UNIVERSIDADE DO MINHO
ESCOLA DE ENGENHARIA

Computação Gráfica

TRABALHO PRÁTICO - FASE III

Adriana Henriques Esteves Teixeira Meireles A82582
Ana Marta Santos Ribeiro A82474
Carla Isabel Novais da Cruz A80564
Jéssica Andreia Fernandes Lemos A82061

MIEI - 3º Ano - 2º Semestre

Braga, 20 de abril de 2019

Índice

1 Introdução

Nesta terceira fase do projeto, espera-se que seja obtido um Sistema Solar constituído por um Sol, os planetas e os respetivos satélites naturais, bem como um cometa. De forma a tornar a sua implementação mais realista recorremos a curvas e superfícies de Bézier e a curvas de Catmull-Rom. Com o intuito de melhorar o desempenho, utilizamos VBOs para o desenho das primitivas.

Ao longo do presente relatório serão apresentadas as decisões tomadas durante o desenvolvimento desta etapa do projeto, assim como serão descritas as estratégias utilizadas para a concretização da mesma.

2 Organização do Código

Tendo em conta a implementação realizada nas duas fases anteriores, optamos por seguir a mesma abordagem, continuando a ter duas aplicações, a *Engine* e *Generator*.

2.1 Generator

Tal como nas fases anteriores esta aplicação é responsável por gerar os pontos dos triângulos que permitem construir as diversas figuras geométricas. Contudo, esta sofreu algumas alterações para a implementação desta fase, nomeadamente permitir realizar o *parse* dos ficheiros no formato *Patch*, como iremos explicar de seguida.

2.1.1 Patch

Nesta classe é realizado o *parse* dos ficheiros de formato *patch* de modo a serem armazenados os pontos de controlo.

- **Patch file**

Os patches de Bézier permitem ilustrar, em formato texto, uma superfície. A função responsável possui o seguinte tipo:

```
1 void Patch::parserPatchFile(string filename);
```

Estes ficheiros possuem um formato relativamente simples e com inúmeras características que se podem observar na função:

⇒ A primeira linha contém o número de patches (nPatches);

```
1 //...
2    getline(file,line);
3    nPatchs = stoi(line);
4 //...
```

⇒ As restantes linhas contêm os 16 índices de cada um dos pontos de controlo que constituem os patches da figura;

```
1 //...
2 for(int i = 0; i < nPatchs; i++){
3     vector<int> patchIndex;
```

```
4       if(getline(file,line)){
5           char* str = strdup(line.c_str());
6           char* token = strtok(str, " ,");
7
8           while (token != NULL){
                patchIndex.push_back(atoi(token));
9               token = strtok(NULL, " ,");
10          }
11
12          patchs[i] = patchIndex;
13          free(str);
14  }
15  //...
```

⇒ Segue-se uma linha que contém o número de pontos de controlo necessários para gerar a figura (nPoints);

```
1     //...
2      getline(file,line);
3      nPoints = stoi(line);
4     //...
```

⇒ No restante ficheiro encontram-se todos os pontos de controlo. A ordem destes é importante, porque cada ponto tem associado um índice.

```
1   //...
2   for(int i = 0; i < nPoints; i++){
3       if(getline(file,line)){
4           char* str = strdup(line.c_str());
5           char* token = strtok(str, " ,");
6
7           float x = atof(token);
8           token = strtok(NULL, " ,");
9           float y = atof(token);
10          token = strtok(NULL, " ,");
11          float z = atof(token);
12          Point *p = new Point(x,y,z);
13          controlPoints.push_back(*p);
14      }
15  //...
```

- **Superfícies de *Bézier***

Para a definição de uma superfície de Bézier usamos 16 pontos de controlo que representamos numa matriz 4 por 4. Para tal, começamos por definir as matrizes necessárias.

⇒ Matrizes a e b

$$a = \begin{bmatrix} a^3 & a^2 & a & 1 \end{bmatrix}$$

$$b = \begin{bmatrix} b^3 & b^2 & b & 1 \end{bmatrix}$$

⇒ Matrizes com Pontos de Controlo

$$P = \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{3}0 & P_{31} & P_{32} & P_{33} \end{bmatrix}$$

oMatrizes de Bézier

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

Tendo em conta os valores de a e b, que se encontram entre 0 e 1, é possível obter um ponto da superfície de Bézier, como podemos verificar de seguida.

$$P(a,b) = a \times M \times P \times M \times b;$$

A função que permite obter as coordenadas de um ponto é a *getPoint* que será apresentada e explicada de seguida. Primeiro ilustramos o seu tipo.

```
1 Point* Patch::getPoint(float ta, float tb, float coordenadasX[4][4],
    float coordenadasY[4][4], float coordenadasZ[4][4])
```

As matrizes recebidas como argumento, dizem respeito às componentes x, y e z de cada um dos pontos de controlo do patch em questão. Primeiramente, é definida a matriz de Bézier e os vetores a e b, sendo essa multiplicada pelos mesmos. Posteriormente, é multiplicado o resultado obtido anteriormente pelas componentes x, y e z de cada um dos pontos de controlo. Finalmente, cada componente de um ponto é adquirida com o resultado obtido até então vezes *bm*.

```
//...
       float m[4][4] = {{-1.0f,  3.0f, -3.0f,  1.0f},
                        { 3.0f, -6.0f,  3.0f,  0.0f},
                        {-3.0f,  3.0f,  0.0f,  0.0f},
                        { 1.0f,  0.0f,  0.0f,  0.0f}};

        float a[4] = { ta*ta*ta, ta*ta, ta, 1.0f};
        float b[4] = { tb*tb*tb, tb*tb, tb, 1.0f};

        float am[4];
        multMatrixVector(*m,a,am);
        float bm[4];
        multMatrixVector(*m,b,bm);

        float amCoordenadaX[4], amCoordenadaY[4],amCoordenadaZ[4];
        multMatrixVector(*coordenadasX,am,amCoordenadaX);
        multMatrixVector(*coordenadasY,am,amCoordenadaY);
        multMatrixVector(*coordenadasZ,am,amCoordenadaZ);

        for (int i = 0; i < 4; i++){
            x += amCoordenadaX[i] * bm[i];
            y += amCoordenadaY[i] * bm[i];
            z += amCoordenadaZ[i] * bm[i];
        }
    //...
}
```

Foi ainda necessário elaborar a função *getPatchPoints* que é responsável por obter os vértices dos triângulos. Assim, primeiro são preenchidas três matrizes (coordenadasX, coordenadasY e coordenadasZ) com os pontos de controlo da patch, ou seja, cada matriz contém a coordenada x, y ou z respetivamente do ponto, de modo a que com o auxílio da função *getPoints*, se possa gerar os vértices que constituem os triângulos. É importante

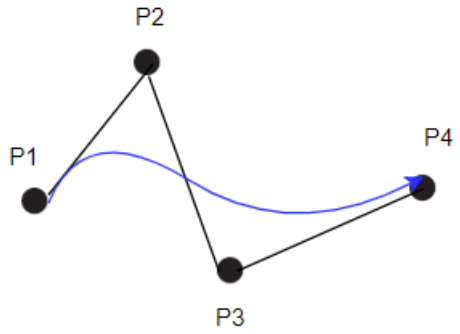
referir, que o valor da tecelagem permite definir a porção com que se vai percorrer u e v. De seguida, estes são armazenados num vetor de pontos de acordo com a regra da mão direita, para que possam posteriormente serem escritos no ficheiro *.3d* respetivo.

```
1  //declaracao de variaveis
2  float t = 1.0f /tessellation;
3  for (int i = 0; i < 4; i++){
4      for (int j = 0; j < 4; j++){
5          Point controlPoint = controlPoints[indexesControlPoints[pos]];
6          coordenadasX[i][j] = controlPoint.getX();
7          coordenadasY[i][j] = controlPoint.getY();
8          coordenadasZ[i][j] = controlPoint.getZ();
9          pos++;
10         }
11     }
12     for(int i = 0; i < tessellation; i++){
13         for (int j = 0; j < tessellation; j++){
14             u = i*t;
15             v = j*t;
16             uu = (i+1)*t;
17             vv = (j+1)*t;
18             Point *p0,*p1,*p2,*p3;
19             p0 = getPoint(u, v, coordenadasX, coordenadasY,
                   coordenadasZ);
20             p1 = getPoint(u, vv, coordenadasX, coordenadasY,
                   coordenadasZ);
21             p2 = getPoint(uu, v, coordenadasX, coordenadasY,
                   coordenadasZ);
22             p3 = getPoint(uu, vv, coordenadasX, coordenadasY,
                   coordenadasZ);
23
24             points.push_back(*p0); points.push_back(*p2);
                   points.push_back(*p1);
25             points.push_back(*p1); points.push_back(*p2);
                   points.push_back(*p3);
26         }
27     }
28     return points;
29 }
```

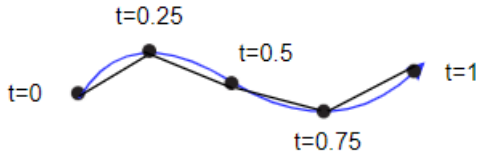
➠ **Curvas de *Bézier***

A obtenção destas curvas de Bézier é feita através de uma equação na qual são precisos quatro pontos de controlo: P1, P2, P3 e P4. Os mesmos, em junção com um parâmetro t, que varia entre 0 e 1, fazem com que seja possível a obtenção de uma curva. De seguida, poderá observar-se a fórmula que define uma curva de Bézier assim como um exemplo da mesma:

$$P(t) = (1-t^3)\cdot P1 + 3t(1-t^2)\cdot P2 + 3t^2(1-t)\cdot P3 + t^3\cdot P4$$



Para se obter uma curva com mais detalhe, é essencial aumentar o número de pontos e de segmentos. Em baixo, é apresentada uma curva com apenas 4 segmentos, com t a variar entre 0 e 1.



2.1.2 Point

Nesta fase optamos por representar os pontos da mesma forma que já tinhamos implementado nas fases anterior na *engine*. Desta forma, surgiu a classe Point. Neste caso destaca-se o facto de as coordenadas x, y e z serem definidas como *public* o que permitiu que não fosse necessário a alteração de código das fases anteriores.

```
public:
    float x;
    float y;
    float z;
    //gets e sets
```

2.1.3 Generator

Nesta classe apenas foi adicionada na main o excerto de código que permite a obtenção do .3d do patch.

2.1.4 Figures

Esta classe não apresenta quaisquer tipo de alterações dado que as primitivas geradas são as mesmas nas fases anteriores.

2.2 Engine

Esta aplicação também sofreu algumas alterações nesta fase do projeto, tais como a implementação de VBOs e de curvas de Catmull-Rom. Foram ainda realizadas alterações de modo a permitir a movimentação dos planetas. Assim, passaremos de seguida a explicar mais destalhadamente estas modificações.

2.2.1 Point

Esta classe foi definida nas outras fases do projeto e é utilizada para o desenvolvimento desta fase. Esta não apresenta alterações.

2.2.2 Shape

Uma das alterações realizadas nesta fase foi a implementação dos VBOs para o desenho das primitivas. O OpenGL oferece a possibilidade de inserir toda a informação sobre os vértices diretamente na placa gráfica através da utilização de *Vertex Buffer Object*.

Assim, para a implementação começamos por criar os vertex buffers, que são arrays que irão conter os vértices das primitivas a desenhar. Para além disso, optamos por armazenar o número de vértices que irá conter.

```
int numVertex;
GLuint bufferVertex[1];
```

Na *prepareBuffer* é gerado e preenchido o buffer do tipo *GLuint*. De seguida, guardamos na memória da placa gráfica *numVertex* * 3 vértices.

```
    //construir array de float com vertices
    glGenBuffers(1,bufferVertex);
    glBindBuffer(GL_ARRAY_BUFFER, bufferVertex[0]);
    glBufferData(GL_ARRAY_BUFFER,
                 sizeof(float) * numVertex * 3,
                 vertexs,
                 GL_STATIC_DRAW);
    //libertar array
```

Para desenhar a informação guardada recorremos à *draw*, pelo que serão desenhados *numVertex* * 3 triângulos.

```
    glBindBuffer(GL_ARRAY_BUFFER, bufferVertex[0]);
    glVertexPointer(3, GL_FLOAT, 0, 0);
    glDrawArrays(GL_TRIANGLES, 0, numVertex * 3);
```

Desta forma, é possível aumentar o desempenho com renderização imediata. Após esta alteração verificamos que os *frames por second* foram superiores aos esperados na fase anterior do projeto.

2.2.3 Transformation

Nesta classe foram adicionadas algumas variáveis relativas a transformações que surgiram nesta fase. Para além disso, adicionamos funções e variáveis que são necessárias à implementação das curvas de Catmull-Rom.

- **Curvas Catmull-Rom**

Para a representação das curvas definimos a utilização da spline de Catmull-Rom. Para a definição da curva serão necessários pelo menos quatro pontos. Os pontos extremos em conjunto com a tensão t permitem a definição de uma curva. Tendo em conta

que a tensão é zero, podemos obter a matriz M:

$$M = \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Para além disso definimos os vetores T e T', no qual t se encontra entre 0 e 1.

$$T = \begin{bmatrix} 3*t^2 & 2*t & 1 & 0 \end{bmatrix}$$

$$T' = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix}$$

Assim, podemos obter os pontos da curva da seguinte forma:

$$P(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} * M * \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

Deste modo, iremos conseguir obter um sistema solar dinâmico. Para tal, implementamos a função *getGlobalCatmullRomPoint* que permitirá a obtenção das coordenadas dos pontos bem como as suas derivadas. Esta, por sua vez, recorre à função *getCatmullRomPoint*, que utiliza as matrizes e vetores apresentados, gerando os valores de retorno da função global, através da multiplicação e derivação destas:

```
// matriz M apresentada anteriormente
    float m[4][4] = {
    { -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 }
    };

    float px[4],py[4],pz[4];
    for(int i = 0; i < 4 ; i++){
        px[i] = controlPoints[indexes[i]]->getX();
        py[i] = controlPoints[indexes[i]]->getY();
        pz[i] = controlPoints[indexes[i]]->getZ();
```

```
14      }
15
16      // Compute A = M * P
17      float a[4][4];
18      multMatrixVector(*m, px, a[0]);
19      multMatrixVector(*m, py, a[1]);
20      multMatrixVector(*m, pz, a[2]);
21
22      // Compute pos = T * A
23      float T[4] = { t*t*t, t*t, t, 1};
24      multMatrixVector(*a, T, p);
25
26      // Compute deriv = T' * A
27      float Tdev[4] = { 3*T[1] , 2*T[2], 1 , 0};
28      multMatrixVector(*a, Tdev, deriv);
```

Para implementarmos as órbitas dos planetas do Sistema Solar definimos a função *setCatmullPoints* apresentada em baixo. A mesma gera os pontos da curva a partir dos pontos recolhidos no ficheiro XML. Para a geração desses pontos recorreu-se à função *getGlobalCatmullRomPoint* que nos permite obter as coordenadas do próximo ponto da curva para um dado valor t, como foi explicado anteriormente. Deste modo ao aplicar-se um ciclo com o valor t de 0 até 1 e incremento 0.01 passamos por 100 pontos da curva.

```
1 //...
2 for(float i = 0; i < 1; i+=0.01){
3     getGlobalCatmullRomPoint(i, ponto, deriv);
4     Point *p = new Point(ponto[0], ponto[1], ponto[2]);
5     pointsCurve.push_back(p);
6 }
7 //...
```

2.2.4 Group

Tal como na segunda fase do projeto, esta classe será responsável por guardar toda a informação de um grupo, nomeadamente as transformações geométricas, as primitivas e ainda os grupos filhos.

2.2.5 Camera

Relativamente à fase anterior não foram modificadas nenhuma das funcionalidades da câmera, pelo que esta classe se mantém igual à apresentada na segunda fase do projeto.

2.2.6 Parser

- **Rotate**

Para a implementação da nova forma de rotação, foi necessário adicionar uma variável *time* à classe Transformation, como podemos observar na função *void parseRotate (Group* group, XMLElement* element)* que se encontra em baixo definida na classe *Parser*. A nova variável indica o tempo, em segundos, necessários para uma rotação de 360 graus.

```
//...
if(element->Attribute("time")){
    float time = stof(element->Attribute("time"));
    angle = 360 / (time * 1000);
    type = "rotateTime";
}
//...
```

- **Translate**

Para além da rotação, também foi necessário implementar uma nova forma de translação. Para tal, foi necessário recorrer a uma variável *time*. Este campo pretende representar o tempo que uma determinada figura ou grupo demora a percorrer a curva definida através dos pontos de controlo contidos dentro do nodo translate. O excerto apresentado de seguida, pertencente à função *void parseTranslate (Group *group, XMLElement *element)*, mostra as alterações efetuadas no *parserTranslate* anterior:

```
//...
if (element->Attribute("time")){
    bool deriv = false;
    if (element->Attribute("derivative"))
        deriv = (stoi(element->Attribute("derivative"))== 1) ? true :
            false;
```

```
6       time = stof(element->Attribute("time"));
7       time = 1 / (time * 1000);
8       element = element->FirstChildElement("point");
9
10      while (element != nullptr){
11          x = stof(element->Attribute("X"));
12          y = stof(element->Attribute("Y"));
13          z = stof(element->Attribute("Z"));
14
15          Point *p = new Point(x,y,z);
16          cPoints.push_back(p);
17          element = element->NextSiblingElement("point");
18      }
19
20      t = new Transformation(time,cPoints,deriv,"translateTime");
21      group->addTransformation(t);
22  }
23  //...
```

2.2.7 Engine

Através de utilização de transformações como a *translate* e o *rotate* foi possível permitir o movimento dos planetas sobre outro corpo ou sobre si própios, respetivamente. Assim, de forma a possibilitar que o utilizador pare o movimento dos planetas, o grupo optou por acrescentar três variáveis globais nesta classe, nomeadamente:

- stop - Flag que indica se os planetas deverão estar em movimento
- eTime - Calcula o tempo que passou quando o movimento está ativo
- cTime - Tempo utilizado para a execução da função applyTransformations

Com o intuito de fornecer informação acerca dos *frames per second*, decidimos criar mais duas variáveis globais nesta classe:

- frame - guarda o número de frames desde o último cálulo de *fps*
- timebase - guarda o tempo do último cálculo de *fps*

De forma a obter *frames per second*, é necessário calcular a quantidade de frames que passaram durante um segundo. Para conseguir obter o intervalo de tempo que passou foi necessário recorrer à *glutGet(GLUT_ELAPSED_TIME)* que nos indica o tempo, em milisegundos, desde que a aplicação iniciou. Assim, a diferença entre este tempo e o *timebase* permite-nos conhecer o tempo que passou desde o último cálculo de *fps*. A informação obtida acerca das *fps* será apresentada no título da janela. Tal como pode ser observado de seguida na função *fps*.

```
void fps() {
    int time;
    char name[30];
    frame++;
    time = glutGet(GLUT_ELAPSED_TIME);
    if (time - timebase > 1000) {
        float fps = frame * 1000.0/(time - timebase);
        timebase = time;
        frame = 0;
        sprintf(name,"SOLAR SYSTEM  %.2f FPS",fps);
        glutSetWindowTitle(name);
    }
}
```

- **Rotation**

Nesta classe também foi definida a função *applyTransformation* que com base no parser feito, irá fazer o strcmp para aplicar a transformação necessária. O tempo será retirado do ficheiro XML que terá o formato:

```
<translate time="25" >
    <point X="30.000000" Y="0.000000" Z="0.000000" />
    <point X="27.716387" Y="0.000000" Z="11.480503" />
    <point X="21.213203" Y="0.000000" Z="21.213203" />
    <point X="11.480503" Y="0.000000" Z="27.716387" />
    <point X="0.000000" Y="0.000000" Z="30.000000" />
    <point X="-11.480503" Y="0.000000" Z="27.716387" />
    <point X="-21.213203" Y="0.000000" Z="21.213203" />
    <point X="-27.716387" Y="0.000000" Z="11.480503" />
    <point X="-30.000000" Y="0.000000" Z="0.000000" />
    <point X="-27.716387" Y="0.000000" Z="-11.480503" />
    <point X="-21.213203" Y="0.000000" Z="-21.213203" />
```

```
13      <point X="-11.480503" Y="0.000000" Z="-27.716387" />
14      <point X="-0.000000" Y="0.000000" Z="-30.000000" />
15      <point X="11.480503" Y="0.000000" Z="-27.716387" />
16      <point X="21.213203" Y="0.000000" Z="-21.213203" />
17      <point X="27.716387" Y="0.000000" Z="-11.480503" />
18  </translate>
19  <rotate time="10" X="0" Y="1" Z="0" />
```

Neste caso, estamos perante o caso "rotateTime"na última linha do XML, pois contém o *time* na sua informação, assim, utilizamos a função *glutGet(GLUT_ELAPSED_TIME)*. Com o valor desta função bem como o ângulo guardado da transformação, ao multiplicar estes iremos obter ângulos diferentes à medida que o tempo passa. Assim, é possível fazer com que a primitiva gire à volta de si mesma. Com as variáveis globais e este novo ângulo calculado, apresentamos como é definida esta rotação:

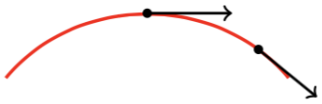
```
1  if(!strcmp(type,"rotateTime")){
2          float nA = eTime * angle;
3          glRotatef(nA,x,y,z);
4  }
```

- **Translate**

Nesta fase também optamos por desenhar as órbitas de cada planeta utilizando para este efeito os pontos gerados pela fórmula Catmull-Rom. À medida que o tempo passa, fazemos o planeta deslocar-se ao longo da sua curva, ao aplicar esta translação. Para ser então possível esta movimentação, recorremos mais uma vez à função *glutGet(GLUT_ELAPSED_TIME)*, que devolve o tempo passado desde que houve a inicialização, e ainda à função *getGlobalCatmullRomPoint*, como podemos verificar no excerto de código apresentado:

```
1  if(!strcmp(type,"translateTime")){
2      float p[4], deriv[4];
3      float dTime = eTime *time;
4      t->getGlobalCatmullRomPoint(dTime,p,deriv);
5      drawOrbits(t);
6      glTranslatef(p[0],p[1],p[2]);
7      //...
8  }
```

A função *applyTransformation* também terá uma condição que será utilizada para definir a trajetória do cometa. Este, para além de se mover segundo a sua trajetória, pretendia-se que fosse possível manter na orientação da curva, mantendo-a tangente, tal como representado:



Assim, o ficheiro XML deverá conter a informação da seguinte forma, para se reconhecer que se pretende a utilização de orientação:

```
<translate time="500" derivative="1" >
    <point X="250.000000" Y="0.000000" Z="0.000000" />
    <point X="0.000000" Y="0.000000" Z="250.000000" />
    <point X="-250.000000" Y="0.000000" Z="0.000000" />
    <point X="-100.000000" Y="0.000000" Z="-100.000000" />
    <point X="0.000000" Y="0.000000" Z="-250.000000" />
</translate>
```

Com este ficheiro, irá obter-se a derivada do ponto da curva. Para que o cometa esteja orientado, é necessário criar uma matrix que irá ser utilizada na função *glMultMatrixf*, mantendo o cometa alinhado com a curva. O *vetorY* inicial toma os valores [1,0,0]. Este vetor será guardado em cada tranformação sendo diferente para cada uma. Assim, apresentamos o excerto que permite esta orientação do cometa.

```
if(t->getDeriv()){
    float res[4];
    t->normalize(deriv);
    t->cross(deriv,t->getVetor(),res);
    t->normalize(res);
    t->cross(res,deriv,t->getVetor());
    float matrix[16];
    t->normalize(t->getVetor());
    t->rotMatrix(matrix,deriv,t->getVetor(),res);

    glMultMatrixf(matrix);
}
```

- **Movimento dos planetas**

Para que exista movimentação dos sistema, são utilizados os *rotates* e *translates*, que já foram explicados anteriormente. Esta movimentação é feita através da variação do tempo e também é necessário existir uma variável que nos diga se os planetas estão parados ou em movimento. Assim, criaram-se três variáveis que foram apresentadas em cima.

Desta forma a variável *stop* poderá tomar o valor 0 ou 1, significando se está ou não ativo. Com isto, na função *applyTransformation* é verificado o valor desta variável e caso o movimento dos planetas tenha sido parado por preferência do utilizador, toma o valor 1, não sendo calculada a variação no tempo, o que faz com que o sistema pare.

3 Demonstração

De forma a correr e demostrar todas as aplicações referidas, iremos apresentar todos os passos para executar cada uma destas.

- **Generator**

```
$ cd generator
$ mkdir build && cd build
$ cmake ..
$ make
$ ./generator torus 0.5 3 20 20 torus.3d
$ ./generator sphere 3 20 20 sphere.3d
$ ./generator -patch teapot.patch 10 teapot.3d
```

Figura 1. Generator

O projeto contém a pasta *files* que contém todos os ficheiros *XML* e onde serão criados os ficheiros do torus e da esfera .3d que serão lidos para gerar o Sistema Solar. Nesta fase ainda é criado o cometa com uma tecelagem de 10, como é apresentado de seguida.

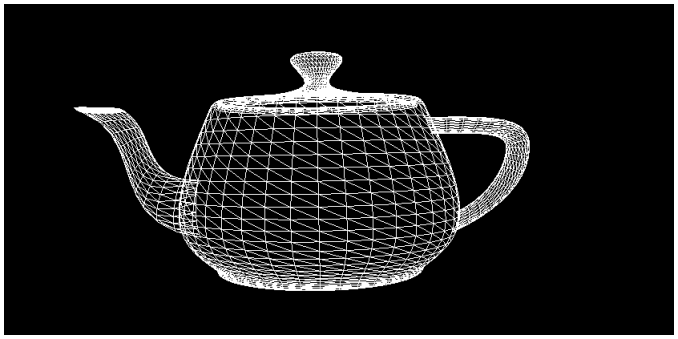


Figura 2. Cometa

- **Engine**

```
$ cd engine
$ mkdir build && cd build
$ cmake ..
$ make
$ ./engine SolarSystem.xml
```

Figura 3. Engine

Como podemos visualizar na Figura 3, irá ser passado como parâmetro o ficheiro *XML* correspondente ao Sistema Solar.

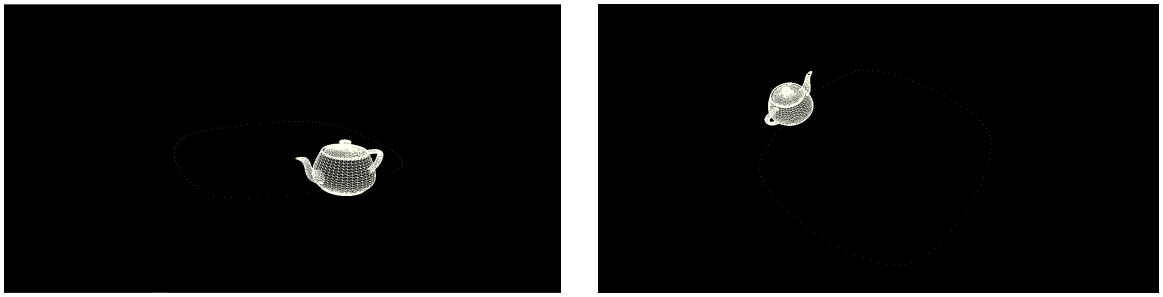


Figura 4. Cometa com orientação

É apresentado de seguida o ficheiro *XML* que irá originar o exemplo do cometa apresentado na Figura 4 que segue a sua trajetória definida pela curva Catmull-Rom e este apresenta a direção para a qual se irá movimentar.

```
<scene>
  <group>
    <group>
        <translate time="10" derivative="1" >
             <point X="10.000000" Y="0.000000" Z="0.000000" />
             <point X="0.000000" Y="0.000000" Z="10.000000" />
             <point X="-10.000000" Y="0.000000" Z="0.000000" />
             <point X="-0.000000" Y="0.000000" Z="-10.000000" />
        </translate>
        <models>
           <model file="teapot.3d" />
        </models>
```

```
13      </group>
14    </group>
15  </scene>
```

3.1 Usabilidade

Foi mantido o menu utilizado nas outras fases que permite ao utilizador conhecer os comandos com os quais poderá interagir com o sistema. Este mostra como o utilzador pode manipular o Sistema Solar de modo a visualizar o que pretende.

```
#_____________________________ HELP _____________________________#
| Usage: ./engine {XML FILE}                                      |
|                 [-h]                                            |
|   FILE:                                                         |
| Specify a path to an XML file in which the information about    |
| the models you wish to create are specified                     |
|                                                                 |
|   ↑ : Rotate your view up                                       |
|                                                                 |
|   ↓ : Rotate your view down                                     |
|                                                                 |
|   ← : Rotate your view to the left                              |
|                                                                 |
|   → : Rotate your view to the right                             |
|                                                                 |
|   F1 : Increase image                                           |
|                                                                 |
|   F2 : Decrease image                                           |
|                                                                 |
|   F6 : Reset zoom                                               |
|                                                                 |
|   FORMAT:                                                       |
|   F3: Change the figure format into points                      |
|                                                                 |
|   F4: Change the figure format into lines                       |
|                                                                 |
|   F5: Fill up the figure                                        |
#_________________________________________________________________#
```

Figura 5. Menu

3.1.1 Menu do Sistema Solar

De modo a tornar o nosso sistema mais interativo, é possível o utilizador parar ou acionar os movimentos do sistema, bem como, sair do mesmo. Para aceder a este menu,

basta pressionar o lado direito do rato. Os sub-menus do menu *System movements* foram criados de forma *responsive*. Neste menu será possível:

- Parar ou Retomar movimentos do sistema
- Fechar janela

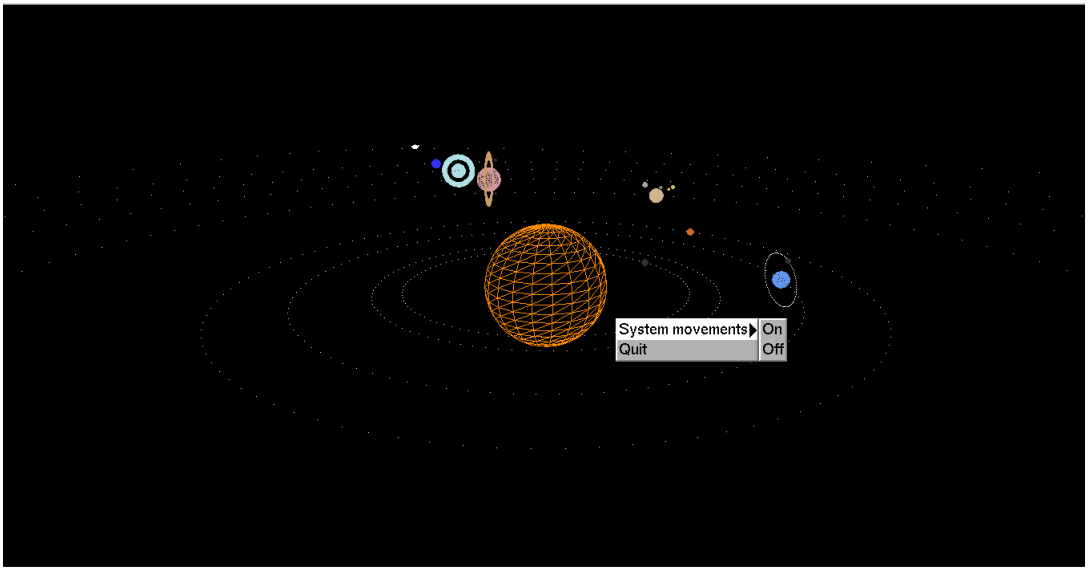


Figura 6. Menu do Sistema Solar

3.2 Sistema Solar

Em baixo é apresentado o exemplo do Sistema Solar que serviu de base para a criação do nosso.

Assim, é ilustrado o Sistema Solar desenvolvido até agora. Este contém todos os planetas, os seus satélites naturais e ainda um cometa, que contêm órbitas definidas por curvas de Catmull Rom.

Ilustramos também como fica o produto final quando usamos a opção de preencher com cores o nosso Sistema Solar.

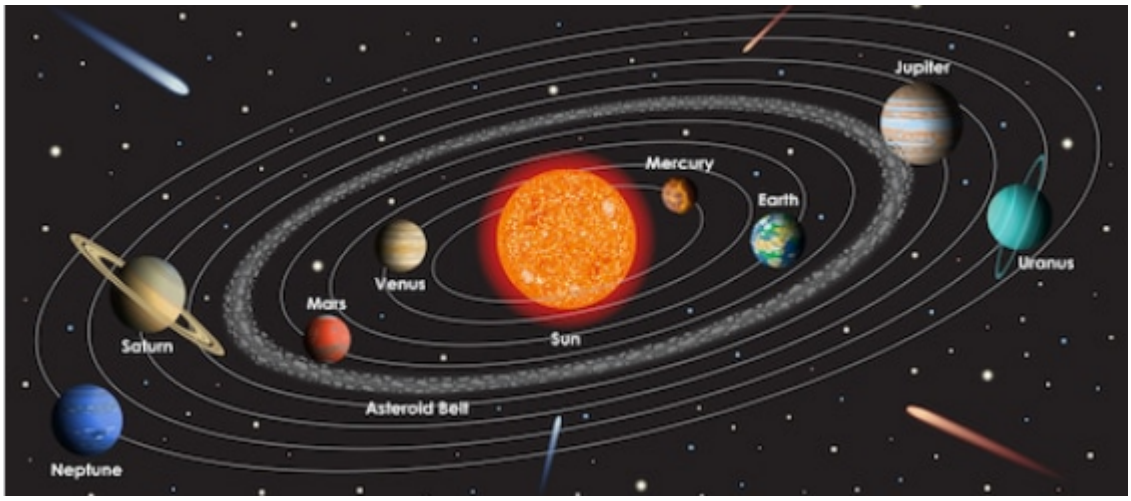


Figura 7. Sistema Solar

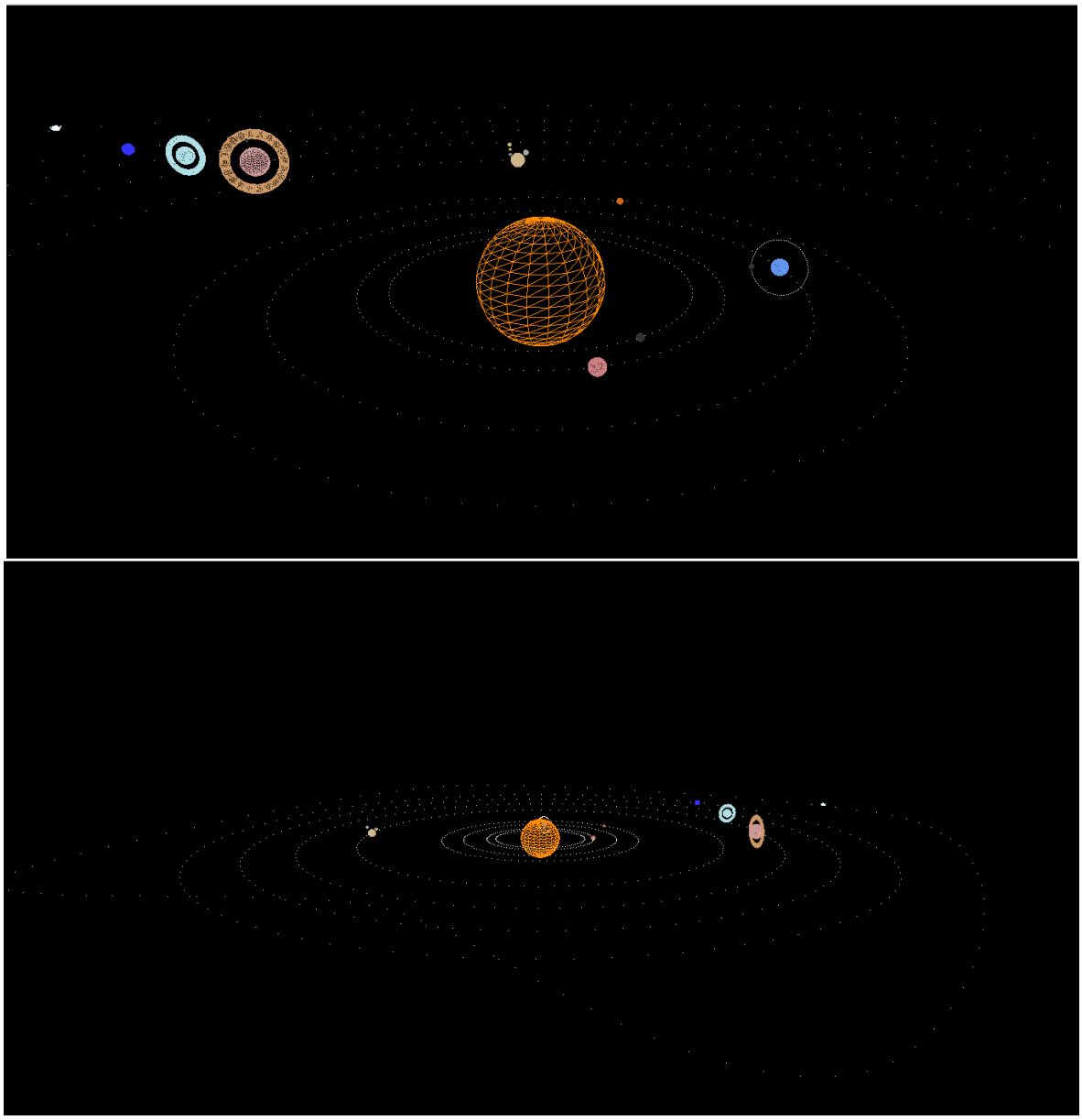


Figura 8. Sistema Solar

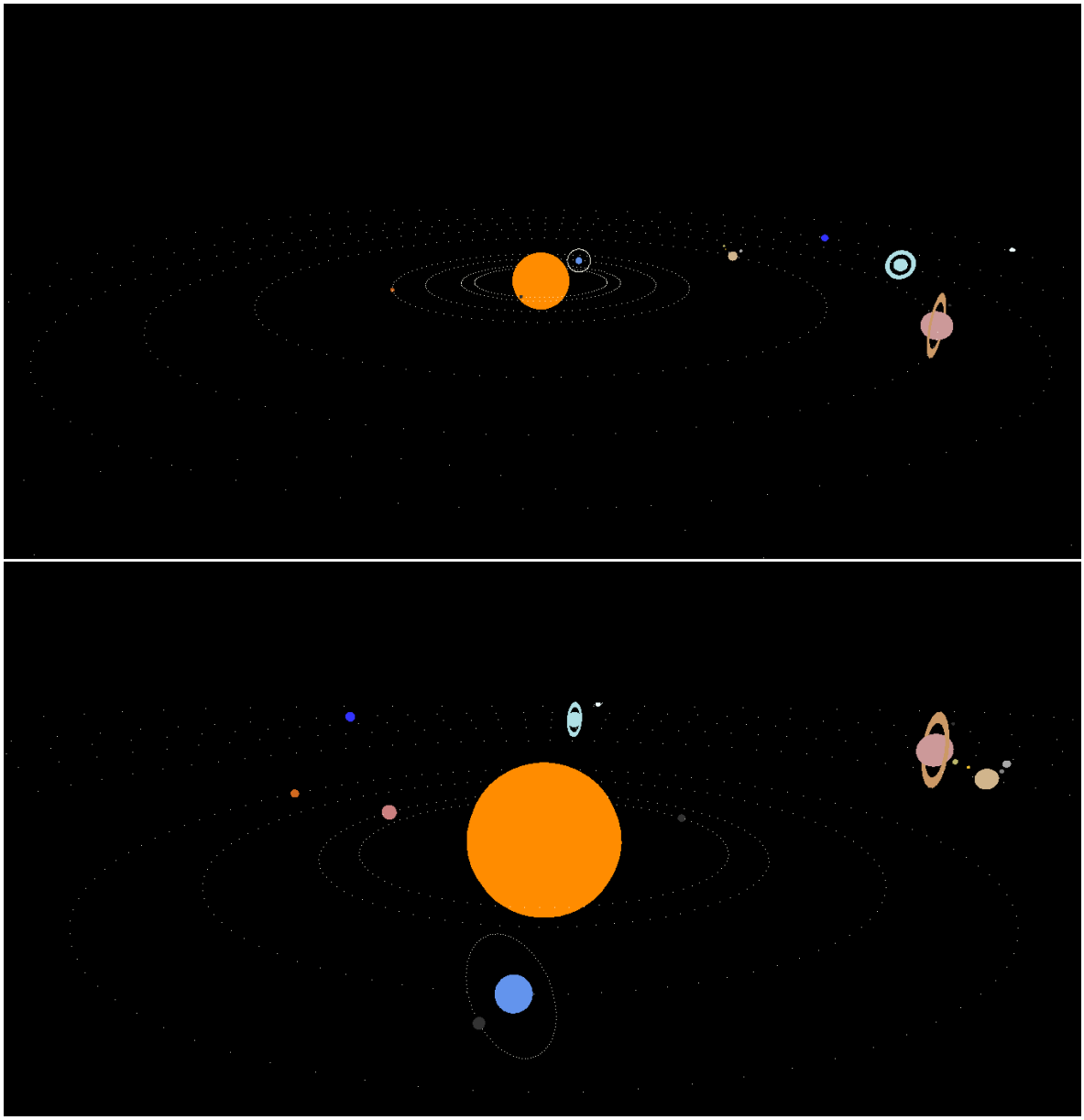


Figura 9. Sistema Solar preechido com cores

4 Conclusão

Nesta fase do projeto foi necessário manipular VBOs, recorrer a curvas de Catmull-Rom e ainda utilizar patch de Bézier, o que se revelou um desafio dado que nunca tínhamos tido contacto com estes. Assim, o grupo teve de efetuar um trabalho de pesquisa de forma a compreender como cada um funciona, para poder aplicar da forma mais correta.

Tal como na fase anterior foi necessário adicionar e alterar algumas classes, com o intuito de cumprir todas as metas de forma organizada e estruturada.

Em última instância consideramos que apesar de todas as dificuldades encontradas, os objetivos para esta fase foram atingidos, uma vez que conseguimos implementar um modelo dinâmico tal como pretendido e ainda foram adicionados alguns pontos extras que achamos importantes no âmbito do trabalho. Contudo espera-se que na última fase seja possível obter um resultado mais realista do sistema solar.