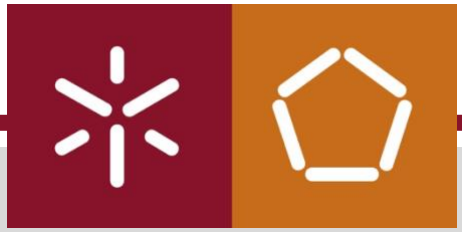
UNIVERSIDADE DO MINHO



Mestrado Integrado em Engenharia Informática

PROJETO DE COMPUTAÇÃO GRÁFICA FASE 3 : CURVAS, SUPERFÍCIES CÚBICAS E VBOS.

maio, 2020




GRUPO 10
Carolina Cunha, A80142 | Hugo Faria, A81283 | João Diogo Mota, A80791 | Rodolfo Silva, A81716

ÍNDICE

ÍNDICE DE FIGURAS

BREVE DESCRIÇÃO DO ENUNCIADO PROPOSTO

Prevê-se, com este trabalho, o desenvolvimento de um mecanismo 3D baseado em mini gráficos de cenas e fornecer exemplos de uso que mostrem o seu potencial. Este trabalho será dividido em quatro fases distintas, de modo a facilitar a elaboração do mesmo. A conceção e desenvolvimento do trabalho seguirá uma abordagem suportada pela ferramenta *OpenGL* e a linguagem C++.

A terceira fase do projeto consistirá no aperfeiçoamento do gerador, tornando-o capaz de criar um novo tipo de modelo baseado nos *patches* de *Bezier*. Deste modo, espera-se que o gerador receba como parâmetros o nome de um ficheiro onde se encontram definidos os pontos de controlo de *Bezier*, bem como o nível de *tesselation* necessário. O ficheiro resultante conterá, assim, uma lista dos triângulos para desenhar a superfície.

No que diz respeito ao *engine*, é pretendida a extensão dos elementos de translação e rotação, cujo objetivo visa a realização de animações com base nas curvas de *Catmull-Rom*. Deve ser tida em consideração a necessidade de fornecer um conjunto de pontos para definir a curva cúbica, assim como o número de segundos para executar uma rotação completa em torno do eixo especificado.

Desta forma, recorrendo aos *patches* de *Bezier*, solicita-se a construção de um *teapot* que retratará, posteriormente, o cometa introduzido no Sistema Solar. Por sua vez, a utilização das curvas de *Catmull-Rom* irá conferir o aspeto dinâmico do Sistema Solar.

Por fim, é também pedido o desenho de modelos com VBOs (*Vertex Buffer Objects*).

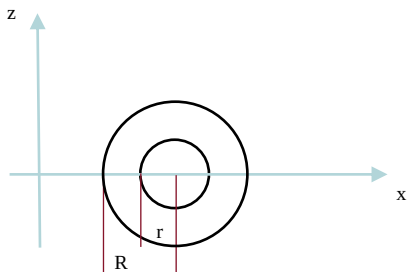
PRIMITIVAS GRÁFICAS

Ainda nesta fase, foram adicionadas duas novas primitivas gráficas, de modo a possibilitar a representação dos anéis dos planetas gasosos, e da cintura de asteroides que se encontra entre as órbitas de Marte e Júpiter.

ARCO

A criação da primitiva arco deve-se à necessidade da representação dos anéis dos planetas gasosos. Para que seja possível a sua criação, são requeridos como argumentos o seu raio interno, raio externo e o número de fatias do arco.

A sua representação é muito semelhante à da base de um cilindro, diferenciando no facto de que os pontos não se conectam ao centro da base, mas sim ao raio interno da figura.



Deste modo, é, primeiramente, necessário ter em consideração o número de fatias definido, de forma a dividir a figura em fatias de igual dimensão, obtendo assim o ângulo para cada fatia.

```
float sliceStep = (float)(2 * M_PI) / slices;
```

Após efetuada a divisão do número de fatias e recorrendo aos raios interno e externo, foram calculados os vértices dos triângulos que irão representar a figura.

O desenho dos triângulos é realizado através de um ciclo que, para cada número de fatias, vai calcular os vértices do triângulo seguinte.

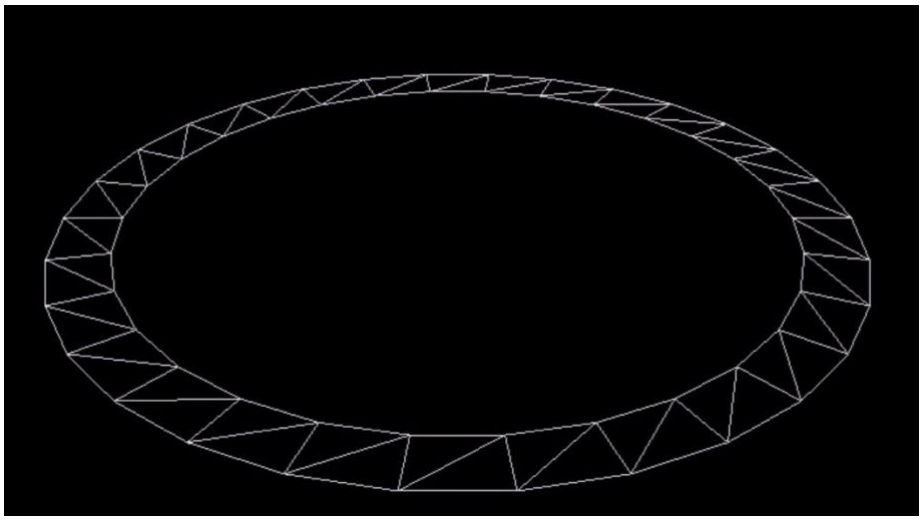


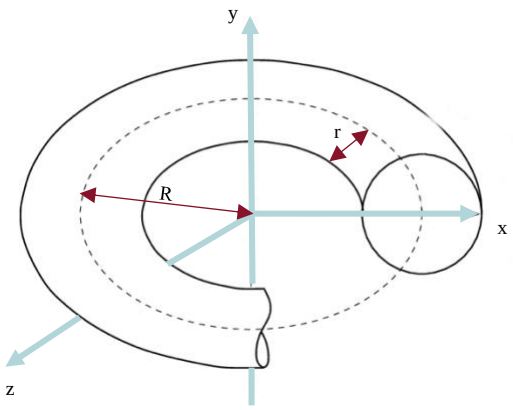
Figura 1: Representação do Arco em OpenGL

CINTO

A primitiva cinto recai na intenção de aplicar um efeito de três dimensões à cintura de asteroides, contrariamente ao necessário para os anéis dos planetas.

Pode, novamente, associar-se esta primitiva à primitiva do cilindro, dado que esta pode ser vista como um cilindro cujas bases foram unificadas.

Para que seja possível a sua representação, são requeridos como argumentos o seu raio interno, definindo a sua espessura; o raio externo, retratando a distância da figura à origem do referencial; o número de fatias do cinto e o número de divisões do mesmo.



Assumindo que 0 < sliceAngle < 2π, responsável pela construção em n fatias do cinto, 0 < stackAngle < 2π, permitindo dividir o cinto em m divisões.

A construção do cinto na sua totalidade exige a criação de dois ciclos aninhados capazes de iterar pelo número de fatias e divisões indicadas como parâmetros. Assim, para cada iteração de divisões, é calculado um novo valor de **stackAngle**,

```
float stackAngle = i * ((2 * M_PI) / stacks);
```

e a cada iteração do número de fatias, é calculado um novo valor de **sliceAngle**,

```
float sliceAngle = j * ((2 * M_PI) / slices);
```

As expressões matemáticas utilizadas para calcular as coordenadas de cada vértice dos triângulos são:

$$x = \cos(stackAngle) * (r * \cos(sliceAngle) + R)$$
$$y = \sin(stackAngle) * (r * \cos(sliceAngle) + R)$$
$$z = r * \sin(sliceAngle)$$

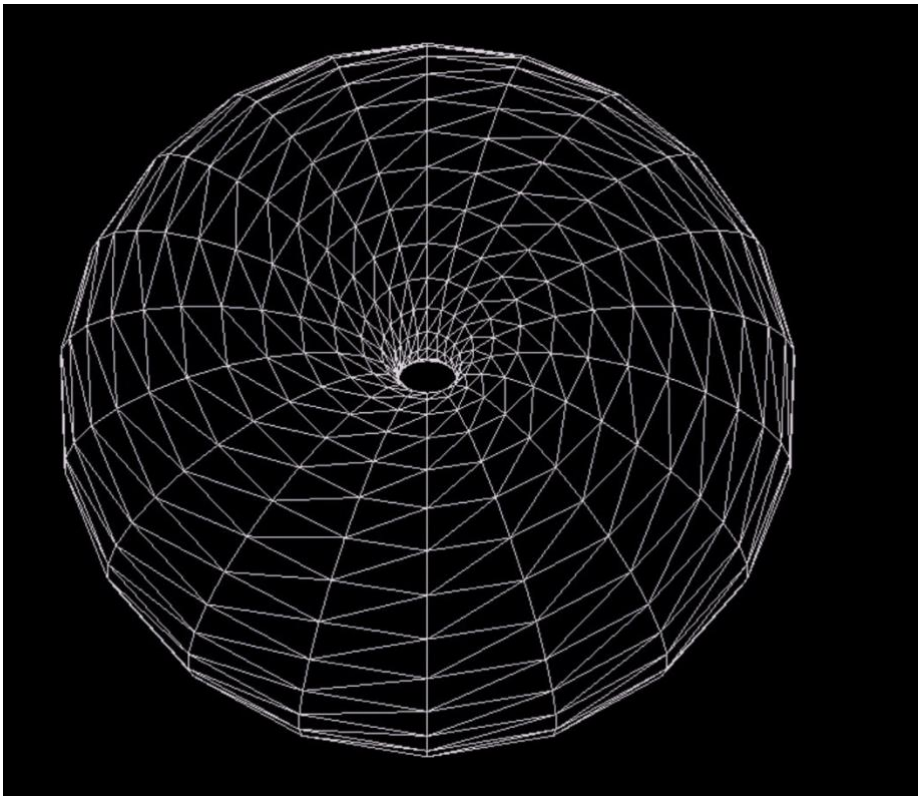


Figura 2: Representação do Cinto em OpenGL

GENERATOR

PATCHES DE BEZIER

Nesta terceira fase, pretende-se a construção de um *teapot* a partir de *patches* de *Bezier*.

Numa primeira etapa, é crucial a extração da informação presente no ficheiro em formato *Patch*, pelo que foi desenvolvido um *parser* que guarda a respetiva informação.

```
int parserBezier(std::string patch);
```

Deste modo, serão guardados o número de *patches* (nPatches), indicado na primeira linha do ficheiro, bem como os índices de pontos correspondentes a cada *patch*, na forma de uma matriz, *indexes[p][ ],* em que p é o número de *patches* existentes. Serão ainda armazenados todos os pontos de controlo (nPontos) em três *arrays*, um por coordenada, isto é, no *array* pontosX estarão armazenadas as coordenadas x de todos os pontos de controlo, pelo que o mesmo se aplica aos *arrays* pontosY e pontosZ.

```
fscanf(f, "%d\n", &nPatches);
        patches = nPatches;
```

```
indexes = (int**)malloc(sizeof(int*) * patches);
        for (int i = 0; i < patches; i++) {
                indexes[i] = (int*)malloc(sizeof(int) * 16);
                fgets(line, SIZE_BUFFER, f);
                char* token = NULL;
                for (j = 0, token = strtok(line, ", "); token && j < 16; token = strtok(NULL, ", "), j++) {
                        indexes[i][j] = atoi(token);
                }
        }
```

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

pontosX = (float*)malloc(sizeof(float) * nPontos);
pontosY = (float*)malloc(sizeof(float) * nPontos);
pontosZ = (float*)malloc(sizeof(float) * nPontos);

for (int k = 0; k < nPontos; k++) {
        fgets(line, SIZE_BUFFER, f);
        pontosX[k] = atof(strtok(line, ", "));
        pontosY[k] = atof(strtok(NULL, ", "));
        pontosZ[k] = atof(strtok(NULL, ", "));
}
```

Dado por terminado o *parser* do ficheiro fornecido, dá-se início ao processo de gerar a figura pretendida. Para que o gerador do *teapot* seja passível de ser criado, e tendo em consideração a necessidade de utilização de 16 pontos de controlo para definir uma superfície de *Bezier*, deve ter-se em consideração as seguintes equações.

$$M = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad (1)$$

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

Através da equação (2), é possível obter um ponto da superfície de *Bezier*. A função que permite obter as coordenadas de um ponto é a seguinte.

```
void getBezierPoint(float u, float v, float* px, float* py, float* pz, float* normalV);
```

Os *arrays* recebidos como argumento dizem respeito às coordenadas x, y e z de cada um dos pontos de controlo de um *patch*.

Foram, então, calculadas as matrizes e os vetores necessários à concretização da equação (2), onde a matriz dos pontos de controlo é constituída pelos *arrays* recebidos como argumento.

Após a implementação de uma função que permita o cálculo dos pontos, é indispensável a criação de uma função responsável pela obtenção dos vértices dos triângulos.

```
int genBezier(std::string name, int tessellation);
```

Assim, para cada *patch*, vão ser preenchidos três *arrays* (arrX, arrY, arrZ) com os pontos de controlo, pelo que cada *array* contém a coordenada x, y e z de cada ponto, respetivamente.

```
for (int i = 0; i < patches; i++) {
        for (int j = 0; j < 16; j++) {
                arrX[j] = pontosX[indexes[i][j]];
                arrY[j] = pontosY[indexes[i][j]];
                arrZ[j] = pontosZ[indexes[i][j]];
        }
```

De seguida, percorrendo u e v por valores de tecelagem (note-se que, quanto maior o nível de tecelagem, maior será o detalhe da figura), serão armazenados nos *arrays* p0, p1, p2 e p3 os pontos obtidos através da função *getBezierPoint*. Estes pontos serão posteriormente escritos no respetivo ficheiro *.3d*, pela ordem obtida através da regra da mão direita.

```
for (int u = 0; u < tessellation; u++) {
        float p0[3];
        float p1[3];
        float p2[3];
        float p3[3];

        for (int v = 0; v < tessellation; v++) {
                getBezierPoint(u / ((float)tessellation), v / ((float)tessellation), arrX, arrY, arrZ, p0);
                getBezierPoint((u + 1) / ((float)tessellation), v / ((float)tessellation), arrX, arrY, arrZ, p1);
                getBezierPoint(u / ((float)tessellation), (v + 1) / ((float)tessellation), arrX, arrY, arrZ, p2);
                getBezierPoint((u + 1) / ((float)tessellation), (v + 1) / ((float)tessellation), arrX, arrY, arrZ, p3);

                myfile << p0[0] << " " << p0[1] << " " << p0[2] << '\n';
                myfile << p2[0] << " " << p2[1] << " " << p2[2] << '\n';
                myfile << p1[0] << " " << p1[1] << " " << p1[2] << '\n';


                myfile << p1[0] << " " << p1[1] << " " << p1[2] << '\n';
                myfile << p3[0] << " " << p3[1] << " " << p3[2] << '\n';
                myfile << p2[0] << " " << p2[1] << " " << p2[2] << '\n';
        }
}
```

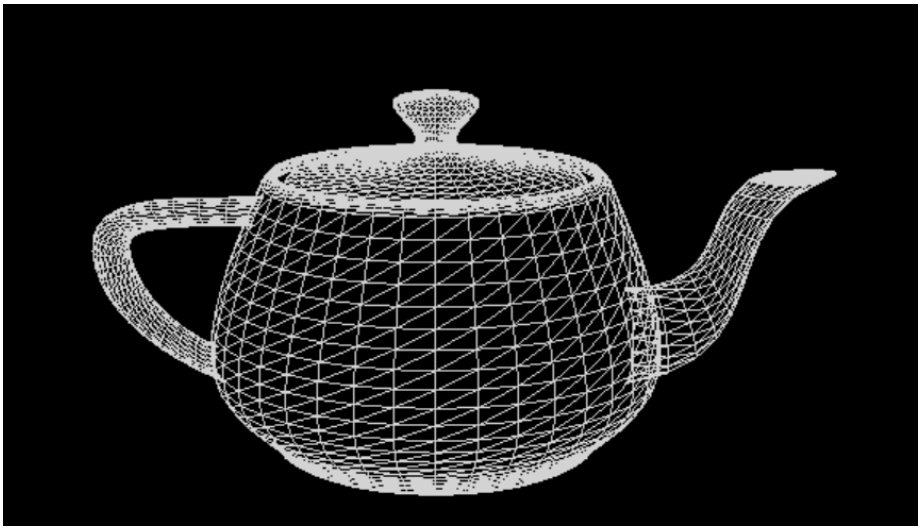


Figura 3: Representação do Teapot com Patches de Bezier

ENGINE

CATMULL-ROM

Igualmente nesta fase, é pretendida a extensão dos elementos de translação e rotação, cujo objetivo visa a realização de animações com base nas curvas de *Catmull-Rom*. Assim sendo, partir-se-á das seguintes equações:

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

$$P'(t) = [3t^2\ 2t\ 1\ 0]\begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}\begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} \quad (4)$$

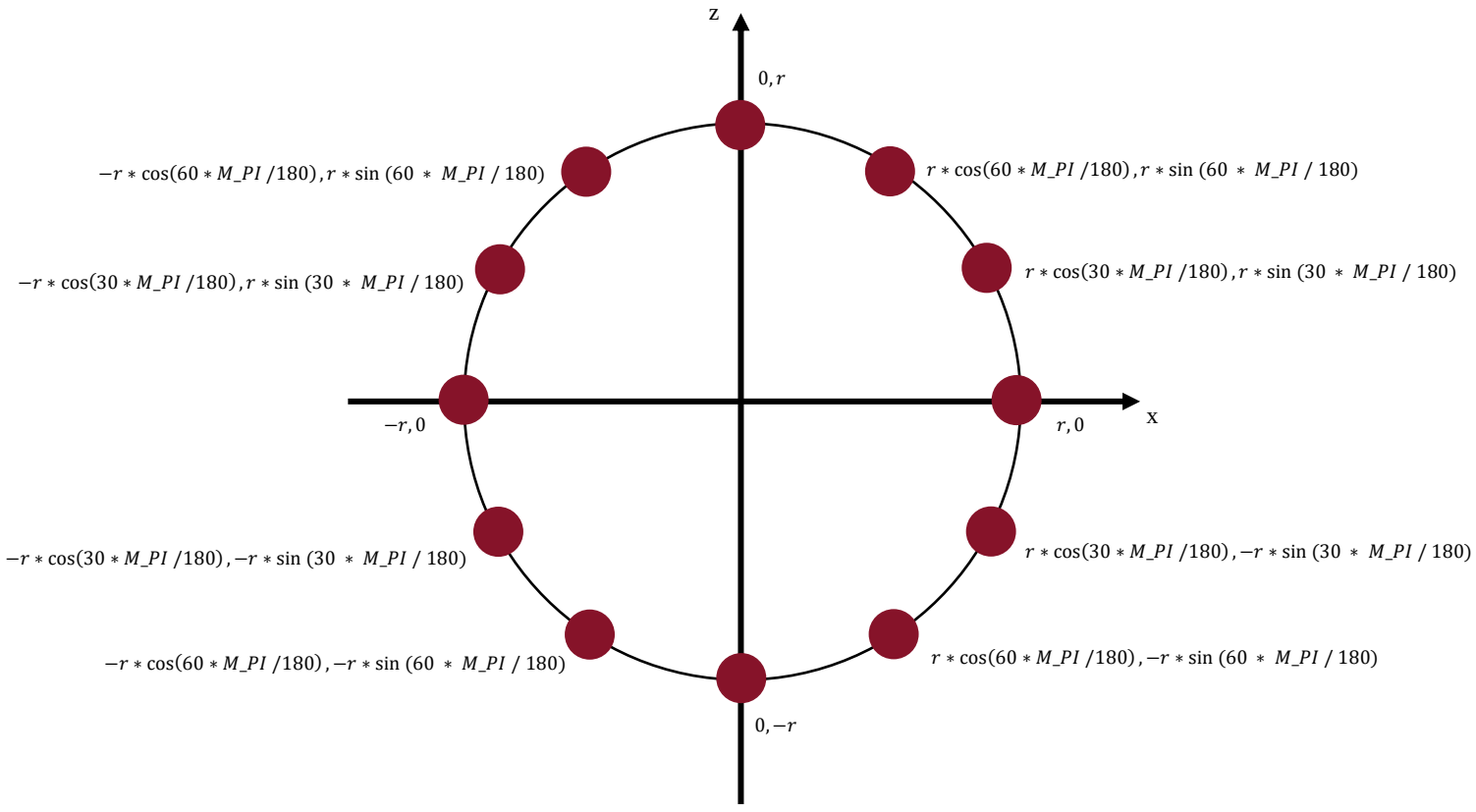
Através destas equações, sabe-se que, num determinado instante *t*, P(t) representa a posição de um objeto a "andar" ao longo da curva e P'(t) o vetor tangente à curva. A aplicação destas equações permitirá o dinamismo do Sistema Solar.

Para este efeito, foram implementadas as funções *getCatmullRomPoint* e *getGlobalCatmullRomPoint*, que permitem a obtenção das coordenadas dos pontos na curva e a trajetória definida por estes.

Por sua vez, a função *planetTranslate* possibilita o desenho da órbita dos corpos celestes, recolhendo os pontos da estrutura de dados de uma figura dada como argumento, e aplicando sobre esses a função *getGlobalCatmullRomPoint*. Considerando que as curvas são compostas por 100 pontos, este processo vai ser iterado para cada ponto da curva, pelo que o ponto resultante será posteriormente desenhado, construindo assim a trajetória desejada.

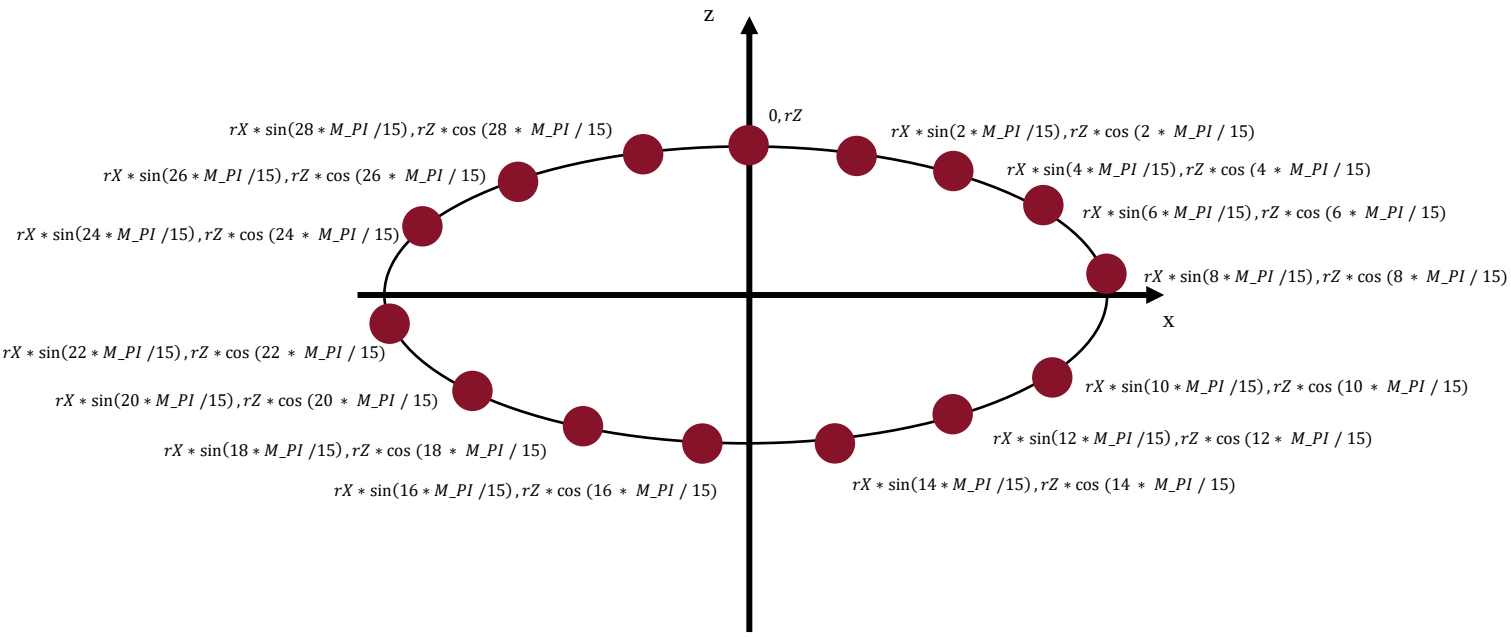
Ainda nesta função, está definido o modo de alocação de uma primitiva gráfica num determinado ponto da trajetória, num dado instante. Deste modo, através da função *getGlobalCatmullRomPoint*, obtém-se o ponto na curva onde terá que ser realizada a translação da figura. Assim, a cada execução da função *renderScene*, o tempo decorrido irá aumentar, conduzindo à translação da primitiva gráfica ao longo da respetiva trajetória.

Na figura que se segue, serão apresentados os doze pontos de controlo da curva para a obtenção das órbitas dos planetas e respetivos satélites naturais, assumindo o valor da coordenada Y como sendo 0.



No que diz respeito à trajetória do cometa inserido no Sistema Solar, esta encontra-se também definida na função *planetTranslate*, funcionando de igual modo.

Na figura abaixo apresentada, encontram-se os quinze pontos de controlo da curva para a obtenção da órbita do cometa, assumindo o valor da coordenada Y como sendo 0.



Rotação

Relativamente à rotação dos corpos celestes, foi necessária a adição de uma nova variável temporal, indicando o tempo que a figura deverá demorar a percorrer a curva na sua totalidade.

Assim sendo, a rotação dos planetas e respetivas Luas encontra-se na função *planetRotate* onde, dada a estrutura de dados de uma figura passada como argumento da função, vai ser calculado o ângulo da figura a cada instante de tempo. Este ângulo é calculado da forma,

```
angle[figurasDesenhadas] += 360 / (time * 1000);
```

onde *time* é a variável temporal introduzida no ficheiro *xml*, e pretende-se que este tempo (em milissegundos) seja uniformemente dividido por 360°, completando assim uma rotação total.

Por outro lado, a rotação do cometa encontra-se intrínseca à função planetTranslate, dado que é pretendido que este tenha a direção da trajetória que segue. Deste modo, é requerido que se apliquem as transformações necessárias para que o *teapot* viaje ao longo da curva, orientado de acordo com a derivada.

```
getGlobalCatmullRomPoint(tt[figurasDesenhadas], pos, deriv, curvePoints, size);

float x[3] = { deriv[0],deriv[1],deriv[2] };
normalize(x);

float z[3];
float y[3];
cross(x, prevY, z);
normalize(z);
cross(z, x, y);
normalize(y);
prevY[0] = y[0]; prevY[1] = y[1]; prevY[2] = y[2];

float rotMatrix[16];
buildRotMatrix(x, y, z, rotMatrix);

if (timeTranslate < numDeFiguras) {
     double ttt = glutGet(GLUT_ELAPSED_TIME);
     elapsedTime[figurasDesenhadas] = 1 / (nodo.getTempoTr() * 100);
     timeTranslate++;
}
tt[figurasDesenhadas] += elapsedTime[figurasDesenhadas];
glTranslatef(pos[0], pos[1], pos[2]);
glMultMatrixf(rotMatrix);
```

VBOs

Um VBO pode ser visto como um *array* residente na memória da placa gráfica. Este *array*, inicialmente criado na memória central, poderá posteriormente ser utilizado para desenhar geometria com um número reduzido de instruções. Com a utilização de VBOs, a comunicação entre o CPU e o GPU é extremamente reduzida na função *renderScene*. [1]

Após extrair os vértices presentes na estrutura de dados de uma figura, é preenchido o vetor *pontos* com esses mesmos vértices. Recorrendo às funções do *OpenGL*, o *buffer figuras* previamente gerado, será preenchido por esses vértices.

```
for (int mainNodeAt = 0; figurasInMemory < numDeFiguras; mainNodeAt++) {
        xmlData ficheiro = figs.at(mainNodeAt);
        std::vector<triangulo> figura = ficheiro.getPontos();

        for (unsigned int k = 0; k < figura.size(); k++) {
            triangulo tr = figura.at(k);
            pontos.push_back(tr.getVerticeX1());
            pontos.push_back(tr.getVerticeY1());
            pontos.push_back(tr.getVerticeZ1());
            pontos.push_back(tr.getVerticeX2());
            pontos.push_back(tr.getVerticeY2());
            pontos.push_back(tr.getVerticeZ2());
            pontos.push_back(tr.getVerticeX3());
            pontos.push_back(tr.getVerticeY3());
            pontos.push_back(tr.getVerticeZ3());
        }

        int size = pontos.size();
        tamanhoFiguras[figurasInMemory] = size;
        glGenBuffers(1, &figuras[figurasInMemory]);
        glBindBuffer(GL_ARRAY_BUFFER, figuras[figurasInMemory]);
        glBufferData(GL_ARRAY_BUFFER, sizeof(float) * size, &pontos[0], GL_STATIC_DRAW);
        pontos.clear();
        figurasInMemory++;
        toBuffers(ficheiro.getFilhos(), &figurasInMemory);
    }
}
```

O desenho da informação armazenada é realizado do seguinte modo:

```
glBindBuffer(GL_ARRAY_BUFFER, figuras[figurasDesenhadas]);
glVertexPointer(3, GL_FLOAT, 0, 0);
glDrawArrays(GL_TRIANGLES, 0, tamanhoFiguras[figurasDesenhadas]);
```

SISTEMA SOLAR

No que diz respeito à animação do Sistema Solar, foi necessário calcular os períodos de rotação e de translação de cada planeta e respetivos satélites naturais. Para que tal fosse possível de forma minimamente realista, teve-se por base o período de translação de Mercúrio.

Deste modo, tem-se:

$$tTranslacao = \frac{pT * 5}{87{,}97}$$

Assim, para cada corpo, o seu tempo de translação é obtido através de uma regra de três, onde se tem que 87,97 dias correspondem a 5 segundos.

Para o cálculo do período de rotação, foi necessária a multiplicação por um fator de 50, uma vez que os períodos de rotação são muito inferiores aos períodos de translação de cada corpo. Assim, tem-se:

$$tRotacao = \frac{pR * 5}{87{,}97} * 50$$

Planeta	Translação	Rotação
Mercúrio	5	166.5
Vénus	12.77	-690.5
Terra	20.76	3
Marte	39.05	3
Júpiter	246.20	1
Saturno	611.54	1
Úrano	1433.11	1.5
Neptuno	3421.09	2
Plutão	5150.30	18

Quanto aos satélites naturais, o seu período de translação é idêntico ao período de translação do respetivo planeta em torno do qual orbitam. Relativamente aos períodos de rotação e revolução, estes são maioritariamente síncronos, pelo que período de rotação corresponde ao tempo que o satélite natural demora a dar uma volta ao planeta, e o período de revolução ao tempo que o satélite demora a dar uma volta sobre o seu próprio eixo.

Satélite Natural	Translação	Rotação	Revolução
Lua	20.76	79.5	79.5
Phobos	39.05	1	1
Deimos	39.05	3.5	3.5
Io	246.20	5	5
Europa	246.20	10	10
Calisto	246.20	47.5	47.5
Genimedes	246.20	20.5	20.5
Mimes	611.54	2.5	2.5
Encélado	611.54	4	4
Tétis	1433.11	5.5	5.5
Ariel	1433.11	7	7
Puck	1433.11	N/A	N/A
Tritão	3421.09	-16.5	-16.5
Larissa	3421.09	N/A	N/A

MOVIMENTO DA CÂMARA

Similarmente à fase anterior, de modo a facilitar a verificação da concretização das transformações geométricas, foram alterados os parâmetros relativos à posição da câmara.

Para este efeito, e uma vez que é necessária a utilização de coordenadas esféricas, foi utilizada a função sphericall2cartesian.

```
void sphericall2cartesian() {
    camX = radius * sin(alfa) * cos(beta);
    camZ = radius * cos(alfa) * cos(beta);
    camY = radius * sin(beta);
}
```

Assim, tem-se

```
gluLookAt(camX + refX , camY + refY , camZ + refZ,
          refX , refY , refZ,
          0.0f , 1.0f , 0.0f);
```

Desta forma, é possível, através de teclas especiais, alterar a posição da câmara, modificando os valores de alfa e de beta.

```
case(GLUT_KEY_RIGHT):
    alfa -= 0.01; break;
case(GLUT_KEY_LEFT):
    alfa += 0.01; break;
case(GLUT_KEY_UP):
    beta += 0.01;
    if (beta > 1.5f)
        beta = 1.5f;
    break;
case(GLUT_KEY_DOWN):
    beta -= 0.01;
    if (beta < -1.5f)
        beta = -1.5f;
    break;

case GLUT_KEY_PAGE_DOWN: radius -= 5.0f;
    if (radius < 1.0f)
        radius = 1.0f;
    break;

case GLUT_KEY_PAGE_UP: radius += 5.0f; break;
```

Por outro lado, através das teclas regulares, é possível alterar o ponto para onde a câmara está a olhar, modificando os valores de refX, refY e refZ.

```
case 'w': {
    refZ -= 5 * cos(alfa);
    refX -= 5 * sin(alfa);
} break;
case 's': {
    refZ += 5 * cos(alfa);
    refX += 5 * sin(alfa);
} break;

case 'd': {
    refX += 5 * cos(alfa);
    refZ += 5 * (-sin(alfa));
} break;
case 'a': {
    refX -= 5 * cos(alfa);
    refZ -= 5 * (-sin(alfa));
} break;
case 'q': {
    refY += 3;
} break;
case 'e': {
    refY -= 3;
} break;
case '-': {
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
} break;
case '.': {
    glPolygonMode(GL_FRONT_AND_BACK, GL_POINT);
} break;
case ',': {
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
} break;
```

CONCLUSÕES

A realização da terceira fase do projeto do Sistema Solar permitiu que o grupo solidificasse conhecimento sobre alguns dos tipos de curvas e superfícies cúbicas mais habituais. A introdução de métodos de melhoria de eficiência (VBOs) possibilitou o melhor entendimento sobre a relevância da sua aplicação, verificando-se assim o aumento do desempenho com a renderização imediata.

Ao longo da realização das tarefas solicitadas para esta fase, o grupo deparou-se com algumas dificuldades no que diz respeito à construção de figuras a partir de um ficheiro em formato *Patch*, dado tratar-se da primeira interação com as mesmas.

A implementação do Sistema Solar dinâmico, bem como o êxito na introdução de *patches* de *Bezier*, *VBOs* e curvas de *Catmull-Rom* possibilitou o que o grupo considera ser o sucesso da terceira fase deste projeto, ansiando pela inserção de texturas e iluminação às figuras existentes, que levarão a um resultado mais apelativo e realista do Sistema Solar.

BIBLIOGRAFIA

[1] Ramires, A., n.d. Vertex Buffer Objects - Notas Para A Componente Prática De Computação Gráfica.

ANEXOS

Apresentam-se em seguida a representação gráfica do Sistema Solar dinâmico, com a introdução dos anéis e cintura de asteroides, bem como as órbitas de cada planeta em torno do Sol e dos satélites naturais de cada planeta.

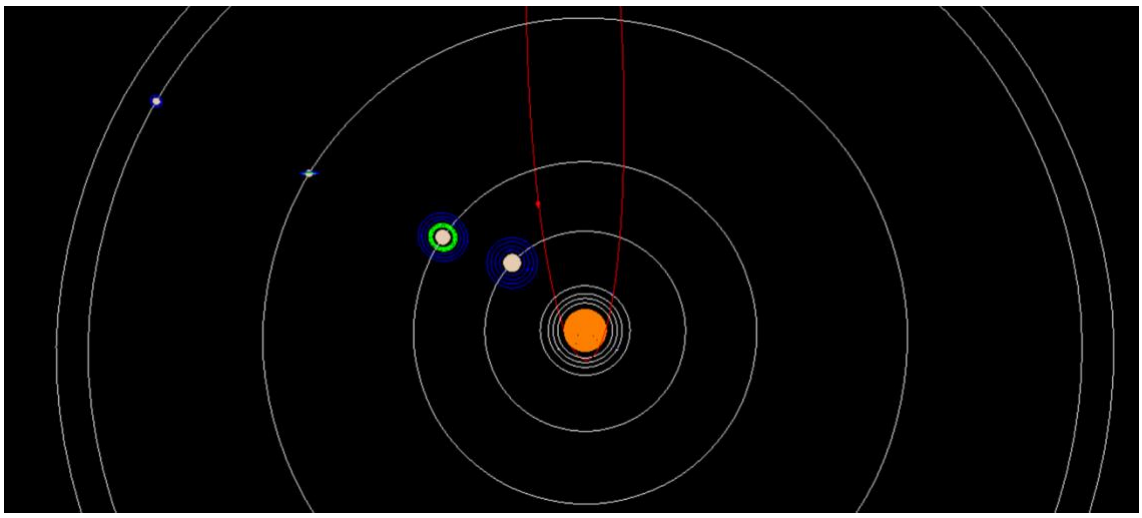


Figura 4: Sistema Solar Dinâmico 1

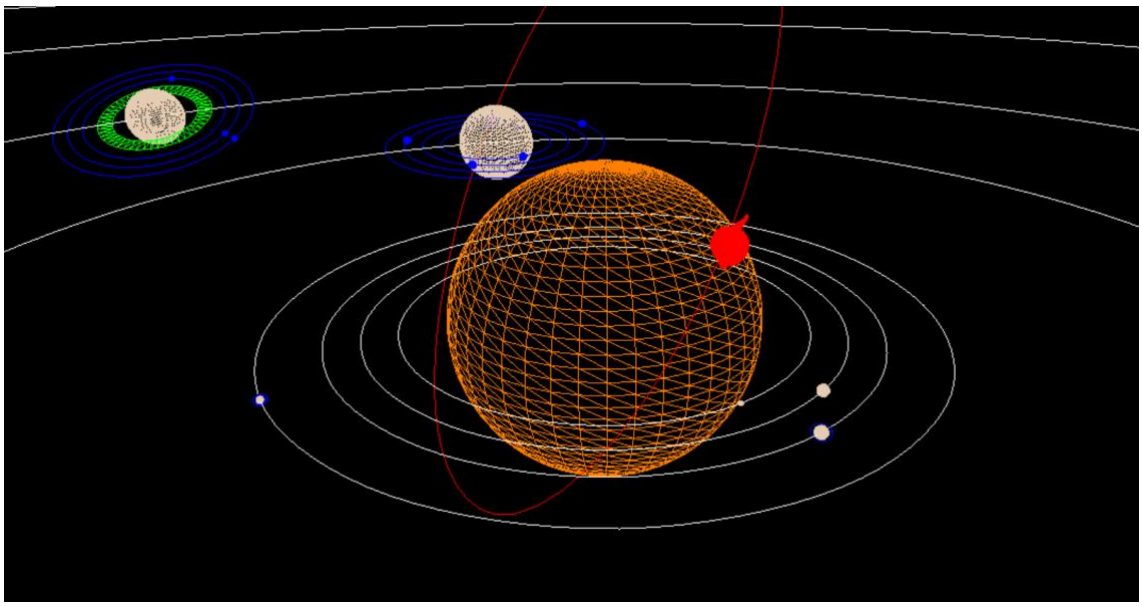


Figura 5: Sistema Solar Dinâmico 2

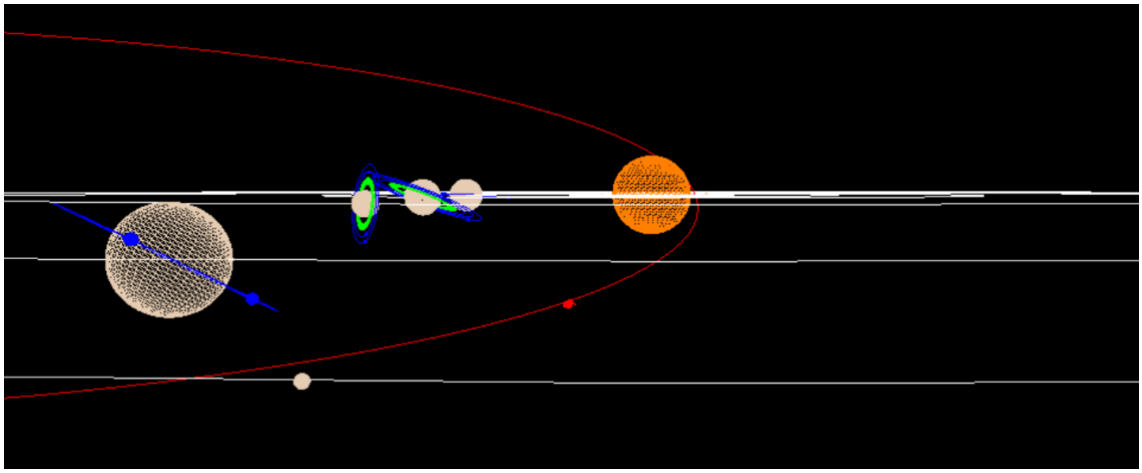


Figura 6: Sistema Solar 3

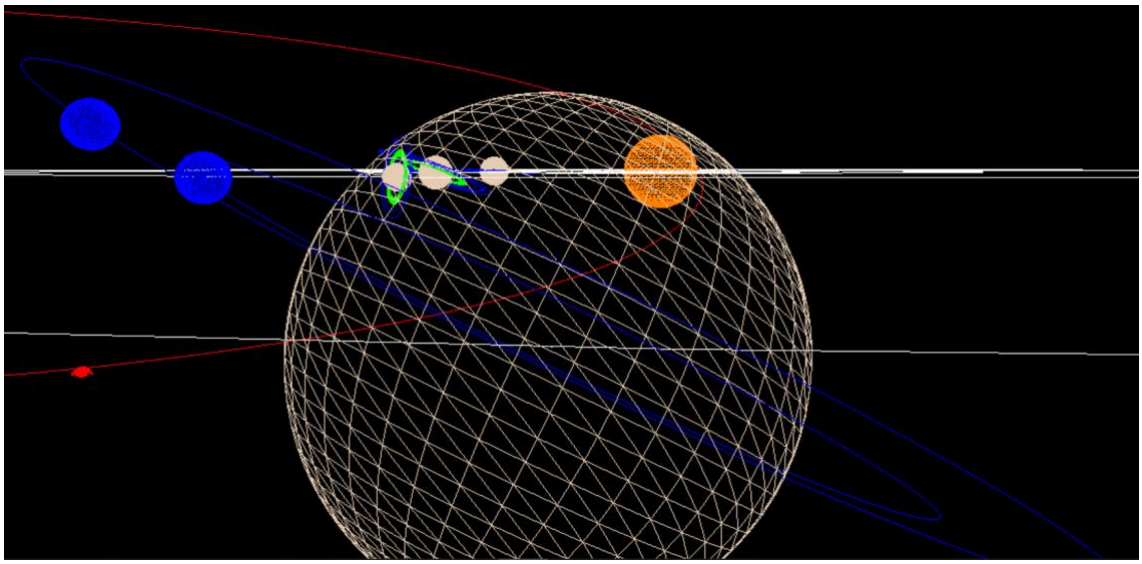


Figura 7: Sistema Solar 4

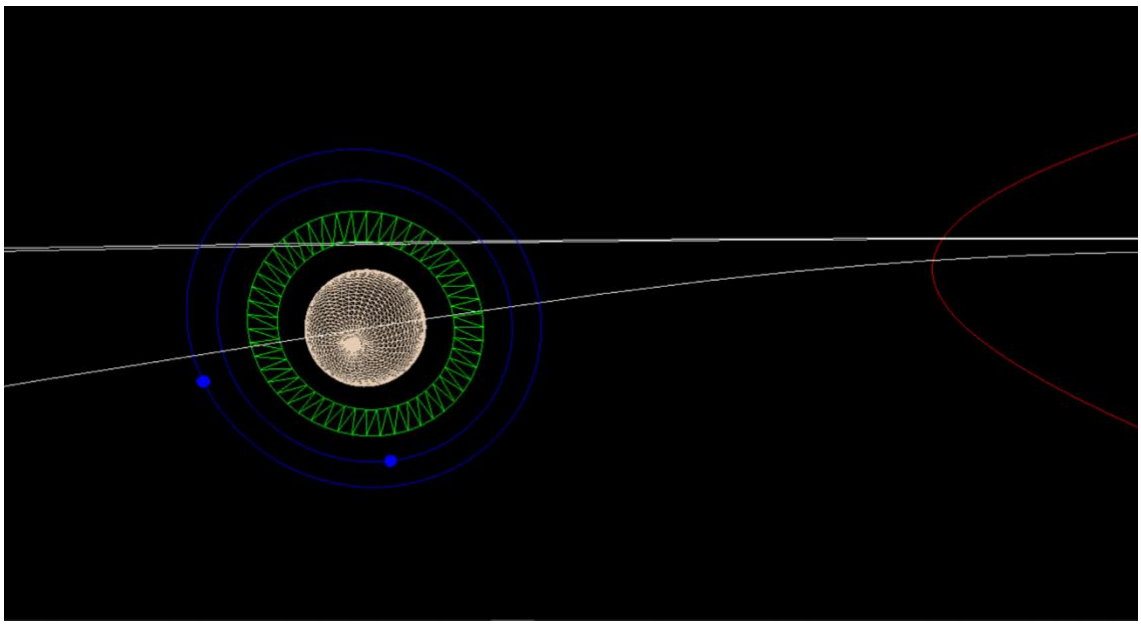


Figura 8: Sistema Solar 5