas para esta fase bem como alguns extras, tais como a criação de um cometa minimamente realista com *patches* de Bezier que permite adicionar um toque de realidade à nossa animação. Para além disso, também implementámos a movimentação livre da câmara que permite visualizar melhor a animação sem estarmos restritos às coordenadas esféricas.

Por fim, consideramos que reunimos ao longo desta fase os elementos necessários para poder progredir para a última fase do projeto.

# Anexos

## Anexo 1 - Como utilizar o *Generator*

### Gerar o cone

```
./generator.exe cone "raio" "altura" "slices" "stacks" "path ficheiro .3d"
```

### Gerar o cubo/caixa

```
./generator.exe box "dimensão" "divisões" "path ficheiro .3d"
```

### Gerar a esfera

```
./generator.exe sphere "raio" "slices" "stacks" "path ficheiro .3d"
```

### Gerar o plano

```
./generator.exe plane "dimensão" "divisões" "path ficheiro .3d"
```

### Gerar o anel

```
./generator.exe ring "raio interno" "raio externo" "slices" "path ficheiro .3d"
```

### Gerar superfícies de Bezier com um ficheiro de *patches*

```
./generator.exe patch "tesselação" "path do ficheiro com os patches" "path ficheiro .3d"
```

## Anexo 2 - Como iniciar o *Engine*

Para iniciar a execução do *Engine*, basta executar o seguinte comando:

```
./engine.exe "caminho para o ficheiro XML"
```

## Anexo 3 - Comandos do *Engine*

### Comandos da câmara no modo *SPHERICAL*

- A** Movimenta a câmara para a esquerda.
- D** Movimenta a câmara para a direita.
- W** Movimenta a câmara para cima.
- S** Movimenta a câmara para baixo.
- ↑ Aproxima a câmara da origem.
- ↓ Afasta a câmara da origem.

### Comandos da câmara no modo *FREE*

- A** Movimenta a câmara para a esquerda em linha reta.
- D** Movimenta a câmara para a direita em linha reta.
- W** Movimenta a câmara para a frente em linha reta.
- S** Movimenta a câmara para trás em linha reta.
- +** Movimenta a câmara para cima em linha reta.
- Movimenta a câmara para baixo em linha reta.
- ↑ Movimenta o olhar da câmara para cima.
- ↓ Movimenta o olhar da câmara para baixo.
- ← Movimenta o olhar da câmara para a esquerda.
- ⇒ Movimenta o olhar da câmara para a direita.

### Restantes comandos

- F** Preenche a primitiva.
- L** Exibe as linhas que definem a primitiva.
- P** Exibe os pontos que constituem a primitiva.
- C** Exibe as curvas de Catmull-Rom, se existirem.
- V** Alterna o modo de movimentação da câmara (*FREE* ou *SPHERICAL*).

## Anexo 4 - Significado das informações que aparecem no título da janela do *Engine*

Quando se utiliza o *Engine*, no título da janela aparecerão algumas informações. Apresenta-se de seguida um exemplo:

```
FPS: 74.95, PCAM: (72.73,70.49,103.93), LA: (1.32,0.49,0.15), alpha = 3.74, beta = -0.51, CamMode: FREE
```

Significado dos valores existentes:

- **FPS:** *Frames* por segundo.

- **PCAM**: Posição da câmara.
- **LA**: Ponto *lookAt* da câmara.
- **alpha**: No modo *SPHERICAL*, este valor indica o ângulo que a câmara faz com o semieixo positivo  $Z$ . Já no modo *FREE* é o ângulo  $\alpha$  indicado figura 3.14.
- **beta**: No modo *SPHERICAL*, este valor indica o ângulo que a câmara faz com o plano  $XZ$ . Já no modo *FREE* é o ângulo  $\beta$  indicado figura 3.14.
- **CamMode**: Modo de movimentação da câmara (*SPHERICAL* ou *FREE*).

## Anexo 5 - Estrutura do projeto

Nesta fase, o projeto tem a seguinte estrutura:

```
Fase_3
├── configs (Ficheiros de configuração XML)
├── outputs (Diretoria que contém os ficheiros .3d)
├── patches (Diretoria que contém os ficheiros com patches de Bezier)
├── src (Diretoria com todo o código fonte)
│   ├── engine (Código fonte do engine)
│   ├── generator (Código fonte do generator)
│   ├── tinyXML (Biblioteca para processar ficheiros XML)
│   ├── utils (Diversas bibliotecas que contém estruturas de dados e funções auxiliares)
│   └── CMakeLists.txt (ficheiro CMake para compilar as duas aplicações)
└── comet.py (Programa feito em Python para gerar um cometa)
```