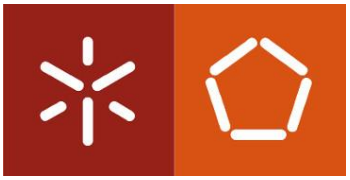


UNIVERSIDADE DO MINHO

DEPARTAMENTO DE INFORMÁTICA

TRABALHO PRÁTICO – Fase 3

Curves, Cubic Surfaces and VBOs

Computação Gráfica
Maio de 2021

Autores:

Ana Filipa Pereira A89589

Carolina Santejo A89500

Raquel Costa A89464

Sara Marques A89477

Índice

Índice de Figuras

Introdução

Esta terceira fase do projeto da cadeira de Computação Gráfica, consistiu na continuação do trabalho realizado em fases anteriores, de forma a melhorar algumas características já desenvolvidas.

Em primeiro lugar, foi preciso considerar que o sistema solar passou a ser dinâmico, contrariamente à fase 1, e que as transformações *translate* e *rotate* possuíam agora, um tempo associado. É de realçar que as translações são feitas recorrendo a curvas de *Catmull-Rom*. Para além disto, foi adicionada a primitiva *teapot*, baseada em *Bézier patches*.

De maneira a melhorar a performance do trabalho, foram feitas alterações para que as primitivas passassem a ser desenhadas com recurso a VBO's.

Assim, ao longo deste relatório serão detalhadas todas as opções e decisões tomadas pelo grupo, ao longo de toda a fase, bem como a demonstração do sistema solar dinâmico final.

Principais Objetivos

Nesta fase 3, o principal objetivo consistiu em aplicar diversos conhecimentos adquiridos nas aulas teóricas e práticas de forma a aumentar a performance e o realismo do projeto com a adição de animações.

É de salientar que este trabalho pretende consolidar não só o conceito de noção de tempo e a sua associação com as transformações geométricas, mas também a utilização das curvas de *Catmull-Rom* e os *Bézier patches* em ambiente OpenGL.

Arquitetura do Programa - Atualização

Nesta fase, o grupo decidiu alterar ligeiramente a arquitetura do programa definida anteriormente, uma vez que não apresentava estar tão clara como o pretendido. O package "src" foi aquele que sofreu maior parte das alterações, os outros packages definidos tal como o "build" e "AsteroidScript" permaneceram iguais.

Em relação ao package "src", uma das alterações mais evidentes é a eliminação do package "structs", que agora todas as classes que pertenciam ao mesmo, estão inseridas no package "MotorUtils". Isto porque o motor.cpp era o único que fazia import das mesmas, logo faz sentido estarem apenas no package "Motor" e não fora. As classes *grupoStruct.cpp* e *transforms.cpp* são aquelas que na fase anterior eram denominadas por *groupUtils.cpp* e *transUtils.cpp*, respetivamente. Estas são responsáveis por armazenar e organizar as transformações e os grupos presentes no ficheiro XML, tal como já foi visto na última fase.

Além disso, no package "Gerador", foi incluído também o "GeradorUtils" que contém todas as classes auxiliares do *gerador.cpp.* O package "Utils" permanece igual, excepto que foi adicionada mais uma classe que serve como auxílio a ambos os packages "Motor" e "Gerador".

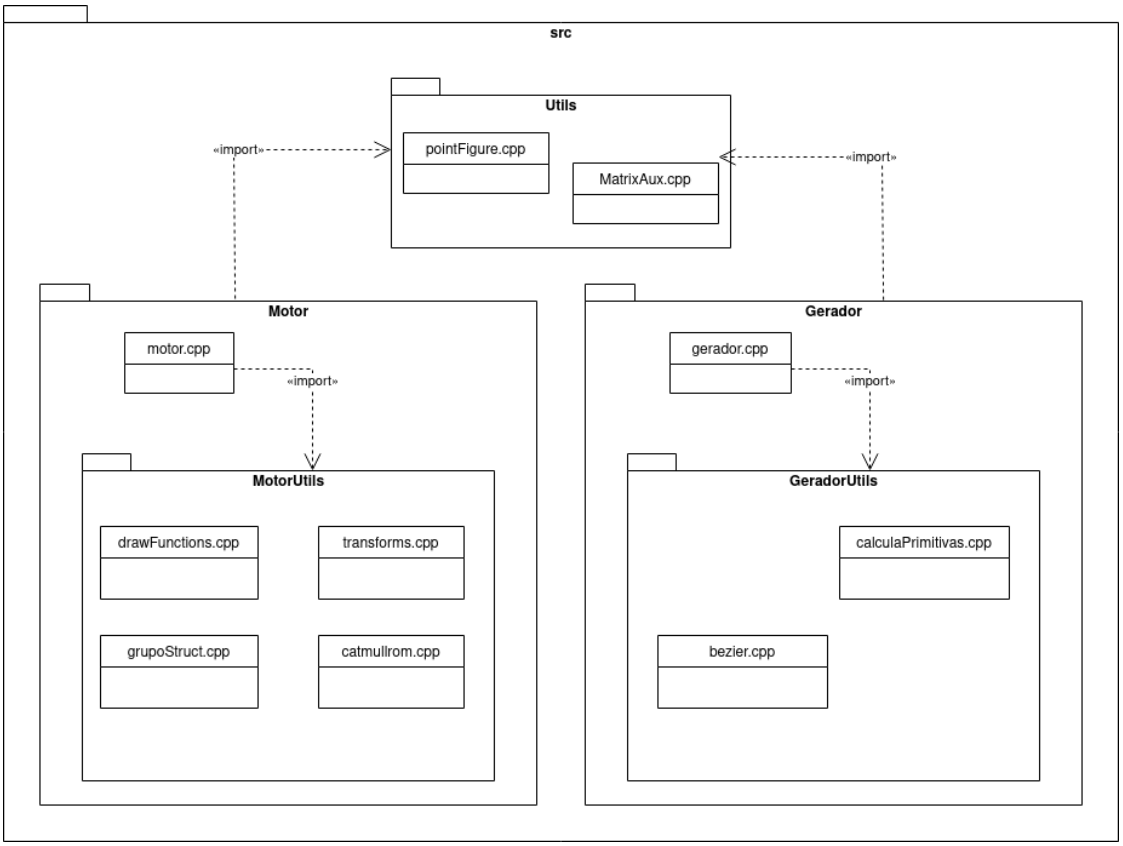


Figura 1 - Diagrama da arquitetura do programa

Motor

VBO's

Em contraste com a fase anterior, na atual foi solicitada a utilização da estrutura *VBO's* como ferramenta para desenhar as figuras. Este tipo de estruturas do *OpenGl* possibilitam o armazenamento dos dados na placa gráfica o que implica um consequente melhor desempenho, uma vez que o seu acesso é mais rápido.

Sendo assim, o grupo decidiu utilizar *VBO's* sem índices de modo a não modificar a anterior estrutura dos ficheiros *.3d.*

Para a sua implementação foi criada a variável global *pontosVBO,* que é um vetor de *floats* onde serão guardadas as coordenadas dos pontos pela ordem com que serão desenhados. O preenchimento deste vetor ocorre durante a leitura do ficheiro XML, onde para cada modelo (ficheiro *.3d*) são lidos do ficheiro e escritos para o vetor, em simultâneo, todos os pontos existentes pela ordem com que aparecem.

De seguida, foi necessário passar os valores lidos para a memoria da placa gráfica, logo foi criada outra variável global *buffers,* que é um *array* de vários buffers na memoria a serem preenchidos. Neste caso, só foi utilizado um buffer logo o *array* tem tamanho 1. Esta variável foi inicializada com recurso às funções do *openGL glGenBuffers e glBindBuffer.* Desta forma, utilizando a função *glBufferData* foram copiados todos os conteúdos do vetor *pontosVBO* para o *array* buffers[1].

Por último, para desenhar as figuras segundo as transformações associadas foi utilizada a função *drawFigures.* Para garantir que os pontos são desenhados no momento certo foi definida a variável global *vboRead* que indica a próxima posição inicial dos pontos a desenhar. Por outro lado, para saber quantos pontos (conjuntos de 3 *floats*) vão ser desenhados é acessada a variável de instância *fSizes* da classe *group* que está a ser utilizada como parâmetro. Sabendo estes dois valores, já é possível desenhar todas as figuras presentes no grupo chamando a função *glDrawArrays,* que dada a posição inicial (*vboRead*) e o número de pontos (*fSizes*), desenha os triângulos correspondentes. No fim do desenho é somada à variável *vboRead* o número de pontos desenhados para serem posteriormente desenhados os restantes elementos do buffer.

Transformações com tempo

Nesta fase foram adicionadas transformações do tipo "rotate" e "translate" que agora contém um atributo "time". No caso do "translate" este irá definir o tempo em segundos que demora a percorrer todos os pontos da curva, se for do tipo "rotate" então será o tempo que demora a rodar 360 graus sobre o vetor dado. A partir disto é possível animar as "scenes" realizadas.

Para tal, foi feita uma estrutura nova que irá tratar deste tipo de transformações, uma vez que aquela que foi usada para as transformações ditas "normais", isto é, sem o atributo de tempo,

não suprime as necessidades deste tipo de transformação. Portanto foi criada a *class transfTime,* que tal como a *class transform,* contém um *transtype* para conseguir distinguir se é uma rotação ou uma translação. Além disso, ainda foram adicionados dois vetores de *point.* O *vector points* armazena os pontos de controlo, ou seja, os pontos que irão definir a spline de Catmull-Rom, sendo que a curva terá de passar por todos eles. E, finalmente, foi definido ainda o *vector curvePoints* que contém os pontos que irão ser calculados para a curva através dos vários métodos definidos na classe *catmullrom.cpp*. Isto foi feito de forma a não termos que estar a calcular os pontos todos da curva várias vezes, evitando assim o desperdício de memória. Portanto, apenas são calculados uma vez, mais concretamente, após serem lidos e armazenados os pontos de controlo, uma vez que caso este cálculo estivesse dentro do método *renderScene()* estaríamos a calcular os pontos todos da curva a cada iteração.

Os vetores apenas são inicializados para o caso do *transtype* ser um *translate*. Portanto, foram criados dois métodos diferentes para a construção da *class transfTime*, um próprio para o *rotate* e outro para o *translate*.

```
class transfTime {
    transtype type;
    int time;
    float x;
    float y;
    float z;
    std::vector<utils::point> points;
    std::vector<utils::point> curvePoints;
```

Figura 2 - Class transfTime

```
enum class transtype { none, translate, rotate, scale, color };

class transform {
    float x;
    float y;
    float z;
    float angle;
    transtype type;
```

Figura 3 - Class transform

Curvas de *Catmull-Rom*

Uma curva Catmull-Rom necessita no mínimo de quatro pontos P_0, P_1, P_2 e P_3, sendo que cada curva será desenhada entre P_1 e P_2. Cada ponto na curva entre P_1 e P_2 é especificado através de t, que indica a porção da distância entre estes dois pontos (sendo que $t \in [0.0,\ldots,1.0]$).

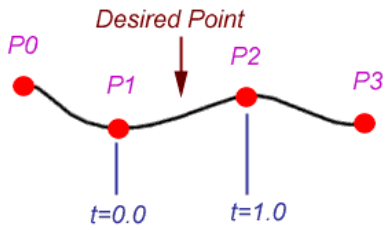


Figura 4 - Spline Catmull-Rom

Neste caso, t será calculado partindo do valor tempo definido no xml para cada translação (correspondente nesta parte do código a t.getTime()), e do ELAPSED_TIME, fazendo a divisão de ambos os valores obtemos o instante de tempo para o qual nós queremos calcular o ponto correspondente da curva. É de notar ainda que foi necessário dividir o ELAPSED_TIME por 1000 de forma a converter o tempo para segundos.

```
float timeT = ((float) glutGet(GLUT_ELAPSED_TIME) / 1000) / (float)(t.getTime());
catmull::getGlobalCatmullRomPoint(&t, timeT, (float*)pos, (float*)deriv);
```

Figura 5 - Cálculo do argumento gt para a funçaõ getGlobalCatmullRomPoint

A função getCatmullRomPoint irá calcular as posições e derivadas de cada ponto da curva, sendo esta chamada na função getGlobalCatmullRomPoint, que irá primeiro identificar os quatro pontos que irão formar o segmento atual da spline Catmull-Rom.

Uma vez obtidos os valores relevantes, é realizada uma translação para as coordenadas guardadas em *pos e o cometa é realinhado com a curva Catmull-Rom através de uma matriz de rotação calculada a partir dos valores guardados em *deriv, recorrendo para isso às funções auxiliares definidas em matrixUtils.

```
glTranslatef(pos[0], pos[1], pos[2]);

float m[4][4];
float x[3], z[3];

matrixUtils::cross(deriv, aux_y, z);
matrixUtils::cross(z, deriv, aux_y);
matrixUtils::normalize(deriv);
matrixUtils::normalize(aux_y);
matrixUtils::normalize(z);
matrixUtils::buildRotMatrix(deriv, aux_y, z, *m);
glMultMatrixf(*m);
```

Figura 6 - Excerto de código

Este segmento de código será, portanto, equivalente a calcular uma matriz de rotação, com vetores normalizados, tal como é descrito nas fórmulas seguintes (sendo X_i equivalente a *deriv e Y_i equivalente a *aux_y, tendo este sido inicializado com os valores {0,1,0} para Y_0).

$$\begin{aligned} \overrightarrow{X_i} &= p'(t) \\ \overrightarrow{Z_i} &= X_i \times \vec{Y}_{i-1} \\ \overrightarrow{Y_i} &= \overrightarrow{Z_i} \times \overrightarrow{X_i} \end{aligned} \qquad M = \begin{bmatrix} X_x & Y_x & Z_x & 0 \\ X_y & Y_y & Z_y & 0 \\ X_z & Y_z & Z_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotação

Tendo o tempo que uma rotação demora a completar uma volta de 360 graus, é necessário calcular o ângulo da mesma.

Para isso foi feito o seguinte cálculo:

$$angulo = \frac{(\text{ELAPSED}_{\text{TIME}}/1000) * 360}{time}$$

É necessário dividir o *ELAPSED_TIME* por 1000, de modo a passar a unidade de tempo de milissegundos para segundos. De seguida multiplica-se por 360 e divide-se pelo tempo que foi obtido através da leitura no *xml* na *tag <rotate>.*

Na figura seguinte está representado o código que representa o cálculo demonstrado:

```
float angle = (((float)glutGet(GLUT_ELAPSED_TIME) / 1000) * 360) / (float)t.getTime();
glRotatef(angle, t.getX(),t.getY(),t.getZ());
```

Figura 7 - Excerto de código glRotatef

Estrutura de Dados – Atualização

Com a inclusão de VBO's de forma a armazenar as figuras que serão desenhadas, e das transformações com tempo, foi necessário alterar a estrutura de dados responsável por guardar toda a informação de cada grupo do ficheiro XML.

Portanto em primeiro lugar excluiu-se o *vector* de *figures*, uma vez que agora usa-se VBO's para armazenar as figuras. E ainda se adicionou uma variável do tipo *size_t* que representa o número

total de pontos de todas as figuras desse grupo. Além disso, também se acrescentou o vetor curvas, que tem como objetivo armazenar todas as transformações que envolvem o atributo tempo.

```
class group {
    size_t fSizes = 0; //soma do numero de pontos(3 floats) de todas as figuras
    std::vector<transform> transformacoes;
    std::vector<transfTime> curvas;
    std::vector<group> filhos;
```

Figura 8 - Class group

Leitura e *Parse* do novo ficheiro XML

De modo a ler as novas adições feitas ao ficheiro XML, isto é, a inclusão de transformações que envolvem o atributo tempo, foi necessário acrescentar novas condições à função encarregue por fazer *parse* do ficheiro XML e armazenar as informações do mesmo na estrutura de dados designada para o efeito.

Sendo assim, em primeiro lugar nas condições que verificam se o nodo do ficheiro XML trata-se de um *rotate* ou *translate*, foi necessário inserir uma condição dentro das mesmas para verificar se contem o atributo *time*. Caso esta condição se verifique então iremos inicializar a estrutura *transfTime*, armazenar as informações adequadas na mesma e de seguida adicionar à estrutura *group*. É importante realçar que no caso de se tratar de uma translação com o atributo tempo, antes de se adicionar à estrutura *group*, é chamada a função "setCurvePoints()" para calcular os pontos da curva e guardar os mesmos dentro da *transfTime* em questão. Além disso, de modo a manter a verificação que o ficheiro XML tem a estrutura correta foi necessário adicionar condições com variáveis que averiguam se não existe mais do que uma mesma transformação ou tag (translate, translate time, rotate, rotate time, color, scale ou models) no mesmo grupo.

```
if(strcmp(elem->Value(),"translate")==0){

    if (elem->Attribute("time")) {
        if (translateTime == 1) {
            validate = 1;
            printf("ERRO TRANSLATE TIME - Couldn't parse XML file\n");
            return g;
        } else {
            time = atoi(elem->Attribute("time"));
            printf("time: %d\n", time);
            TiXmlElement *point = elem->FirstChildElement("point");
            std::vector<utils::point> pontos;
            while (point) {
                utils::point p;
                p.x = atof(point->Attribute("X"));
                p.y = atof(point->Attribute("Y"));
                p.z = atof(point->Attribute("Z"));
                pontos.push_back(p);
                //next sibling
                point = point->NextSiblingElement("point");
            }
            tTime.setTranslateTime(time, pontos);
            tTime.setCurvePoints();
            g.addCurva(tTime);
            translateTime = 1;
        }
    }
```

Figura 9 - Excerto de código da função parseXml()

Gerador

Patches de Bezier

Nesta fase, foi necessário acrescentar ao sistema solar já desenvolvido nas fases anteriores, um cometa com a forma de um *teapot*, sendo que este deveria ser baseado nos *patches de Bézier.*

Em primeiro lugar foi necessário analisar com atenção o formato do ficheiro input (*.patch)* fornecido. Após esta análise concluiu-se o seguinte:

- O primeiro valor indica o número de *patches* a considerar
- Há tantas linhas quanto o número de *patches*. Cada uma destas linhas contém 16 dígitos que indicam os índices dos pontos de controlo que fazem parte do *patch*.
- Após os índices, surge um valor que indica o número de pontos de controlo
- Por último surgem todos os pontos de controlo a considerar

Com estas informações, foi criado no *generator* uma função à qual chamamos *readBezier,* que recebe como argumentos o nome do ficheiro *patch* a ler e o valor da tecelagem. Esta função guarda o número de *patches* numa variável designada *numPatches*, coloca os pontos de controlo numa *figure (o tipo figure contém um vetor de pontos)* e guarda os índices desses pontos num vetor designado *auxIndices.* Após serem calculados todos os pontos necessários, a função coloca-os numa estrutura *figure,* sendo que, na função *main*, quando é pedido para gerar a superficie de Bézier, esses pontos calculados serão escritos para um ficheiro *.3d*.

Antes de ser explicado o algoritmo criado para o processamento de patches, é necessário entender como funcionam as curvas e as superfícies de Bézier.

Curvas de Bézier

Uma curva de Bézier é uma curva polinomial expressa como a interpolação linear entre alguns pontos representativos, designados pontos de controlo. Existem vários tipos de curvas de Bézier, mas a mais utilizada é curva cúbica (de grau 3), que necessita de 4 pontos de controlo (pontos de 3 coordenadas).

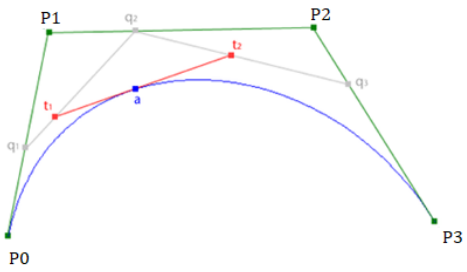


Figura 10 - Exemplo de curva cúbica de Bézier

Para calcular uma posição na curva, recorre-se à seguinte equação:

$$B(t) = (1 - t)^3 * P0 + 3 * t * (1 - t)^2 * P1 + (3\,t)^2 * (1 - t) * P2 + t^3 * P3$$

Em que P0, P1,P2 e P3 são os pontos de controlo e t é uma variável que pertence ao intervalo [0,1].

Superfícies de Bézier

As superfícies de *Bézier* são generalizações das curvas de *Bézier* a dimensões de ordem superior. Para criar estas superfícies (e desenhar o *teapot*), basta aplicar a fórmula anteriormente descrita, no entanto deixamos de ter um único parâmetro *t*, e passamos a ter 2 parâmetros ao quais chamamos *u* e *v*. É de realçar que, quanto maior for o valor da tecelagem, mais bem definida será a figura.

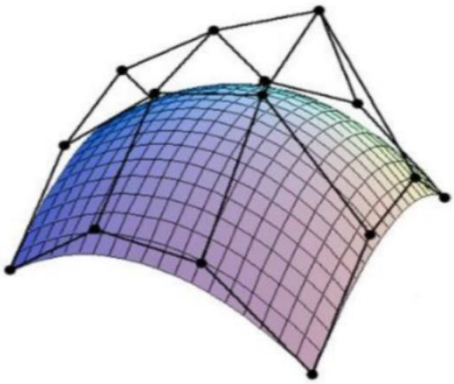


Figura 11 - Exemplo de superfície de Bézier

Assim sendo, a equação que determina um ponto na superfície de *Bézier*, é a seguinte:

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

M é uma matriz já pré-definida e M^T é a sua transposta (que acaba por ser igual a M):

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

Desta forma, já foi possível desenvolver um algoritmo em C++ para obter todos os pontos necessário para o desenho da primitiva *teapot*.

Em *Bezier.cpp*, presente na pasta *GeradorUtils,* foi primeiro definida uma função designada *getBezierPoint,*que recebe como argumentos os parâmetros *u* e *v,* as matrizes *matrixX*, *matrixY* e *matrixZ* que representam respetivamente as componentes x,y e z dos 16 pontos de controlo que estão a ser processado no momento. A função recebe também um vetor *pos*, no qual será colocado a posição final calculada.

```
void bezier::getBezierPoint(float u, float v, float** matrixX, float** matrixY, float** matrixZ, float* pos) {
    float bezierMatrix[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 } };

    float vetorU[4] = { u * u * u, u * u, u, 1 };
    float vetorV[4] = { v * v * v, v * v, v, 1 };

    float vetorAux[4];
    float px[4];
    float py[4];
    float pz[4];

    float mx[4];
    float my[4];
    float mz[4];

    multMatrixVector((float*)bezierMatrix, vetorV, vetorAux);
    multMatrixVector((float*)matrixX, vetorAux, px);
    multMatrixVector((float*)matrixY, vetorAux, py);
    multMatrixVector((float*)matrixZ, vetorAux, pz);

    multMatrixVector((float*)bezierMatrix, px, mx);
    multMatrixVector((float*)bezierMatrix, py, my);
    multMatrixVector((float*)bezierMatrix, pz, mz);


    pos[0] = (vetorU[0] * mx[0]) + (vetorU[1] * mx[1]) + (vetorU[2] * mx[2]) + (vetorU[3] * mx[3]);
    pos[1] = (vetorU[0] * my[0]) + (vetorU[1] * my[1]) + (vetorU[2] * my[2]) + (vetorU[3] * my[3]);
    pos[2] = (vetorU[0] * mz[0]) + (vetorU[1] * mz[1]) + (vetorU[2] * mz[2]) + (vetorU[3] * mz[3]);

}
```

Figura 12 - Excerto de código

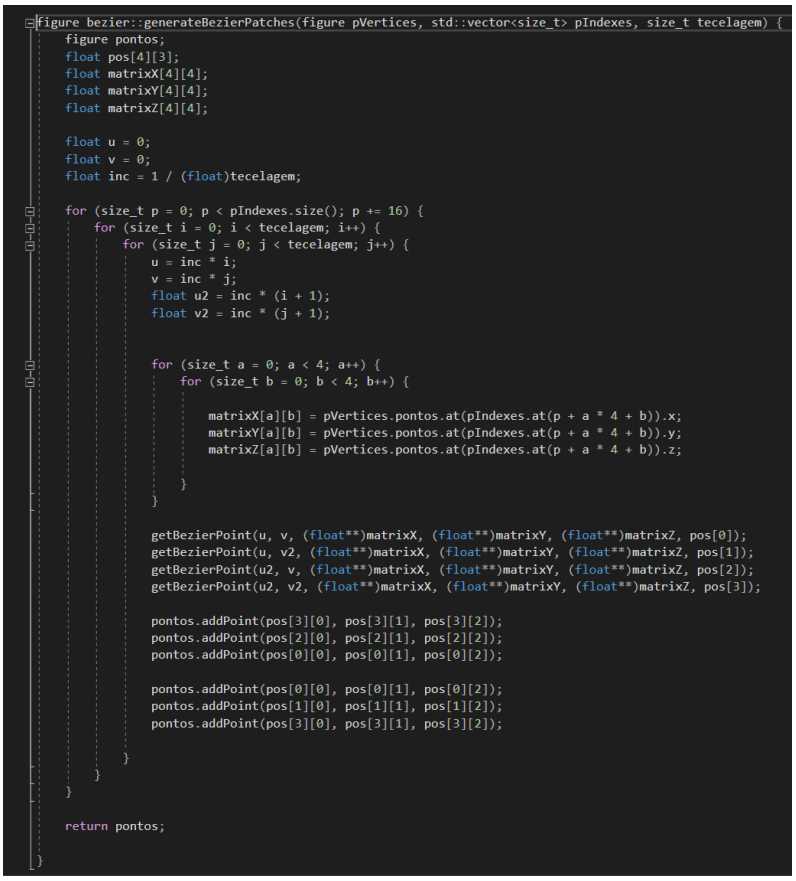
De seguida, foi definida a função *generateBezierPatches,* que devolve uma *figure,* com todos os pontos a desenhar. Aqui, utiliza-se um ciclo para percorrer todos os índices, 16 de cada vez, dado que cada *patch* a desenhar tem 16 pontos de controlo (indicados pelos índices).

Dentro do primeiro ciclo *for,* foram definidos mais dois ciclos, que serão executados tantas vezes quanto o valor da tecelagem.

São inicializadas e preenchidas as matrizes *matrixX*, *matrixY* e *matrixZ,* com as componentes x,y e z dos pontos de controlo do *patch* a ser processado no momento. Tendo isto, é chamada a função *getBezierPoint 4* vezes, sendo colocado em *pos[0], pos[1], pos[2] e pos[3]*, os 4 pontos necessários para o desenho do *patch*.

É de realçar que foram definidos os parâmetros *u, v, u2* e *v2*, uma vez que para o desenho de um *patch*, o P0, utiliza o parâmetro *u* e *v* da iteração atual, o P1 utiliza o parâmetro *u* da iteração atual e o *v* da iteração seguinte (v2), o P2 utiliza o parâmetro *u* da iteração seguinte(u2) e o *v* da iteração atual e, por fim, o P3 o P1 utiliza o parâmetro *e* v da iteração seguinte. São utilizados os valores dos parâmetros des iterações seguintes, para haver continuidade entre as várias "secções" de um *patch* originadas pela tecelagem.

Por fim, esses pontos são adicionados à estrutura *figure*, já referida.



```
figure bezier::generateBezierPatches(figure pVertices, std::vector<size_t> pIndexes, size_t tecelagem) {
    figure pontos;
    float pos[4][3];
    float matrixX[4][4];
    float matrixY[4][4];
    float matrixZ[4][4];

    float u = 0;
    float v = 0;
    float inc = 1 / (float)tecelagem;

    for (size_t p = 0; p < pIndexes.size(); p += 16) {
        for (size_t i = 0; i < tecelagem; i++) {
            for (size_t j = 0; j < tecelagem; j++) {
                u = inc * i;
                v = inc * j;
                float u2 = inc * (i + 1);
                float v2 = inc * (j + 1);


                for (size_t a = 0; a < 4; a++) {
                    for (size_t b = 0; b < 4; b++) {

                        matrixX[a][b] = pVertices.pontos.at(pIndexes.at(p + a * 4 + b)).x;
                        matrixY[a][b] = pVertices.pontos.at(pIndexes.at(p + a * 4 + b)).y;
                        matrixZ[a][b] = pVertices.pontos.at(pIndexes.at(p + a * 4 + b)).z;

                    }
                }

                getBezierPoint(u, v, (float**)matrixX, (float**)matrixY, (float**)matrixZ, pos[0]);
                getBezierPoint(u, v2, (float**)matrixX, (float**)matrixY, (float**)matrixZ, pos[1]);
                getBezierPoint(u2, v, (float**)matrixX, (float**)matrixY, (float**)matrixZ, pos[2]);
                getBezierPoint(u2, v2, (float**)matrixX, (float**)matrixY, (float**)matrixZ, pos[3]);

                pontos.addPoint(pos[3][0], pos[3][1], pos[3][2]);
                pontos.addPoint(pos[2][0], pos[2][1], pos[2][2]);
                pontos.addPoint(pos[0][0], pos[0][1], pos[0][2]);

                pontos.addPoint(pos[0][0], pos[0][1], pos[0][2]);
                pontos.addPoint(pos[1][0], pos[1][1], pos[1][2]);
                pontos.addPoint(pos[3][0], pos[3][1], pos[3][2]);

            }
        }
    }

    return pontos;

}
```

Figura 13 - Excerto de código

Câmara em Primeira Pessoa

Com o intuito de oferecer ao utilizador uma maior liberdade de visualização, foi implementada no motor uma camara em primeira pessoa (FPS). Esta funcionalidade extra permite ao usuário percorrer livremente o cenário. A sua execução implicou o uso da estratégia de manter a câmara estática e movimentar todas as figuras para dar a sensação de movimento. Este tipo de camara pode ser ativada durante a execução do programa pressionando a tecla '2'. Para voltar a ativar a camara que aponta para o centro do referencial é necessário pressionar a tecla '1'.

Para a sua implementação, em primeiro lugar foi necessário definir 6 variáveis globais de auxílio:

- cameraMoves: *array* com as coordenadas da posição da câmara;
- *lastx* e *lasty*: para guardar a última posição do rato no ecrã;
- *valid:* boleano que verifica se o utilizador pressionou o botão esquerdo do rato;
- *xrot:* ângulo de rotação para rodar a camara para cima ou baixo;
- *yrot:* ângulo de rotação para rodar a camara para a esquerda ou direita.

Para definir a rotação da camara, cada vez que é pressionado o botão esquerdo do rato são atualizadas as variaveis *lastx*, *lasty* e *valid.* Consequentemente, quando é movido o rato no ecrã, é calculada a diferença entre a posição atual e a antiga e é subtraído aos ângulos de rotação *xrot* e *yrot* os respetivos valores.

Por outro lado, para capturar as ações do teclado que vão modificar a posição da camara no referencial, é utilizada a função *keyboard* que dependendo da tecla, vai incrementar ou decrementar as coordenadas da câmara, de acordo com o movimento que se deseja.

Sabendo os ângulos de rotação *xrot* e *yrot* e a posição da camara no referencial, cada vez que é desenhado o cenário são feitas rotações de *xrot* graus em torno de x e *yrot* graus em torno de y e é feita a traslação segundo o vetor (-cameraMoves[0], -cameraMoves[1], -cameraMoves[2]). Estes valores são negativos porque como estamos a fazer transformações sobre o 'mundo' em relação à camara a sua posição será simétrica à da camara.

Sistema Solar

Cometa

Em adição à fase anterior, foi solicitada a implementação do desenho de um cometa no sistema solar. Para além do desenho da figura, uma vez que estamos perante um sistema solar dinâmico, foi também necessário definir a trajetória que o cometa vai percorrer no referencial.

Em primeiro lugar, para fazer o desenho do cometa foi utilizado o ficheiro dado, *teapot.patch*, que contém os pontos de controlo para desenhar um *teapot* aplicando superfícies de Bezier. Desta forma, com recurso ao gerador foram gerados todos os pontos necessários ao seu desenho com tecelagem igual a 5.

Posteriormente, como referência para o tipo de trajetória da órbita do cometa foi utilizada a do famoso cometa *Halley,* que no nosso sistema consideramos aproximadamente elíptica. Para a sua definição foi utilizada a nova funcionalidade do motor *translate time* que utilizando curvas de *Catmull-Rom* e pontos de controlo, efetua translações ao longo do tempo. Sendo assim, depois de várias tentativas com diferentes valores ficaram definidos os 8 pontos de controlo (A,B,C,D,E,F,G,H) representados no referencial abaixo.

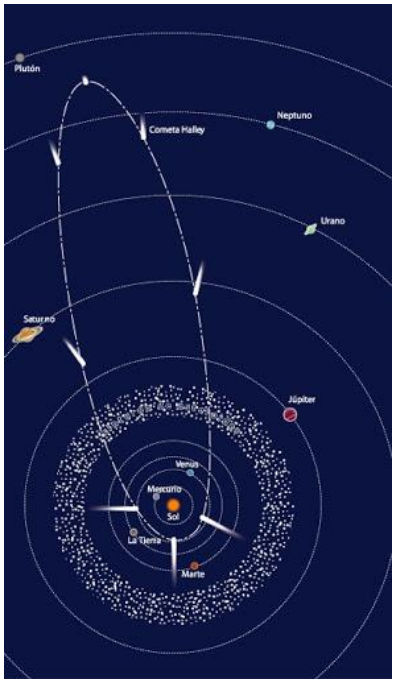


Figura 15 - Órbita do cometa Halley

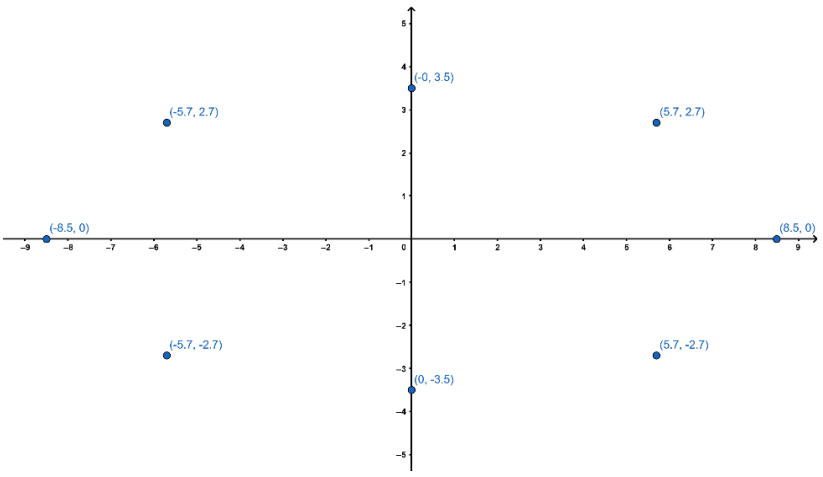


Figura 14 - Pontos de controlo da órbita do cometa

Uma vez que os pontos acima definem uma 'elipse' centrada no referencial e pretendemos que esteja 'deslocada', foi necessário efetuar uma translação de todos os pontos segundo o vetor $(4.75, 0, 0)$. Para além disso, ao *teapot* foram aplicadas uma escala $(0.05, 0.05, 0.05)$ para reduzir o seu tamanho e uma rotação de -90 graus em torno do eixo do x para o colocar 'de pé' na órbita. Sabendo todos os pontos da órbita e respetivas transformações e tendo o ficheiro teapot.*3d* criado foram passados todos os dados necessários ao desenho para o respetivo ficheiro XML.

```
<group>
    <color R="0.3" G="0.3" B="0.3"/>
    <translate X="4.75" Y="0" Z="0"/>
    <translate time="20" >
        <point X="0" Y="0" Z="3.5" />
        <point X="-5.7" Y="0" Z="2.7" />
        <point X="-8.5" Y="0" Z="0" />
        <point X="-5.7" Y="0" Z="-2.7" />
        <point X="0" Y="0" Z="-3.5" />
        <point X="5.7" Y="0" Z="-2.7" />
        <point X="8.5" Y="0" Z="0" />
        <point X="5.7" Y="0" Z="2.7" />
    </translate>
    <group>
        <color R="0.48" G="0.48" B="0.49"/>
        <scale X="0.05" Y="0.05" Z="0.05"/>
        <rotate angle="-90" axisX="1" axisY="0" axisZ="0"/>
        <models>
            <model file="teapot.3d" />
        </models>
    </group>
</group>
```

Figura 17 – Execerto do ficheiro XML referente ao cometa

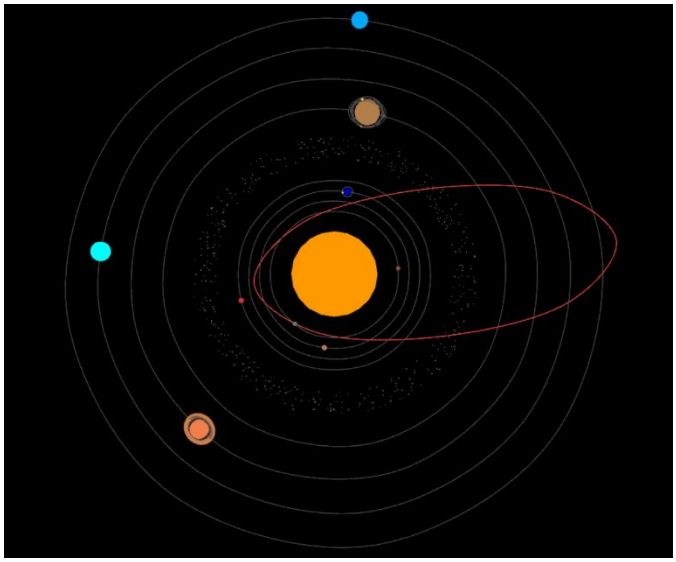


Figura 16 - Desenho da órbita do cometa no sistema solar

Ficheiro XML

De modo a incluir as animações e o cometa no sistema solar de forma a torná-lo dinâmico foi necessário alterar o ficheiro XML anteriormente criado. Para tal acrescentou-se para cada planeta uma tag do tipo translate que contém o atributo time. Dentro desta tag definimos 8 pontos de controlo de modo que o resultado final seja uma curva com um formato próximo da trajetória real dos planetas do sistema solar. Além disso, foi também necessário fazer uma breve pesquisa sobre o tempo de translação de cada planeta, uma vez que cada planeta tem a sua própria velocidade. Assim sendo, foi possível definir o atributo tempo da transformação em questão. O mesmo raciocínio foi aplicado aos satélites naturais de cada planeta.

Ainda foi decidido aplicar uma rotação com o atributo tempo ao sol de forma a fazer com que ele rode em torno de si mesmo, mas como ainda não aplicamos texturas, a sua visualização não é tão notória.

Tal como na fase passada, a escrita deste ficheiro foi feita de forma gradual, sendo que sempre que aplicávamos uma nova transformação a um planeta testávamos o output de forma a verificar se o resultado se encontrava do nosso agrado.

Na figura seguinte, de modo a exemplificar o raciocínio aplicado, encontra-se um excerto do ficheiro XML que define o desenho e a animação do planeta Terra.

```
<group>
    <color R="0.3" G="0.3" B="0.3"/>
    <rotate angle="234" axisX="0" axisY="1" axisZ="0"/>
    <translate time="10">
        <point X="2.828427128" Y="0" Z="2.828427128"/>
        <point X="0" Y="0" Z="4"/>
        <point X="-2.828427128" Y="0" Z="2.828427128"/>
        <point X="-4" Y="0" Z="0"/>
        <point X="-2.828427128" Y="0" Z="-2.828427128"/>
        <point X="0" Y="0" Z="-4"/>
        <point X="2.828427128" Y="0" Z="-2.828427128"/>
        <point X="4" Y="0" Z="0"/>
    </translate>
    <group>
        <color R="0" G="0" B="0.8"/>
        <scale X="0.08" Y="0.08" Z="0.08"/>
        <models>
            <model file="sphere.3d"/>
        </models>
        <group>
            <color R="0.3" G="0.3" B="0.3"/>
            <translate time="10">
                <point X="2.121320346" Y="0" Z="2.121320346"/>
                <point X="0" Y="0" Z="3"/>
                <point X="-2.121320346" Y="0" Z="2.121320346"/>
                <point X="-3" Y="0" Z="0"/>
                <point X="-2.121320346" Y="0" Z="-2.121320346"/>
                <point X="0" Y="0" Z="-3"/>
                <point X="2.121320346" Y="0" Z="-2.121320346"/>
                <point X="3" Y="0" Z="0"/>
            </translate>
            <group>
                <color R="0.9" G="0.9" B="0.9"/>
                <scale X="0.25" Y="0.25" Z="0.25"/>
                <models>
                    <model file="sphere.3d"/>
                </models>
```

Figura 18 - Exemplo do código em XML da Terra

AsteroidScript

Para finalizar a escrita do ficheiro XML, de modo a facilitar a conceção da cintura de asteroides, foi utilizado o script em python criado na fase anterior, sendo que foi necessário efetuar algumas mudanças.

Visto estarmos perante um sistema solar dinâmico foi necessário modificar este código para que os asteroides se passassem a movimentar no cenário. Desta forma, uma vez que não pretendíamos desenhar as suas orbitas pois tornariam o desenho confuso, foi apenas acrescentada a transformação *rotate time,* que vai rodar os cometas em torno do eixo y. De modo a manter a sua aleatoriedade foi calculado para cada asteroide um tempo de rotação aleatório entre 20 e 30.

Output

A partir do ficheiro *sistemaSolar.xml* podemos observar o seguinte output, onde inclui as translações de cada planeta, do cometa e ainda o movimento da cintura de asteroides. 4

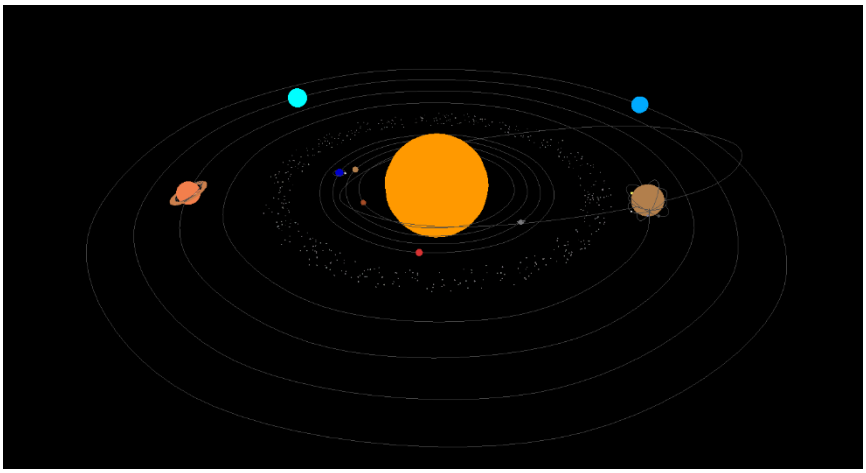


Figura 19 - Sistema Solar com cometa

No ficheiro *SistemaSolarSemCometa.xml*, é possível visualizar o sistema solar sem a trajetória do cometa.

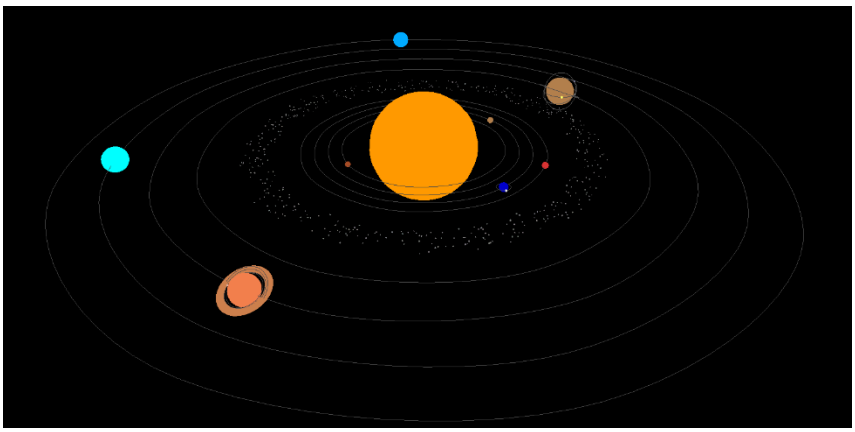


Figura 21 - Sistema Solar sem cometa

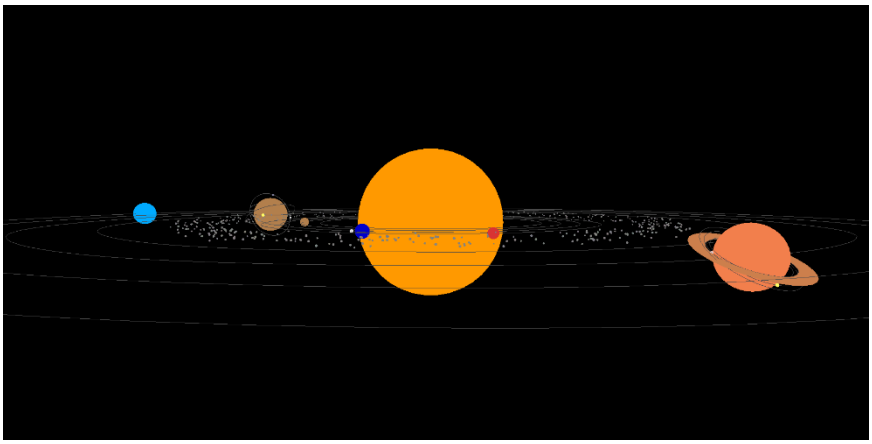


Figura 20 - Vista em plano horizontal do Sistema Solar

Conclusão

Neste trabalho foi possível aplicar e consolidar todo o conhecimento adquirido nas aulas teóricas e práticas da cadeira de Computação Gráfica, relativo aos VBO's, *Bézier patches*, curvas de *Catmull-Rom*, transformações com noção de tempo e animações.

O grupo considera que realizou esta fase com sucesso, na medida em que conseguiu implementar todas as funcionalidades pedidas no enunciado. No entanto, consideramos que esta fase 3 foi mais complexa e trabalhosa que as anteriores tendo, por isso, exigido mais tempo e atenção em determinados pontos tais como os Bézier *patches.*

Concluindo, foi conseguida a criação do sistema solar dinâmico, sendo que o grupo se encontra bastante satisfeito com o resultado final.