



UNIVERSIDADE DO MINHO
Departamento de Informática

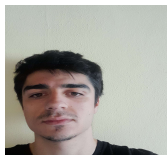
COMPUTAÇÃO GRÁFICA

Phase 3 - Curves Cubic Surfaces and VBOs

Grupo 37

Feito por:

Dinis Gonçalves Estrada (A97503)
Emanuel Lopes Monteiro da Silva (A95114)



A97393



A95114

May 5, 2023
Ano Letivo 2022/23

Conteúdo

1	Introdução	2
2	Objetivos	2
3	Arquitetura atualizada	3
4	Generator	4
4.1	Comandos	4
4.2	Patches de Bézier	4
4.3	Curvas de Bézier	4
4.3.1	Superfícies de Bézier	5
5	Engine	8
5.1	VBO's	8
5.2	Time Transformations	8
5.3	Catmull-Rom Curves	8
5.4	Rotation	10
5.5	Estrutura de Dados - Update	10
5.6	Leitura e Parse do novo ficheiro XML	11
6	Sistema Solar	12
6.1	Cometa	12
6.2	XML file	13
6.3	Output	14
7	Conclusão	15

1 Introdução

Nesta terceira fase do projeto da unidade curricular de Computação Gráfica foi necessário fazer alterações ao trabalho elaborado nas fases anteriores por forma a executar os novos pedidos.

Na realização desta etapa foi necessário alterar o generator para obter uma melhor performance. Para que tal fosse possível todas as primitivas existentes passaram a ser calculadas com recurso a VBO's. Apesar de esta não ser calculada pelo Generator foi adicionado um ficheiro com os pontos de um teapot baseado em *Bézier patches*.

Tal como o Generator, o Engine também foi modificado de forma a poder receber o novo tipo de ficheiros. Para além disto foi introduzida as curvas de *Catmull-Rom* por forma a tornar o Sistema Solar, desenvolvido nas fases anteriores, dinâmico. Para que isto seja possível, agora as transformações translate e rotate têm um tempo associado.

No decorrer desta fase, diversas decisões foram tomadas pelo grupo, as quais serão apresentadas ao longo deste relatório, bem como a demonstração do sistema solar dinâmico.

2 Objetivos

Esta terceira fase teve o principal objetivo de otimizar e tornar mais realista o projeto aplicando os diversos conhecimentos adquiridos nas aulas teóricas e práticas.

Para que estes objetivos fossem possíveis de alcançar tivemos como tarefa: ter uma noção de tempo associado às transformações geométricas introduzidas na fase anterior no decorrer da aplicação e também a aplicação das curvas de *Catmull-Rom* e *Bézier patches* em ambiente OpenGL.

3 Arquitetura atualizada

Nesta fase, alteramos ligeiramente a arquitetura do programa definida na fase anterior.

Em relação ao package "src", todos os packages dentro deste foram alterados passando então a haver novas classes dentro do do package "Engine", "Generator" e "Utils".

No package "Engine", todas as classes feitas nas fases anteriores continuam presentes no world.cpp, foi adicionada a classe *Catmull-Rom*, atualizamos a classe "Group" em que esta agora pode ter um vector e também a classe "Transform" que suporta as transformações com tempo.

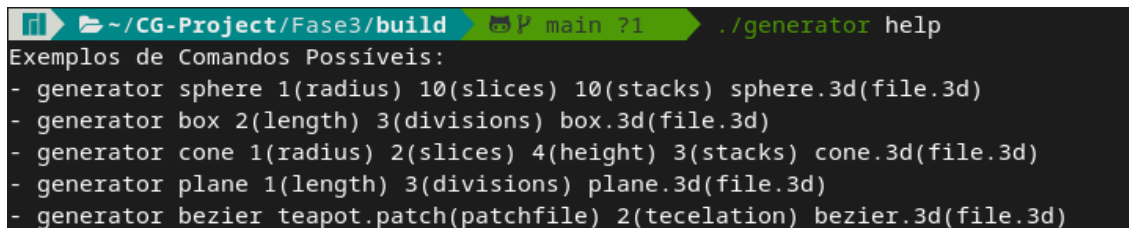
No package "Generator", modificamos o ficheiro "Generator.cpp" de forma a incluir a opção de cálculo da curva de Bézier. O cálculo desta está então presente numa nova classe chamada bezier no ficheiro "bezier.cpp".

O package "Utils" permanece quase igual com a exceção de uma nova classe "matrix" que serve de auxílio a ambos os packages "Engine" e "Generator".

4 Generator

4.1 Comandos

O comando "help" foi atualizado com a nova opção que o "generator" permite no programa.



```
~/CG-Project/Fase3/build main ?1 ./generator help
Exemplos de Comandos Possíveis:
- generator sphere 1(radius) 10(slices) 10(stacks) sphere.3d(file.3d)
- generator box 2(length) 3(divisions) box.3d(file.3d)
- generator cone 1(radius) 2(slices) 4(height) 3(stacks) cone.3d(file.3d)
- generator plane 1(length) 3(divisions) plane.3d(file.3d)
- generator bezier teapot.patch(patchfile) 2(teculation) bezier.3d(file.3d)
```

Figure 1: Menu Help Atualizado

4.2 Patches de Bézier

Um dos objetivos desta fase era acrescentar ao sistema solar desenvolvido nas fases anteriores, um cometa com a de forma de um teapot que deveria orbitar pelo sistema utilizando os *patches de Bézier*.

Para que tal fosse possível foi necessário analisar com atenção o formato do ficheiro input (.patch) fornecido. Após a análise chegamos ao seguinte:

- O primeiro valor indica o número de *patches* a considerar;
- Existe tantas linhas quanto o número de *patches* e que cada uma destas linhas contém 16 dígitos;
- Os 16 dígitos indicam os índices dos pontos de controlo que fazem parte do patch;
- A seguir a estes índices segue-se um valor que indica o número de pontos de controlo;
- No final estão todos os pontos de controlo a usar;

Para que fosse possível interpretar esta informação, foi criada no "generator.cpp" uma função ao qual chamamos *readBezier*, que recebe como argumentos o nome do ficheiro *patch* a ler e o valor de tecelagem. Esta função começa por guardar o número de *patches* numa variável *nPatches*, coloca os pontos de controlo numa *figure* (o tipo *figure* contém agora um vetor de pontos) e guarda os índices desses pontos num vetor designado *auxIndices*. Após serem calculados todos os pontos necessários, estes são colocados numa estrutura *figure*, sendo que, na função *main*, quando é pedido para gerar a superfície de Bézier, esses pontos calculados são escritos num ficheiro .3d como nome que o utilizador inseriu inicialmente.

Para que seja mais fácil de explicar o algoritmo desenvolvido para o processamento de patches, é necessário perceber como funcionam as curvas de Bézier.

4.3 Curvas de Bézier

A curva de Bézier é uma curva polinomial expressa como a interpolação linear entre alguns pontos representativos, designados por pontos de controlo. Há vários tipos de curvas de Bézier mas a mais utilizada é a curva cúbica que necessita de 4 pontos de controlo como é ilustrado na figura seguinte.

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

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

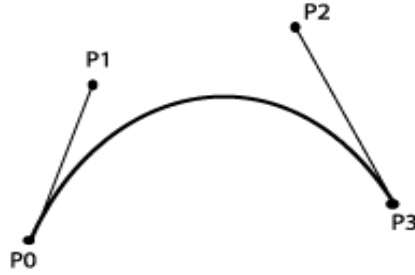


Figure 2: Exemplo de uma curva cúbica de Bézier

Em que P_0 , P_1 , P_2 e P_3 são os pontos de controlo e t é uma variável que pertence ao intervalo $[0, 1]$.

4.3.1 Superfícies de Bézier

Uma vez que sabemos o conceito de uma curva de Bézier, as superfícies de *Bézier* são nada mais, nada menos do que generalizações das curvas de Bézier a dimensões de ordem superior.

Para criar estas superfícies e ser possível de desenhar o teapot, basta aplicar a fórmula anteriormente descrita, só que desta vez deixamos de ter um único parâmetro t e passamos a ter 2 parâmetros aos quais chamamos u e v . Deve-se realçar o facto de que quanto maior for o valor da tecelagem melhor definida será a figura.

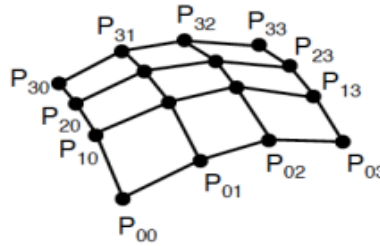


Figure 3: Exemplo de superfície de Bézier

Dado isto, a equação que detremina um ponto na superfície de *Bézier*, é a seguinte:

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

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

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

De acordo com o apresentado acima foi possível desenvolver o algoritmo para obter todos os pontos necessários para o desenho da primitiva do teapot.

No ficheiro "bezier.cpp" é então possível encontrar este algoritmo, que é composto por uma função designada por *getBezierPoint*, que recebe como argumentos os parâmetros *u*, *v*, as matrizes *matrixX*, *matrixY* e *matrixZ* que representam as componentes x, y e z dos 16 pontos de controlo que estão a ser processados no momento e por último recebe também um vetor *pos* que receberá a posição final calculada.

```
void bezier::getBezierPoint(float u, float v, float** matrixX, float** matrixY, float** matrixZ, float* pos) {
    float bezierMatrix[4][4] = { [0]={ [0]=-1.0f, [1]=3.0f, [2]=-3.0f, [3]=1.0f },
                                  [1]={ [0]=3.0f, [1]=-6.0f, [2]=3.0f, [3]=0.0f },
                                  [2]={ [0]=-3.0f, [1]=3.0f, [2]=0.0f, [3]=0.0f },
                                  [3]={ [0]=1.0f, [1]=0.0f, [2]=0.0f, [3]=0.0f } };

    float vetorU[4] = { [0]=u * u * u, [1]=u * u, [2]=u, [3]=1 };
    float vetorV[4] = { [0]=v * v * v, [1]=v * v, [2]=v, [3]=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]);
}
```

Figure 4: Função *getBezierPoint*

A acrescentar à função acima definimos a função *generateBezierPatches* que devolve uma *figure* com todos os pontos a desenhar. Esta utiliza um ciclo para percorrer todos os índices, 16 de cada vez, dado que cada *patch* tem 16 pontos de controlo (indicados pelos índices).

```
figure bezier::generateBezierPatches(figure pVertices, std::vector<size_t> pIndexes, int tecelation) {
    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)tecelation;

    for (size_t p = 0; p < pIndexes.size(); p += 16) {
        for (size_t i = 0; i < tecelation; i++) {
            for (size_t j = 0; j < tecelation; 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(n: pIndexes.at(n: p + a * 4 + b)).x;
                        matrixY[a][b] = pVertices.pontos.at(n: pIndexes.at(n: p + a * 4 + b)).y;
                        matrixZ[a][b] = pVertices.pontos.at(n: pIndexes.at(n: p + a * 4 + b)).z;
                    }
                }

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

Figure 5: Função *generateBezierPatches*

Neste primeiro ciclo *for*, definimos mais dois ciclos que serão executados de acordo com o valor da tecelagem. Inicializamos e preenchemos as matrizes mencionadas acima com os pontos de controlo do *patch* a ser processado no momento. Tendo isto, é chamada a função *getBezierPoint* 4 vezes, sendo colocado no vector *pos*, os 4 pontos necessários para o desenho do *patch*.

É importante referir 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 que é o *v2*, o P2 utiliza o parâmetro *u* da iteração seguinte (*u2*) e o *v* da atual e por último o P3 utiliza o *u* e *v* da iteração seguinte.

São utilizados os valores dos parâmetros das iterações seguintes, para haver uma continuidade entre as várias "secções" de um *patch* originadas pela tecelagem. Todos estes pontos são então adicionados à estrutura *figure*.

5 Engine

5.1 VBO's

Para a implementação dos VBO's começou-se por atualizar a class *World*, adicionando a variável *VBOpoints* que consiste num vector de floats onde estão armazenados todos os pontos pela ordem na qual serão desenhados. Seguidamente é preciso guardar todos os valores na memória da placa gráfica pelo que foi necessário adicionar a variável *buffers*, que corresponde a um array de vários buffers na memória a serem preenchidos, nesta fase apenas utilizamos um buffer logo o array possui tamanho 1. Esta variável foi inicializada na main usando as funções existentes no *glut*, *glGenBuffers* é responsável por inicializar o buffer, a função *glBindBuffer* é responsável para pôr o buffer ativo e por último a função *glBufferData* é utilizada para copiar o conteúdo do vetor *VBOpoints* para o array *buffers[1]*.

Finalmente para desenhar as figuras foi utilizada a função *drawFigures*. A variável *VBORead* foi acrescentada à classe *World* para indicar a próxima posição inicial dos pontos a desenhar e também foi adicionada a instância *fSizes* à classe *Group* que permite saber quantos pontos, conjuntos de 3 floats, vão ser desenhados. Com estas duas variáveis é possível agora desenhar todas as figuras presentes no grupo chamando a função *glDrawArrays*. No fim do desenho é somada à variável *VBORead* o número de pontos desenhados para serem posteriormente desenhados os restantes elementos do buffer.

5.2 Time Transformations

Nesta terceira fase foi pedido para expandir as transformações de rotação e translação que agora possuem o atributo *time*. Para a rotação, *time*, corresponde ao tempo em segundos que demora a rodar 360º sobre o vetor enquanto que para a translação corresponde ao tempo que demora a percorrer todos os pontos da curva. Com isto agora é possível tornar as *scenes* dinâmicas.

Para tal, foi necessário atualizar a classe *Transform* e acrescentar dois novos tipos de *trans.type*, *rotate.time* e *translate.time*. Foi acrescentado um booleano que irá indicar, no caso de ser uma translação, se o objeto irá ficar alinhado com a curva; um inteiro, *time*, para guardar o tempo no caso de ser uma transformação que o use; um vetor de pontos, *points* que irá armazenar a spline de Catmull-Rom, os pontos de controlo; e por fim outro vetor de pontos, *curvePoints*, que contém os pontos que irão ser calculados para a curva através dos vários métodos definidos na classe *catmull-rom.cpp*. Desta maneira estamos a fazer com que os pontos da curva sejam calculados uma vez, evitando o desperdício de memória, isto é, só os calculamos após serem lidos e armazenados os pontos de controlo, se este cálculo fosse aquando a renderização, usando o método *renderScene*, estaríamos, a cada iteração, a calcular todos os pontos da curva.

5.3 Catmull-Rom Curves

As curvas de Catmull-Rom são das curvas mais simples de calcular daí serem usadas em animações. O que distingue esta curva das outras é que basta definir um conjunto de pontos por onde queremos que a curva passe sem ser preciso especificar mais nada nem haver a preocupação de como os pontos se ligam.

$$p(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} P0 \\ P1 \\ P2 \\ P3 \end{bmatrix}$$

Figure 6: *Catmull Rom Curve Formula*

Uma curva Catmull-Rom necessita no mínimo de quatro pontos P_{i-1} , P_i , P_{i+1} e P_{i+2} , sendo que cada curva será desenhada entre P_i e P_{i+1} . Cada ponto na curva entre P_i e P_{i+1} é especificado através de t , que indica a porção da distância entre estes dois pontos (sendo que $t \in]0.0; 1.0[$).

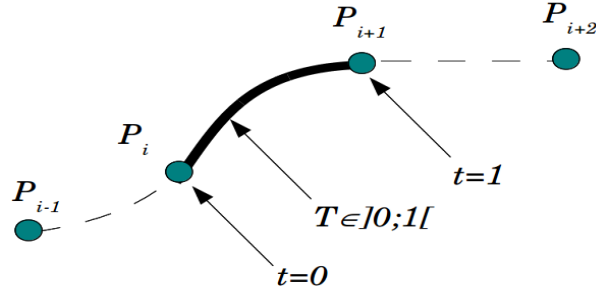


Figure 7: *Catmull-Rom Spline*

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.

A função `getCatmullRomPoint`, da classe `catmull-rom.cpp`, 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*.

Caso a translação tenha o *align* como valor booleano *true*, verificado usando o método `getAlign` da classe *Transform*, irá proceder se à construção da matriz de rotação:

$$\begin{cases} X_i = p'(t) \\ Z_i = X_i * Y_{i-1} \\ Y_i = Z_i * X_i \end{cases} \rightarrow \text{Matriz de rotação} = \begin{bmatrix} Xx & Yx & Zx & 0 \\ Xy & Yy & Zy & 0 \\ Xz & Yz & Zz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 8: Fórmula da *Matriz de rotação*

```
if(t.getAlign()) {
    float m[4][4];
    float x[3], z[3];

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

Figure 9: Implementação da *Matriz de rotação*

A Matriz de Rotação é calculada a partir dos valores guardados em *deriv*, recorrendo para isso às funções auxiliares definidas em *utils*. O segmento de código na imagem acima é equivalente a calcular uma matriz de rotação, com vetores normalizados, tal como é descrito na fórmula da Matriz de Rotação (sendo "Xi" equivalente a *deriv* e *Yi* equivalente a *aux.y*, tendo este sido inicializado com os valores {0,1,0} para *Y0*).

5.4 Rotation

Nesta fase, foi-nos pedido que adicionássemos uma variável de tempo opcional substituta ao ângulo já implementado nas fases anteriores. O tempo seria o número de segundos que demoraria para a rotação completar 360 graus em torno do eixo especificado. Tendo isto em mente, e com a ajuda da função *glutGet(GLUT_ELAPSED_TIME)* que nos dá o tempo em milissegundos desde que a *glutInit* foi chamada, fizemos uma simples regra de três simples: se 360 graus está para tempo dado, então o ângulo da rotação a ser efetuado está para o tempo recebido da função *glutGet(GLUT_ELAPSED_TIME)*, dividido por 1000 para passar para segundos.

```
case trans_type::rotate_time: {
    float angle = (((float)glutGet(query: GLUT_ELAPSED_TIME) / 1000) * 360) / (float)t.getTime();
    glRotatef(angle, x: t.getCoords()[0], y: t.getCoords()[1], z: t.getCoords()[2]);
    break;
}
```

Figure 10: Excerto de código *glRotate*

5.5 Estrutura de Dados - Update

Com a adição de VBO's foi necessário alterar a estrutura de dados encarregue por guardar a informação de cada grupo. Com isso removeu-se o vetor de "figures" e adicionou-se uma variável do tipo *size_t* que representa o número total de pontos de todas as figuras desse grupo.

```
class Group {
public:
    vector<transformations::Transform> transforms;
    vector<Group> child_nodes;
    size_t fSizes = 0;
public:
    Group() {};
    vector<transformations::Transform> getTrans();
    vector<Group> getChilids();
    size_t getFSizes();
    void addGroup(Group g);
    void addTransform(transformations::Transform);
    void addFSize(size_t);
};
```

Figure 11: Class *Group*

5.6 Leitura e Parse do novo ficheiro XML

Para 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 se verifique iremos ler todas as informações, inicializar a estrutura *Transform* e 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 *Transform* em questão.

```
if(strcmp(s1, transElement->Value(), s2, "translate") == 0) {
    if(transElement->Attribute(name: "time")) {
        int time = atoi(npstr, transElement->Attribute(name: "time"));
        bool align = strcmp(s1, transElement->Attribute(name: "align"), s2, "true") == 0;
        TiXmlElement *point = transElement->FirstChildElement(value: "point");
        std::vector<utils::point> pontos;
        while (point) {
            utils::point p;
            p.x = atof(npstr, point->Attribute(name: "x"));
            p.y = atof(npstr, point->Attribute(name: "y"));
            p.z = atof(npstr, point->Attribute(name: "z"));
            pontos.push_back(x, p);
            //next sibling
            point = point->NextSiblingElement(next: "point");
        }
        Transform transTime;
        transTime.setTranslateTime(time, align, pontos);
        transTime.setCurvePoints();
        group.addTransform(transTime);
    } else {
        float translateV[3];
        translateV[0] = stof(str, transElement->Attribute(name: "x"));
        translateV[1] = stof(str, transElement->Attribute(name: "y"));
        translateV[2] = stof(str, transElement->Attribute(name: "z"));
        Transform t = Transform();
        t.setTransform(translateV, angle, trans_type: translate);
        group.addTransform(t);
    }
}
```

Figure 12: Excerto de código da função readXMLgroup

6 Sistema Solar

6.1 Cometa

Nesta fase foi nos pedido uma demo dinâmica do sistema solar que incluía um cometa, construído usando o ficheiro *teapot.patch* que contém os pontos de controlo para desenhar um teapot aplicando superfícies de *Bezier*, com uma trajetória definida usando uma curva de *Catmull-Rom*. Com recurso ao *generator* 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 engine 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.

Os pontos definem uma "elipse" centrada ao que foi aplicada uma translação de todos os pontos segundo o vetor (4.75,0,0) para que esteja deslocada do centro. Também foi aplicada uma rotação de -90 graus em torno do eixo do x para colocar o teapot "em pé" na órbita e ainda uma escala de (0.05,0.05,0.05) para reduzir o seu tamanho.

```
<group>
  <transform>
    <color R="0.3" G="0.3" B="0.3"/>
    <translate x="4.75" y="0" z="0"/>
    <translate time="20" align="true">
      <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>
  </transform>
  <group>
    <transform>
      <color R="0.48" G="0.48" B="0.49"/>
      <scale x="0.05" y="0.05" z="0.05"/>
      <rotate angle="-90" x="1" y="0" z="0"/>
    </transform>
    <models>
      <model file="bezier_10.3d" />
    </models>
  </group>
</group>
```

Figure 13: Excerto do ficheiro XML referente ao cometa

6.2 XML file

Para acrescentar as animações e o cometa no sistema solar foi necessário adicionar para cada planeta uma tag do tipo `translate` que contém o atributo `time`, em que se definiu 8 pontos de controlo para que a curva se aproxime da trajetória real dos planetas do sistema solar. Foi também preciso fazer uma pesquisa sobre o tempo de translação de cada planeta, pois cada um tem a sua própria velocidade. Com isto conseguimos então definir o atributo tempo da transformação em questão.

Aplicamos uma rotação com atributo tempo ao sol de forma a que este rode em torno de si mesmo, apesar de ser pouco notório. Segue-se um excerto do ficheiro XML que define o desenho e a animação da Terra.

```
<group>
  <transform>
    <color R="0.3" G="0.3" B="0.3"/>
    <rotate angle="234" x="0" y="1" z="0"/>
    <translate times="10" align="true">
      <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>
  </transform>
  <group>
    <transform>
      <color R="0" G="0" B="0.8"/>
      <scale x="0.08" y="0.08" z="0.08"/>
    </transform>
    <models>
      <model file="sphere.3d"/>
    </models>
    <group>
      <transform>
        <color R="0.3" G="0.3" B="0.3"/>
        <translate times="10" align="true">
          <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>
      </transform>
      <group>
        <transform>
          <color R="0.9" G="0.9" B="0.9"/>
          <scale x="0.25" y="0.25" z="0.25"/>
        </transform>
        <models>
          <model file="sphere.3d"/>
        </models>
      </group>
    </group>
  </group>
```

Figure 14: Excerto do ficheiro XML referente à Terra

6.3 Output

A partir do ficheiro *solar_system.xml* podemos observar o seguinte output, onde inclui as translações de cada planeta e do cometa.

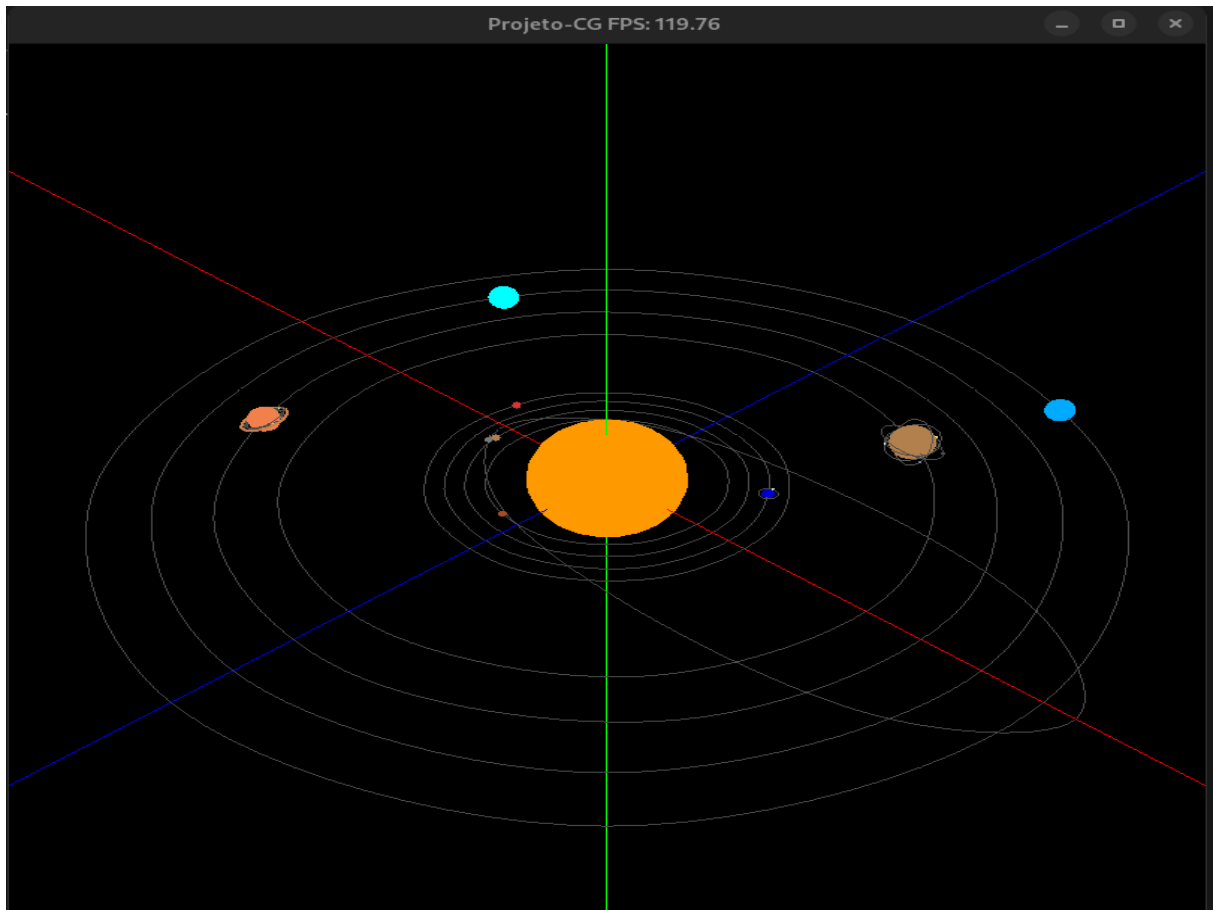


Figure 15: Sistema Solar

7 Conclusão

Com a realização desta fase do trabalho foi possível aplicar e consolidar o conhecimento adquirido nas aulas práticas e teóricas relativo aos *VBO's*, *Béziers patches*, curvas de *Catmull-Rom*, transformações com noção de tempo e animações.

Uma vez que conseguimos aplicar todas estas funcionalidades pedidas no enunciado, o grupo considera que realizou esta fase com sucesso. Apesar deste sucesso, esta fase mostrou-se bastante desafiadora por causa da implementação dos *VBO's* e pelos *Bézier patches*.

Para terminar, o grupo encontra-se bastante satisfeito com o resultado final uma vez que foi conseguida a criação do sistema solar dinâmico.