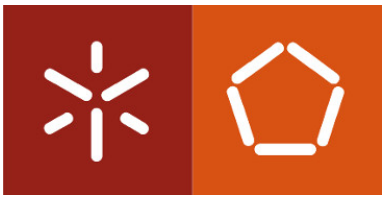


Computação Gráfica

Trabalho Prático - Fase 3

2 de maio de 2021

Patrícia Pereira, A89578

Meriem Khammasi, A85829

Jaime Oliveira, A89598

Luís Magalhães, A89528

Conteúdo

1 *Abstract*

Terceira fase do trabalho prático realizado no âmbito da Unidade Curricular de Computação Gráfica da Universidade do Minho. Esta fase consiste na adição de novas funcionalidades e na alteração do funcionamento do trabalho já realizado, com o intuito de conseguir obter um programa que seja capaz de efetuar transformações mais complexas aos objetos e que também os ilustre de forma mais eficaz.

Adicionou-se uma nova figura ao *generator*, o *Teapot*, que será usado principalmente para a criação de uma primitiva que irá mover em orbita no nosso sistema solar.

2 Introdução

Neste relatório vamos esclarecer os aspetos mais importantes relativamente à terceira fase do nosso trabalho, onde faremos uma síntese e explicação do código elaborado, tal como os resultados obtidos.

Iremos também descrever e explicar as abordagens que tomamos quanto à realização do *generator* e do *engine*, bem como o seu funcionamento apresentando algumas imagens de exemplo para facilitar a compreensão. Iremos também especificar os diversos raciocínios utilizados para ultrapassar as barreiras encontradas.

3 Organização do Código

Tendo em conta a implementação realizada na primeira fase do projeto, optamos por seguir a abordagem estabelecida anteriormente, pelo que continuamos a ter duas aplicações, o *Generator* e o *Engine*.

3.1 *Generator*

Tal como na fase anterior, este será responsável por gerar os pontos dos triângulos que permitem construir as diversas figuras geométricas. Para além das primitivas elaboradas nas fases anteriores foi nos exigido construir a figura *Teapot*.

3.1.1 *Teapot*

A última primitiva a ser adicionada ao gerador foi o *Teapot*. Contudo, o que diferencia esta primitiva das restantes é a forma de como ela é criada, visto que esta toma como referência um ficheiro "**.patch**"no qual é armazenada toda a informação para que seja possível representar a forma usando *Patches de Bezier*.

O ficheiro que foi mencionado está localizado na pasta *"files"* juntamente com os restantes ficheiros *".xml"* e *".3d"*. Dentro do ficheiro é possível encontrar um conjunto de vértices juntamente com um conjunto de índices que indicam a ordem pela qual os vértices são invocados para desenhar cada *patch* (deve ser mencionado que por si, os indices estão

agrupados em conjuntos de 16 índices, porque são necessários ao todo 16 vértices para desenhar um só *"patch"*).

A função *teapot* do ficheiro *generator.cpp* recebe como parâmetros um *int* que corresponde ao nível de subdivizão desejado e uma string que indica o nome do ficheiro "*.patch*"e tal como as funções das restantes primitivas retorna um vetor de pontos. A função começa no ínicio a interpretar o ficheiro .patch e armazenas todos os índices num vetor de *int*s e os respetivos vértices são armazenados num vetor de pontos, de seguida é efetuado um return de um vetor de pontos que contém todos os pontos ordenados de forma a que seja possível desenhar a superfície da chaleira seguindo a sua ordem.

Este último vetor é criado usando a função createBezierSurf, essa recebe como parâmetros os vetores dos índices e dos vértices dos patches de Bezier, e cria um vetor de pontos para desenhar a superfície da chaleira. Para criar este vetor é necessário usar 3 *for loop*s intercalados: o loop exterior percorre 16 elementos do vetor de indices por cada iterção; o loop do meio vai desenhado colunas do patch em cada iteração e o loop interior desenha 1 par de triângulos por cada iteração.

```
for(k=0; k < ni; k++){

  for(i = 0; i < tess; i++){

    for(j = 0; j < tess; j++){

      a0 = bezierPoint(u, ctrl[ind[k*16]], ctrl[ind[k*16+1]], ctrl[ind[k*16+2]], ctrl[ind[k*16+3]]);
      a1 = bezierPoint(u, ctrl[ind[k*16+4]], ctrl[ind[k*16+5]], ctrl[ind[k*16+6]], ctrl[ind[k*16+7]]);
      a2 = bezierPoint(u, ctrl[ind[k*16+8]], ctrl[ind[k*16+9]], ctrl[ind[k*16+10]], ctrl[ind[k*16+11]]);
      a3 = bezierPoint(u, ctrl[ind[k*16+12]], ctrl[ind[k*16+13]], ctrl[ind[k*16+14]], ctrl[ind[k*16+15]]);

      b0 = bezierPoint(u+intv, ctrl[ind[k*16]], ctrl[ind[k*16+1]], ctrl[ind[k*16+2]], ctrl[ind[k*16+3]]);
      b1 = bezierPoint(u+intv, ctrl[ind[k*16+4]], ctrl[ind[k*16+5]], ctrl[ind[k*16+6]], ctrl[ind[k*16+7]]);
      b2 = bezierPoint(u+intv, ctrl[ind[k*16+8]], ctrl[ind[k*16+9]], ctrl[ind[k*16+10]], ctrl[ind[k*16+11]]);
      b3 = bezierPoint(u+intv, ctrl[ind[k*16+12]], ctrl[ind[k*16+13]], ctrl[ind[k*16+14]], ctrl[ind[k*16+15]]);

      p1 = bezierPoint(v, a0, a1, a2, a3);
      p2 = bezierPoint(v, b0, b1, b2, b3);
      p3 = bezierPoint(v+intv, a0, a1, a2, a3);
      p4 = bezierPoint(v+intv, b0, b1, b2, b3);

      points.push_back(p1);
      points.push_back(p2);
      points.push_back(p4);

      points.push_back(p4);
      points.push_back(p3);
      points.push_back(p1);

      u += intv;
    }
    u = 0;
    v += intv;
  }
  v = 0;
  u = 0;
}
```

Figura 1: Segmento do código da função *createBezierSurf*

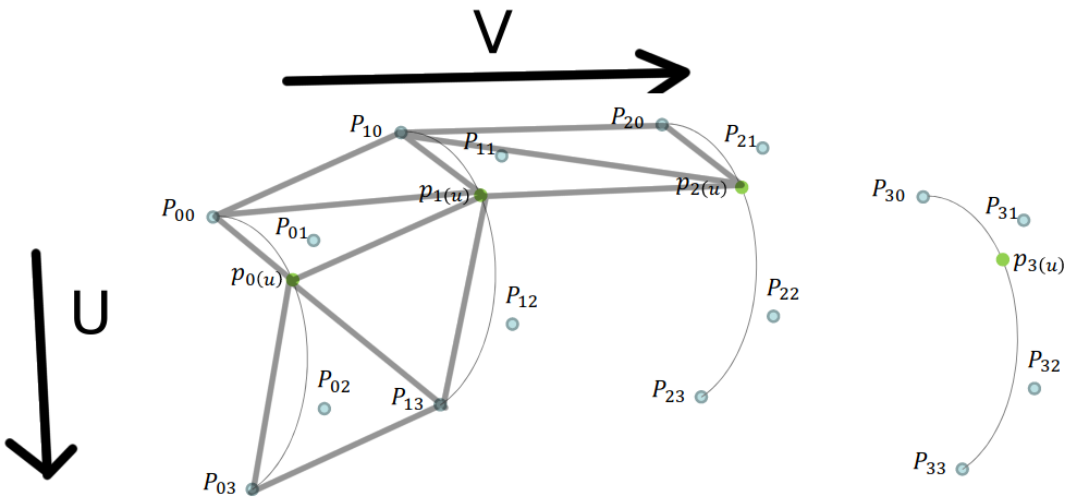


Figura 2: Esquema da construção do *patch* através dos *for loops*

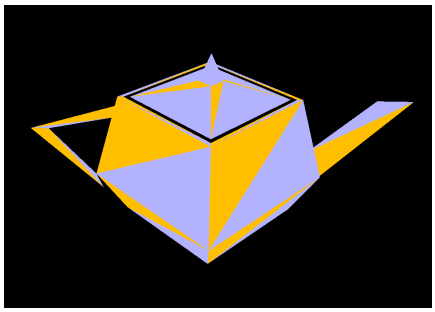
Dentro do *for loop* interior é invocada várias vezes a função *bezierPoint*. Esta função é tem como objetivo calcular um ponto desenhado de uma determinada curva de Bezier (sendo esta defenida por 4 pontos como também por um valor de 0 a 1 para indicar a posição do ponto desejado). A função é constuída de maneira a respeitar a seguinte fórmula:

$$P(t) = t^3 * p0 + 3 * t^2 * (1-t) * p1 + 3 * t * (1-t)^2 * p2 + (1-t)^3 p3 \qquad (1)$$

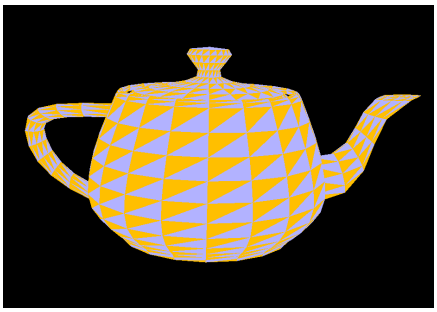
```
Point bezierPoint(float t, Point p0, Point p1, Point p2, Point p3){

  Point p;
  float it = 1.0f - t;
  float at = pow(it, 3);
  float bt = 3 * t * pow(it, 2);
  float ct = 3 * pow(t, 2) * it;
  float dt = pow(t, 3);

  p.x = (at*p0.x) + (bt*p1.x) + (ct*p2.x) + (dt*p3.x);
  p.y = (at*p0.y) + (bt*p1.y) + (ct*p2.y) + (dt*p3.y);
  p.z = (at*p0.z) + (bt*p1.z) + (ct*p2.z) + (dt*p3.z);

  return p;
}
```

Figura 3

3.1.2 Interpretação do *Teapot* no *Engine*



Level 1 tessellation.



Level 5 tessellation.

Level 20 tessellation.

3.2 *Engine*

De modo implementar as funcionalidades esperadas para esta fase, decidimos que deveríamos alterar a estruturação do código já elaborado. Desta forma, realizamos implementar *VBOs* e curvas de *Catmull-Rom*, tal como outras alterações de modo a permitir a movimentação dos planetas. Assim, passaremos de seguida a explicar mais detalhadamente estas modificações.

3.2.1 Point

Nas outras fases do projeto optamos por criar uma estrutura *Points*, que continuamos a recorrer à sua utilização no *Generator* e que passou a ser chamada de *Pt* para a sua utilização na classe *geoTransforms*. Contudo consideramos que deveríamos criar uma classe para armazenar as coordenadas de um ponto.

```
Point::Point(){
    x = 0;
    y = 0;
    z = 0;
}

Point::Point(float px, float py, float pz){
    x = px;
    y = py;
    z = pz;
}

float Point::getX(){
    return x;
}

float Point::getY(){
    return y;
}

float Point::getZ(){
    return z;
}
```

Figura 4: Classe Point.

3.2.2 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 diretamentena 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 armazenaar o número de vértices que irá conter. De seguida, na *PrepareBuffer* é gerado e preenchido o *buffer* do tipo *GLuint*, depois guardamos na memória da placa gráfica *numVertex* * 3 vértices.

```
Shape::Shape(){}

Shape::Shape(vector<Point*> vertex){
    numVertex = vertex.size();
    prepareBuffer(vertex);
}

void Shape::prepareBuffer(vector<Point*> vertex){
    int index = 0;
    float* vertexs = new float[vertex.size() * 3];
    for(vector<Point*>::const_iterator vertex_it = vertex.begin(); vertex_it != vertex.end(); ++vertex_it){
        vertexs[index++] = (*vertex_it)->getX();
        vertexs[index++] = (*vertex_it)->getY();
        vertexs[index++] = (*vertex_it)->getZ();
    }
    glGenBuffers(1,bufferVertex);
    glBindBuffer(GL_ARRAY_BUFFER, bufferVertex[0]);
    glBufferData(GL_ARRAY_BUFFER,
                 sizeof(float) * numVertex * 3,
                 vertexs,
                 GL_STATIC_DRAW);
    delete [] vertexs;
}

void Shape::draw(){
    glBindBuffer(GL_ARRAY_BUFFER, bufferVertex[0]);
    glVertexPointer(3, GL_FLOAT, 0, 0);
    glDrawArrays(GL_TRIANGLES, 0, numVertex * 3);
}
```

Figura 5: *Shape*.

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

3.2.3 Transformations

Nesta fase 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.

Deste modo, iremos conseguir obter um sistema solar dinâmico. Para tal,implementamos a função *getGlobalCatmullRomPoint* que perimitirá 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.

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.

```
void Transformations::getCatmullRomPoint(float t, int *indexes, float *p, float *deriv) {
    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();
    }

    // Compute A = M * P
    float a[4][4];
    multMatrixVector(*m, px, a[0]);
    multMatrixVector(*m, py, a[1]);
    multMatrixVector(*m, pz, a[2]);

    // Compute pos = T * A
    float T[4] = { t*t*t, t*t, t, 1};
    multMatrixVector(*a, T, p);

    // Compute deriv = T' * A
    float Tdev[4] = { 3*T[1] , 2*T[2], 1 , 0};
    multMatrixVector(*a, Tdev, deriv);
}

void Transformations::getGlobalCatmullRomPoint(float gt, float *p, float *deriv) {
    int num = controlPoints.size();
    float t = gt * num;
    int index = floor(t);

    t = t - index;

    int indexes[4];
    indexes[0] = (index + num - 1)  % num;
    indexes[1] = (indexes[0]+1)     % num;
    indexes[2] = (indexes[1]+1)     % num;
    indexes[3] = (indexes[2]+1)     % num;

    getCatmullRomPoint(t, indexes, p, deriv);
}

void Transformations::setCatmullPoints(){
    float ponto[4];
    float deriv[4];

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

Figura 6: *Curvas Catmull-Rom.*

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. geoTransforms

Tal como na segunda fase do projeto, esta classe será responsável por guardar toda a informação sobre as translações, rotações e escalas que serão usadas para transformar o grupo num vetor de *"Transformations"* e todos os astros secundários que também sofrem as transformações mencionadas, mas que possuem transformações, cores, primitivas e filhos próprios num vetor da mesma classe. engine

- Translate

Para a implementação da nova forma de translação, 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.

```
void readTranslate(geoTransforms *group, XMLElement *element){
    float x=0, y=0, z=0, time = 0;
    vector<Point*> cPoints;
    Transformations *t;

    if (element->Attribute("time"))
    {
        bool deriv = false;
        if (element->Attribute("derivative"))
            deriv = (stoi(element->Attribute("derivative"))== 1) ? true : false;
        time = stof(element->Attribute("time"));
        time = 1 / (time * 1000);
        element = element->FirstChildElement("point");

        while (element != nullptr)
        {
            x = stof(element->Attribute("X"));
            y = stof(element->Attribute("Y"));
            z = stof(element->Attribute("Z"));

            Point *p = new Point(x,y,z);
            cPoints.push_back(p);

            element = element->NextSiblingElement("point");
        }

        t = new Transformations(time,cPoints,deriv,"translateTime");
        group->addTransformations(t);
    }
    else{

        if(element->Attribute("X"))
            x = stof(element->Attribute("X"));

        if(element->Attribute("Y"))
            y = stof(element->Attribute("Y"));

        if(element->Attribute("Z"))
            z = stof(element->Attribute("Z"));

        t = new Transformations("translate",x,y,z,0.0f);
        group->addTransformations(t);

    }
}
```

Figura 7: *Translate*.

- Rotate

Para a implementação da nova forma de rotação, foi necessário adicionar uma variável time à classe *Transformations*, esta indica o tempo, em segundos, necessários para uma rotação de 360 graus.

```
void readRotate (geoTransforms *group, XMLElement *element){
    float angle = 0, x = 0, y = 0, z = 0;
    string type = "rotate";
    Transformations *t;

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

    if(element->Attribute("X"))
        x = stof(element->Attribute("X"));

    if(element->Attribute("Y"))
        y = stof(element->Attribute("Y"));

    if(element->Attribute("Z"))
        z = stof(element->Attribute("Z"));

 t = new Transformations(type,x,y,z, angle);
    group->addTransformations(t);
}
```

Figura 8: *Rotate*.

3.2.4 Representação do sistema solar

Através de utilização de transformações como o *rotate* e a *translate* foi possível permitir o movimento dos planetas sobre outro corpo ou sobre si próprios, respetivamente.

Assim, de forma a possibilitar que o utilizador pare o movimento dos planetas, o grupo optou poor acrescentar três variáveis globais nesta classe, nomeadamente:

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

Com intuito de fornecer mais informação a cerca dos *frames per second*, decidimos criar mais duas varáveis globais:

- *timebase* - Guarda o tempo do último cálculo de fps.
- *frame* - Guarda o número de frames desde o 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 *time base* 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.

```
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);
    }
}
```

Figura 9: *FPS*.

Movimentação do Sistema 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.

4 Demonstração

Para demonstrar as aplicações previamente referidas, iremos demonstrar de que forma somos capazes de executar cada uma delas: -*Generator*

```
meriam@meriam-E205SA:~$ cd Desktop/CGFase2/Generator/
meriam@meriam-E205SA:~/Desktop/CGFase2/Generator$ g++ generator.cpp  -o
generator -lGL -lGLU -lglut
meriam@meriam-E205SA:~/Desktop/CGFase2/Generator$ ./generator torus 3 2
10 20 torus.3d
```

Figura 10: *Generator*.

O projeto inclui uma pasta files que contém todos os ficheiros XML e onde serão criados os ficheiros do *torus*, da esfera.3d e do teapot.3d que posteriormente serão lidos para a geração do Sistema Solar.

Como podemos visualizar na Figura anterior, irá ser passado como parâmetro o ficheiro XML de todo o Sistema Solar resultante da leitura dos ficheiros .3d previamente gerados.

5 Sistema Solar Final

Em baixo é apresentado o Sistema Solar desenvolvido até agora. Este contém todos os planetas, os seus satélites naturais e ainda um teapot, 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.

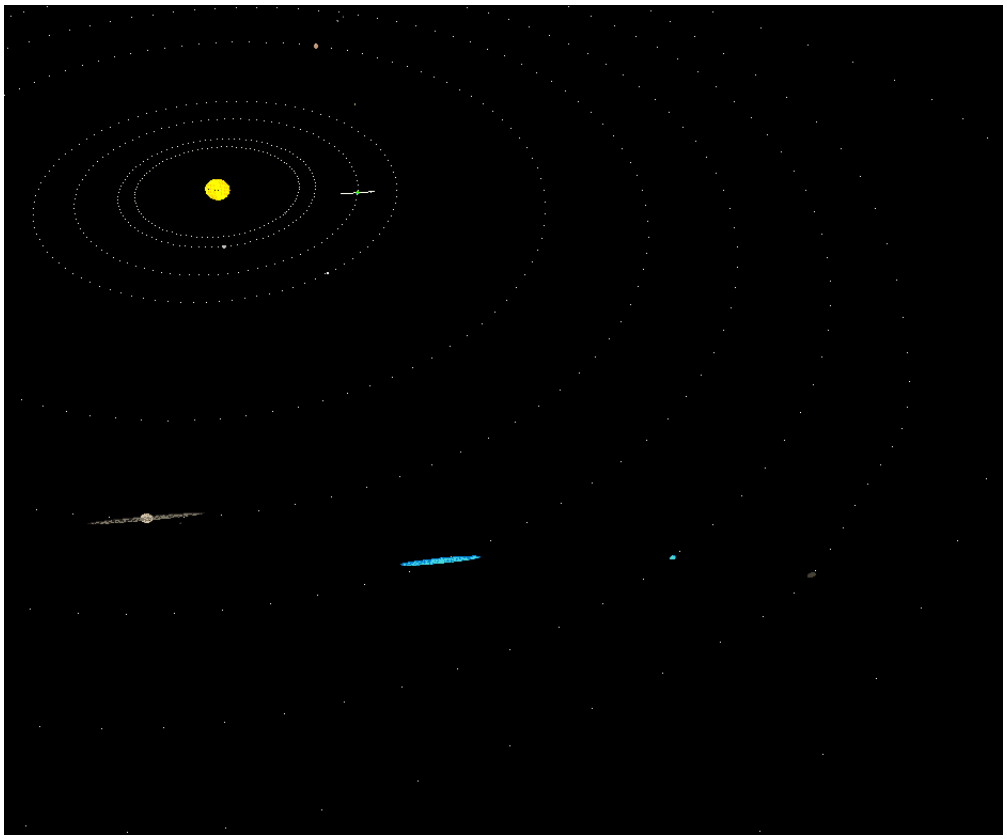


Figura 11: *SistemFinal.*

6 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 pes-quisa de forma a compreender como cada um funciona, para poder aplicar da forma mais correta. Na realização desta fase, devido à complexidade da estrutura usada nas fases anteriores e à grande quantidade de informação armazenada decidimos alterar as estruturas, sendo ao nível do *engine* feita uma reformulação do código. De modo a construirmos o Sistema Solar foi necessário a criação

de um novo ficheiro XML pelo que foram realizadas alterações na realização do *parsing* do mesmo. Foi necessária a implementação da primitiva *Teapot*, de modo a conseguir uma representação mais realista e dinamica do Sistema Solar. Em última instância, esperamos que o resultado obtido nesta fase corresponda às expectativas e sirva de suporte para a elaboração da próxima etapa do projeto.