

Universidade do Minho
Engenharia Informática

Trabalho Prático
Computação Gráfica
2º Semestre - 2021/2022

RELATÓRIO CG – Fase 3

Abril 2022

Diogo da Costa e Silva Lima Rebelo (A93180)

Diogo Miguel da Silva Araújo (A93313)

Joel Costa Araújo (A76603)

Francisco Peixoto (A84668)

Resumo

O presente documento serve como base de acompanhamento à realização da terceira fase do projeto de computação gráfica, onde nos foram propostos os objetivos de criar um novo tipo de modelo baseado em “Bezier patches”, de evoluir as transformações de rotação e translação para aceitarem tempos de duração e executar animações com elas em curvas de Catmull-Rom. Como último parâmetro a concretizar, os modelos utilizados passam a ser desenhados com VBO's (*vertex buffer object*).

Dadas as metas como cumpridas, o objetivo final é apresentar uma cena que represente um sistema solar dinâmico, incluindo um cometa construído utilizando *bezier patches*.

Índice

Resumo	2
Índice.....	3
Índice de figura	4
1.Introdução	5
2.Correções à fase anterior	6
<i>Parser</i> de xml	6
Estrutura simples	6
Estrutura aninhada (c/ luas).....	7
Conteúdo extraído	7
3.Generator	8
Patch file	8
Construção do modelo patch.....	8
4.Engine	10
VBO's.....	10
Transformations.....	10
Rotate.....	11
Scale	12
Translate	13
Desenho das primitivas.....	16
Apresentação FPS	17
5.Resultados	18
6.Conclusão.....	19

Índice de figura

Figura 1 - Exemplo de um corpo celestial sem luas	6
Figura 2 - Exemplo corpo celestial com satélite natural	7
Figura 3 - Classe Primitive	7
Figura 4 - Exemplificação do formato de um patch file, fornecido pelos docentes	8
Figura 5 - Calcular curva	9
Figura 6 - processo de declaração de VBO.....	10
Figura 7 - Class Rotation.....	11
Figura 8 - Apply Rotation	11
Figura 9 - exemplo xml de varios rotates a serem aplicados a um modelo.....	12
Figura 10 - Exemplo XML scale	12
Figura 11 - Classe & Apply Scale	13
Figura 12 - Translation Apply() e Pontos de translação XML	13
Figura 13 - função auxiliar para calcular pontos das orbitas	14
Figura 14 - Desenhar primitivas	16
Figura 15 - update FPS	17
Figura 16 - Apresentação dos FPS.....	17
Figura 17 - Resultado final do Sistema Solar.....	18
Figura 18 - Resultado final do Sistema Solar.....	18

1. Introdução

O intuito da fase três deste projeto visa evoluir o trabalho até então realizado por forma a conseguir construir um modelo dinâmico do sistema solar, com movimento/animação, ao contrário do que acontecia nas fases anteriores, onde tudo era estático.

Seguindo as ideias propostas para o efeito, numa primeira fase foi necessário alterar o nosso *generator* para conseguir criar um novo tipo de modelos baseado em *Bazier patches*, utilizando os pontos de controlo associados, os quais são posteriormente utilizados para construir as superfícies desejadas.

Por sua vez, no *engine* foi necessário refazer algumas tarefas menos bem executadas na fase anterior, às quais se seguiram a evolução daquilo que é uma transformação de uma primitiva, nomeadamente em relação às rotações e translações, que agora apresentam novas características como o tempo e pontos que definem uma curva a seguir, no caso da translação. Como tarefa final proposta, os modelos passam também a ser desenhados com VBO's (*vertex buffer object*), ao contrário do modo imediato usado até então.

Por forma a documentar todas estas alterações aplicadas ao nosso projeto, segue-se neste presente documento todas as informações para que se possa acompanhar o processo de aplicação das mesmas.

2. Correções à fase anterior

Parser de xml

Como atentado numa das observações do relatório anterior, o *parser* construído para ler o ficheiro XML relativo à nossa configuração do sistema solar precisava de ser refeito, visto que estava organizado com uma estrutura bastante rígida e orientada somente para um sistema solar com número de corpos predefinido, o que não seria o objetivo, pois pretendemos que a nossa solução aplicacional seja capaz de construir outro tipo de cenas/modelos. De modo análogo, classes previamente construídas com nomenclatura de “CelestialBody” e “Orbit”, por exemplo, deixam de existir, visto que não seguem o alinhamento das ideias que pretendemos executar.

Estrutura simples

Para representar uma figura simples, isto é, sem outras figuras anexadas, devem ser usados dois grupos, um referente à órbita e outro às especificações da figura em si. Durante essa leitura, as informações percorridas são associadas a um objeto designado por **Primitive**.

```
<!-- Sol (Origem) -->
<group>
  <!-- Orbit -->
  <rotate angle = 1 axisY = 1/>

  <group>

    <rotate axisY = 1 timeR = 19.97/>
    <scale scaleX = 14 scaleY = 14 scaleZ = 14/>

    <!-- <color R = 253 G = 184 B = 19 /> Sun / Yellow Orange -->
    <color R = 1 G = 0.549 B = 0/> <!-- Dark Orange -->
    <models>
      <!-- Models, Name-->
      <model file = "sphere.3d" texture = "sun" />
    </models>
  </group>
</group>
```

Figura 1 - Exemplo de um corpo celestial sem luas

Estrutura aninhada (c/ luas)

Caso existam um ou mais satélites naturais a orbitar ao redor desse planeta, uma nova primitiva é criada para cada um. Assim, a primitiva relativa ao planeta contém um vetor de primitivas onde armazena as primitivas das luas, permitindo assim processar figuras anexadas a uma hierarquia superior.

```
<group>
  <!-- Orbits -->
  <rotate angle = 7.155 axisY = 1 />
  <orbit radX = 32 radZ = 31/>

  <!-- Planeta -->
  <group>
    <!-- Pontos de translação (Orbita do planeta)-->
    <translate timeT = 365.25 align = True>...
    </translate>

    <!-- Inclinação do planeta -->
    <rotate angle = 23.5 axisX = 1 />
    <!-- Rotação do planeta -->
    <rotate axisY = 1 timeR = 29.78/>
    <scale scaleX = 1.3 scaleY = 1.3 scaleZ = 1.3 />
    <!-- Deep Sky Blue <color R = 0 G = 191 B = 255/> -->
    <color R = 0.0 G = 0.749 B = 1 />

    <models> ...
  </models>
</group>

<!-- Lua -->
<group>
  <!-- Orbits -->
  <rotate angle = 1 axisY = 1/>
  <orbit radX = 1.75 radZ = 1.75/>

  <group>
    <!-- Pontos de translação (Orbita da lua)-->
    <translate timeT = 27 align = True> ...
    </translate>
    <!-- Rotação da Lua -->
    <rotate axisY = 1 timeR = 27/>
    <scale scaleX = 0.4 scaleY = 0.4 scaleZ = 0.4/>
    <!-- Gray <color R = 255 G = 0 B = 0/> -->
    <color R = 0.502 G = 0.502 B = 0.502 />

    <models>
      <model file = "sphere.3d" texture = "Earth_Moon" />
    </models>
  </group>
</group>
</group>
```

Figura 2 - Exemplo corpo celestial com satélite natural

Conteúdo extraído

Visto que é deste *parser* que retiramos toda a informação necessária à configuração do nosso modelo, é necessário armazenar e persistir esses dados. Para tal, esta operação de leitura é capaz de criar um conjunto de objetos, proclamados de *Primitive*, cujas variáveis intrínsecas visam armazenar as informações relativas ao que foi lido, assim como proceder à leitura dos ficheiros 3d associados a cada primitiva, guardando os seus pontos, entre outros.

```
Primitive::Primitive(){
    filename = "";
    textureFilename = "";
    vBuffer[0] = 0;
    vBuffer[1] = 0;
    nPoints = 0;
    nIndexes = 0;
    r = g = b = 0.0f;
    transformations = vector<Transformation*>();
    appendedPrimitives = vector<Primitive>();
}
```

Figura 3 - Classe Primitive

3. Generator

Patch file

Nesta terceira fase, o *generator* recebe uma tarefa de construir um novo modelo baseado em *Bazier patches*, utilizando por base um ficheiro com uma estrutura de apoio bem definida, a qual identificamos brevemente de seguida.

Example:

```
2 <- number of patches
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
3, 16, 17, 18, 7, 19, 20, 21, 11, 22, 23, 24, 15, 25, 26, 27
28 <- number of control points
1.4, 0, 2.4 <- control point 0
1.4, -0.784, 2.4 <- control point 1
0.784, -1.4, 2.4 <- control point 2
0, -1.4, 2.4
1.3375, 0, 2.53125
1.3375, -0.749, 2.53125
```

indices for the first patch

indices for the second patch

Figura 4 - Exemplificação do formato de um patch file, fornecido pelos docentes

Deste modo, temos então o início do ficheiro com a indicação do número de linhas que identificarão diferentes patches, onde cada um indica, por sua vez, os índices dos pontos utilizados para a sua construção, pontos esses que são apresentados posteriormente à indicação do número de pontos que existem no ficheiro.

Construção do modelo patch

Para proceder à construção do nosso novo modelo, o generator agora contém um novo ficheiro, *patch.cpp*, cujo intuito é ler os ficheiros *patch* e proceder à sua construção.

Para o cálculo dos mesmos, utilizamos curvas cúbicas, isto é, formadas por quatro vértices. Os vértices inicial e final são onde a curva efetivamente passa, servindo os restantes apenas como atratores da curva.

Seguindo também o conteúdo lecionado nas aulas teóricas, para calcular os pontos fazemos uso de matrizes, nomeadamente a seguinte equação:

$$B(u, v) = U * M * P * M^T * V$$

, tendo por base que

$$U = [u^3 \ u^2 \ u \ 1]$$
$$V = [v^3 \ v^2 \ v \ 1]$$
$$M = M^T = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

e que P representa a matriz 4x4 contendo os 16 pontos de controlo fornecidos no ficheiro *patch*.

Deste modo, o que precisamos de fazer foi representar cada um dos vetores apresentados, assim como proceder às suas multiplicações.

Por fim, de modo a construir as superfícies de *Bezier*, iteramos sobre o conjunto de pontos calculados por forma a configurar os triângulos necessários para construir a figura em si.

```
void Patch::calculateCurve(vector<Point_3D> *result, int patchLevel, float u, float v, float interval){  
  
    Point_3D p01 = calculatePatchVertex(patchLevel, u, v);  
    Point_3D p02 = calculatePatchVertex(patchLevel, u, v + interval);  
    Point_3D p03 = calculatePatchVertex(patchLevel, u + interval, v);  
    Point_3D p04 = calculatePatchVertex(patchLevel, u + interval, v + interval);  
  
    /**  
     *      2 ----- 4  
     *      |         |  
     *      |         |  
     *      1 ----- 3  
     */  
  
    result->push_back(p01);  
    result->push_back(p04);  
    result->push_back(p02);  
  
    result->push_back(p04);  
    result->push_back(p01);  
    result->push_back(p03);  
}
```

Figura 5 - Calcular curva

4. Engine

VBO's

Até agora, para desenhar cada uma das figuras presentes no nosso modelo tínhamos de aceder a uma lista, guardada em memória, que continha os pontos que definiam cada uma das mesmas, procedendo posteriormente ao respetivo desenho. Contudo, através de VBO's, *vertex buffer object*, somos capazes de mandar esse conjunto de vértices para serem armazenados na memória gráfica e, quando necessário, renderizados diretamente pelo dispositivo de vídeo, o que proporciona um aumento de desempenho em relação ao modo de renderização imediato.

Para atingir este objetivo, após efetuar o *parse* de todos os vértices no ficheiro 3d relativo a cada figura, geramos um *buffer*, deixamos as próprias funções disponibilizadas pelo *OpenGL* atribuir-lhe um identificador único e, por fim, atribuímos-lhe os pontos a serem guardados (libertando, assim, o espaço alocado em memória).

```
glEnableClientState(GL_VERTEX_ARRAY);  
// VBO  
glGenBuffers(1, &vBuffer[0]);  
glBindBuffer(GL_ARRAY_BUFFER, vBuffer[0]);  
glBufferData(GL_ARRAY_BUFFER, arraySize * sizeof(float), points, GL_STATIC_DRAW);  
glDisableClientState(GL_VERTEX_ARRAY);  
free(points);
```

Figura 6 - processo de declaração de VBO

Transformations

Por forma a generalizar as diferentes transformações que são possíveis de aplicar a qualquer figura, decidimos configurar uma classe abstrata, *Transformation*, cuja função virtual pura se designa por *Apply()*, a qual é obrigatoriamente definida em qualquer classe que se designe sob esta. Essa função tem como objetivo aplicar a respetiva transformação ao sistema.

Rotate

A rotação é a primeira que recebe uma novidade em relação à sua versão anterior, visto que também tem em conta fatores como o tempo durante o qual é suposto aplicar a transformação. Deste modo, quando não há tempo, uma simples chamada da função **glRotatef(angle, x, y, z)** aplica a transformação. No entanto, quando temos o tempo (<rotate axisY = 1 timeR = 35>), antes de fazer a chamada a função **glRotatef()** é necessário calcular o movimento a aplicar ao modelo sobre o eixo indicado. Neste caso, o atributo timeR representa a duração de uma rotação de um corpo sobre o eixo indicado. Assim, através da contagem do tempo desde a função glutInit até ao presente pela função glutGet(GLUT_ELAPSED_TIME), é possível efetuar o cálculo do novo ângulo, obtendo o ângulo final ao somar esse resultado com o ângulo inicial estipulado pelo utilizador aquando da criação do ficheiro xml.

```
Rotation::Rotation(float angle, float x, float y, float z, float time){
    this->angle = angle;
    this->x = x;
    this->y = y;
    this->z = z;
    this->time = time;
}
```

Figura 7 - Class Rotation

```
void Rotation::Apply(){
    if(!time){
        glRotatef(angle, x, y, z);
    }
    else{
        float elapsed, newAngle;
        elapsed = glutGet(GLUT_ELAPSED_TIME) % (int)(time * 1000);
        newAngle = (elapsed*360) / (time * 1000);

        glRotatef(angle+newAngle, x, y, z);
    }
}
```

Figura 8 - Apply Rotation

```

<group>
  <!-- Orbita -->
  <rotate angle = 38.6 axisY = 1/>
  <orbit radX = 27 radZ = 25/>

  <!-- Venus -->
  <group>
    <translate timeT = 224.7 align = True >...
  </translate>

    <rotate angle = 177 axisX = 1 />
    <rotate axisY = 1 timeR = 35.02/>
    <scale scaleX = 1.2 scaleY = 1.2 scaleZ = 1.2 />
    <!-- Pearl <color R = 234 G = 224 B = 200/> -->
    <color R = 0.918 G = 0.878 B = 0.784 />

    <models>
      <!-- Models, Name-->
      <model file = "sphere.3d" name = "Venus"/>
    </models>
  </group>
</group>

```

Figura 9 - exemplo xml de varios rotates a serem aplicados a um modelo

Scale

O scale é a transformação mais simples presente no nosso projeto, visto que apenas precisa de três valores referentes a cada um dos três eixos x, y e z.

Deste modo, a sua definição da função **Apply()** fica facilmente definida aquando da chamada da função **glScalef (x, y , z)**.

```

<rotate axisY = 1 timeR = 19.97/>
<scale scaleX = 14 scaleY = 14 scaleZ = 14/>

<!-- <color R = 253 G = 184 B = 19 /> Sun / Yellow Orange -->
<color R = 1 G = 0.549 B = 0/> <!-- Dark Orange -->
<models>
  <!-- Models, Name-->
  <model file = "sphere.3d" texture = "sun" />
</models>

```

Figura 10 - Exemplo XML scale

```

Scale::Scale(float x, float y, float z){
    this->x = x;
    this->y = y;
    this->z = z;
}

void Scale::Apply(){
    glScalef(this->x, this->y, this->z);
};

```

Figura 11 - Classe & Apply Scale

Translate

O translate é a transformação que mais mereceu a nossa atenção pelo ganho de complexidade em relação à versão anterior.

Analogamente à transformação da rotação, uma translação pode, ou não, apresentar indicação de tempo. Quando se refere a uma translação simples, somente com os valores de x,y e z, a aplicação do **Apply()** apenas faz uso da função **glTranslatef(x,y,z)**.

Contudo, uma translação, nesta versão, pode receber valores de pontos que definem uma curva Catmull-Rom cúbica sobre a qual a figura em si deverá seguir no seu movimento de translação e um valor do tempo que se pretende aplicar a esse movimento. Como último parâmetro temos o *align*, podendo ter valores “true” ou “false”, que designam se o objeto deverá seguir a orientação da curva, ou não.

```

if (time != 0 && catmullCurvePoints.size() != 0){
    // y axis
    static float yAxis[3] = {0.0f, 1.0f, 0.0f};
    float pos[3], zAxis[3], deriv[3];
    float rotationMatrix[16]; // easier because of c++ matrix mult.

    float elapsedT = glutGet(GLUT_ELAPSED_TIME) % (int)(time * 1000);
    float globalT = elapsedT / (time * 1000);

    drawCurve();

    getCatmullPoint(globalT, pos, deriv);
    glTranslatef(pos[0], pos[1], pos[2]);
    normalizeVector(deriv);
    normalizeVector(yAxis);
    vectorsCross(deriv, yAxis, zAxis);
    vectorsCross(zAxis, deriv, yAxis);
    // x, y, z, rotMatrix
    calculateRotMatrix(deriv, yAxis, zAxis, rotationMatrix);
    glmMultMatrixf(rotationMatrix);
}
else {
    glTranslatef(this->x, this->y, this->z);
}

```

```

<!-- Planeta -->
<group>
    <!--Pontos de translação (Orbita do planeta)-->
    <translate timeT = 365.25 align = True>
        <point X = 32.00000 Z = -0.00000 />
        <point X = 30.90880 Z = -8.02637 />
        <point X = 27.70962 Z = -15.50535 />
        <point X = 22.62065 Z = -21.92686 />
        <point X = 15.98896 Z = -26.85296 />
        <point X = 8.26681 Z = -29.94769 />
        <point X = -0.01913 Z = -30.99999 />
        <point X = -8.30376 Z = -29.93810 />
        <point X = -16.02208 Z = -26.83443 />
        <point X = -22.64769 Z = -21.90065 />
        <point X = -27.72874 Z = -15.47325 />
        <point X = -30.91868 Z = -7.99058 />
        <point X = -31.99998 Z = 0.03706 />
        <point X = -30.89888 Z = 8.06216 />
        <point X = -27.69047 Z = 15.53742 />
        <point X = -22.59358 Z = 21.95304 />
        <point X = -15.95581 Z = 26.87146 />
        <point X = -8.22986 Z = 29.95724 />
        <point X = 0.05738 Z = 30.99995 />
        <point X = 8.34070 Z = 29.92846 />
        <point X = 16.05518 Z = 26.81585 />
        <point X = 22.67470 Z = 21.87441 />
        <point X = 27.74781 Z = 15.44113 />
        <point X = 30.92852 Z = 7.95477 />
    </translate>
</group>

```

Figura 12 - Translation Apply() e Pontos de translação XML

Como extra para obter os pontos que pretendíamos da translação, optamos por fazer uma função que pudesse automatizar os cálculos da curva. Ao calcular a órbita achamos melhor

que estas não fossem circulares, mas sim elípticas ($\text{radX} \neq \text{radY}$) para tentar representar uma imagem mais fiel à realidade.

```
aux = 0.261899
t = 0
for t in range(0,24):
    r = ((math.cos(t*aux) * radX , (((math.sin(t*aux))* radZ)*-1) + 30))
    print("<point X = ", r[0], " Z = " ,r[1], " />")
```

Figura 13 - função auxiliar para calcular pontos das orbitas

Catmull-Rom cubic curve

Os pontos que podem ser atribuídos à translação de um objeto são referentes a curvas Catmull-Rom cúbicas, ou seja, há, no mínimo, quatro pontos a serem apresentados.

Procedendo à leitura e armazenamento dos mesmos, resta avançar com os respectivos cálculos.

O cálculo associado às curvas segue os seguintes indicadores,

$$P(t) = T * M * P,$$
$$T = [t^3 \ t^2 \ t \ 1],$$
$$M = \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ 0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix},$$

onde P representa os quatro pontos de controlo da curva.

Para o cálculo da derivada, basta substituir o vetor T pela sua derivada,

$$T' = [3t^2 \ 2t \ 1 \ 0].$$

Para aplicar estes conhecimentos, basta representar cada um dos vetores/matrizes e aplicar a respetiva multiplicação.

Por fim, de modo a alinhar a orientação de cada objeto com a orientação da curva, em si, foi necessário definir um sistema de eixos para ser considerado em cada figura. Começamos por definir qual seria o eixo do X, escolhendo a tangente à curva para esse efeito, ou seja, a derivada da curva em cada ponto.

Observação: para os cálculos abaixo, é necessário normalizar ambos os vetores Y e derivada.

Para determinar o eixo Z, aplicamos o produto vetorial aos vetores da derivada e UP da figura, ou seja, $y = [0 \ 1 \ 0]$. Por fim, e aplicando a mesma linha de raciocínio, para descobrir o Y real, reaplicamos o produto vetorial aos eixos X e Z.

Chegados aos valores, basta configurar a matriz de rotação e aplicá-la ao nosso modelo.

$$\begin{matrix} & X & Y & Z \\ R = & \begin{bmatrix} & & & 0 \\ & & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} & , X = P'(t) \end{matrix}$$

Desenho das primitivas

Chegando à parte de desenhar as primitivas começamos por chamar a função de OpenGL ***glPushMatrix()*** para posteriormente aplicar as transformações de cada primitiva e, em seguida desenha-las.

Neste seguimento, vamos aplicar a cor atribuída ao objeto, aplicamos cada uma das transformações de acordo com a ordem estipulada no xml. Aplicadas as transformações, vamos aceder ao VBO onde estão guardados os pontos da primitiva e apenas recorreremos à chamada da função ***glDrawArrays()*** que trata do processo de desenho. Na eventualidade de uma primitiva ter outras anexadas (neste caso, satélites naturais), damos uso à recursividade da mesma função para desenhar cada um desses objetos, apenas fazendo ***glPopMatrix()*** no final da recursividade.

```
void Primitive::Draw(){

    int totalPoints = nIndexes * 3;

    glPushMatrix();
    glColor3f(r, g, b);

    for (Transformation* t : transformations){
        t->Apply();
    }

    glEnableClientState(GL_VERTEX_ARRAY);
    // Para adicionar cores a cada triângulo, tem de

    // Dar bind ao identificador associado da primitiva
    glBindBuffer(GL_ARRAY_BUFFER, vBuffer[0]);
    // Definir o modo de leitura do VBO (3 vértices por índice)
    glVertexPointer(3, GL_FLOAT, 0, 0);
    // Draw da primitiva, a começar no índice 0, nPoints
    glDrawArrays(GL_TRIANGLES, 0, totalPoints);
    glDisableClientState(GL_VERTEX_ARRAY);

    for (Primitive p : appendedPrimitives){
        p.Draw();
    }

    glPopMatrix();
}
```

Figura 14 - Desenhar primitivas

Apresentação FPS

De modo a apresentar ao utilizador a taxa de quadros por segundo (*FPS*, *framerate per second*), criamos duas variáveis de modo a controlar a passagem do tempo e um contador de *frames*. Quando a diferença entre a variável de tempo atual e a variável de tempo base for superior a 1000 milissegundos (ou 1 segundo), são calculadas quantas *frames* foram apresentadas nesse segundo, dividindo o contador de *frames* pelo intervalo de tempo desde o último cálculo, apresentando no nome da janela o resultado obtido.

```
void updateFPS(){
    frame++;
    times = glutGet(GLUT_ELAPSED_TIME);

    if (times - timeBase > 1000) {
        fps = frame * 1000.0 / (times - timeBase);
        timeBase = times;
        frame = 0;
    }

    sprintf(title, "CG@DI-UM - Fase 3 - Grupo 11 <-> FPS: %d - Time: %d ", fps, times / 1000);
    glutSetWindowTitle(title);
}
```

Figura 15 - update FPS


 CG@DI-UM - Fase 3 - Grupo 11 <-> FPS: 286 - Time: 5

Figura 16 - Apresentação dos FPS

5. Resultados

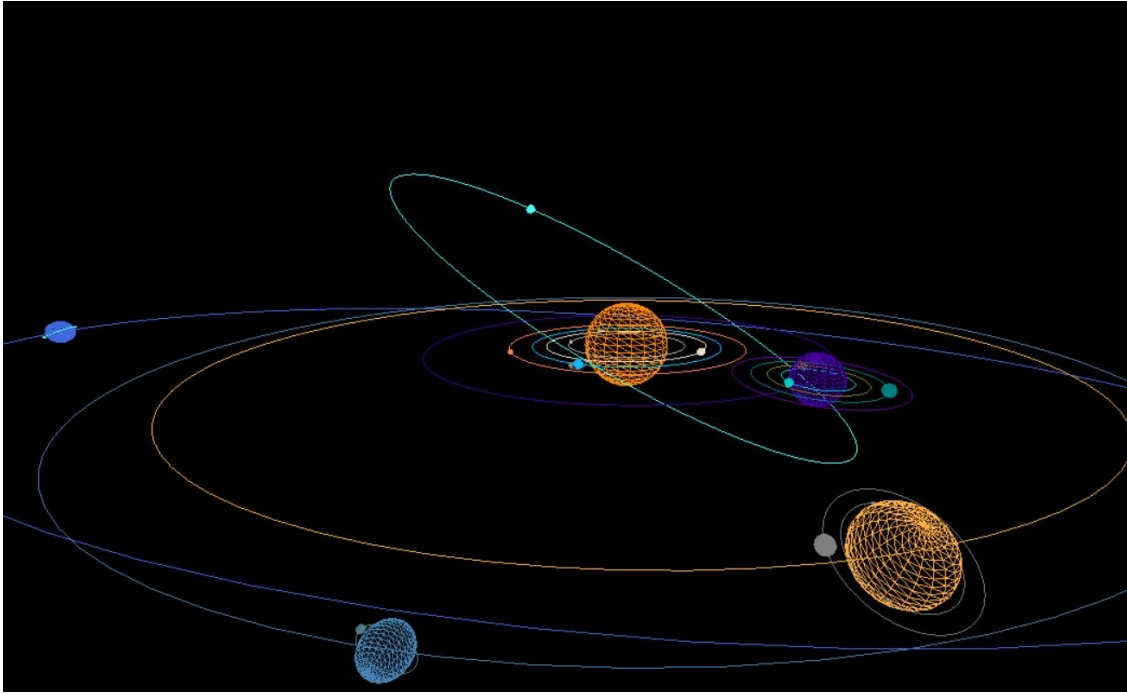


Figura 17 - Resultado final do Sistema Solar

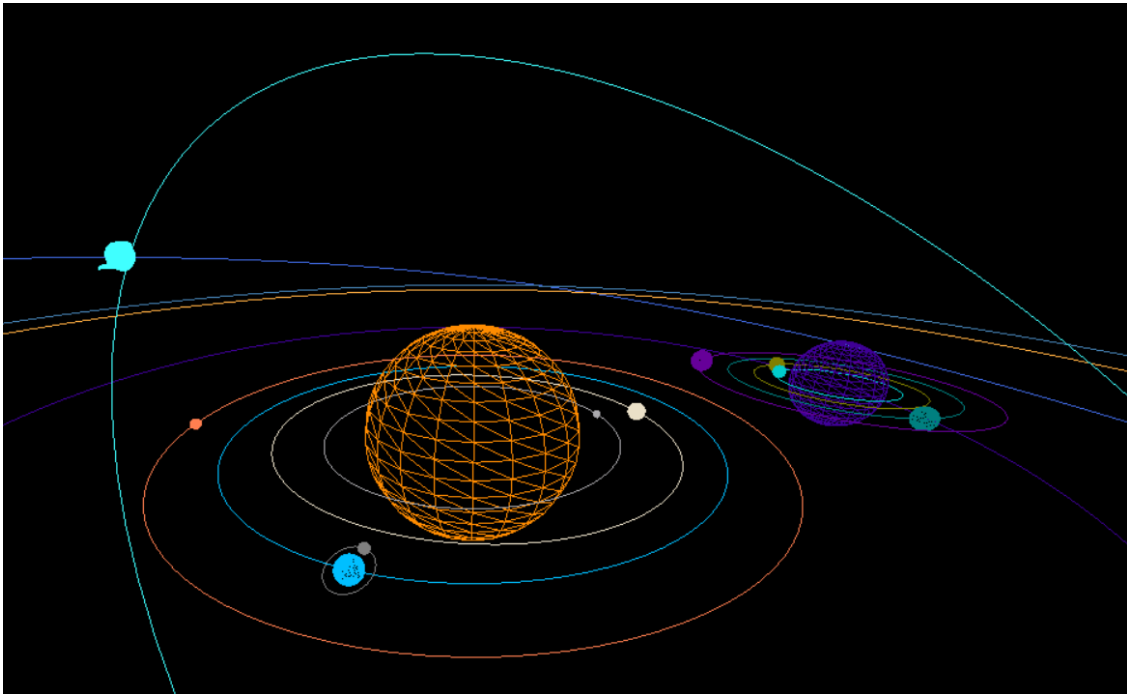


Figura 18 - Resultado final do Sistema Solar

6. Conclusão

Dado por terminada a execução desta fase, tivemos oportunidade de explorar mecanismos que aumentam a eficiência no desenho da nossa aplicação, como, por exemplo, a utilização de *VBO's*. Tivemos alguma dificuldade em meter em prática o cálculo das superfícies de *Bezier*, visto serem um processo complexo, desde o cálculo dos *patches* até à configuração das superfícies, mas esse trabalho acabou por facilitar a posterior construção das curvas de *Catmull*, o que nos entusiasmou na sua conclusão, visto já conseguirmos obter um sistema dinâmico.

Como melhorias, pensamos que há valores no *xml* que podem ser alterados de maneira a melhor configurar o modelo do sistema solar, assim como pretendemos adicionar alguma nova figura para tentar preencher mais o nosso modelo, de maneira a tornar o ambiente mais interessante e agradável.