Universidade do Minho

LEI 3ºAno 2ºSemestre

Computação Gráfica

Mini Motor 3D

Fase 3 - Curvas, Superfícies e VBOs

Daniel Caldas a67691
José Cortez a67716
Marcelo Gonçalves a67736
Ricardo Silva a67728

4 de Maio de 2015

Conteúdo

1 Introdução

1.1 Contexto

No âmbito da UC de Computação Gráfica, surge um projeto prático para consolidar os conceitos adquiridos ao longo do semestre: a criação de um mini motor 3D.

Usando as tecnologias C++ e OpenGL, será então desenvolvido o motor 3D ao longo de quatro fases, sendo que esta segunda está resumida em tópicos na subsecção seguinte.

1.2 Resumo

Nesta etapa foi efetuada:

- Implementação de **VBOs** para o desenho de todas as formas geométricas previamente definidas;
- Desenhar um **Teapot** à custa de **patches de Bézier**;
- Implementação de funções que **definem** e **desenham** curvas como elementos que definem as translações dos planetas (*Catmull-Rom Curves*) ;
- Recalcular distâncias entre planetas do sistema solar por forma a incorporar as **translações** e **rotações** (animação);
- Definir trajetórias elípticas para as translações e cálculo de pontos dessas mesmas elipse;
- Efetuar cálculos para que as rotações e translações dos planetas estejam a uma escala o mais realista possível;
- Atualização do *parser* XML por forma a aceitar ficheiros com novo tipo de informação;
- Atualização das estruturas de dados para incorporação de novas funcionalidades;
- Criação do modelo do sistema solar animado (**incluindo o "Cometa Teapot"**) e incorporação do mesmo no **Mini Motor 3D**.

2 Nota relativa ao trabalho anterior

Relativamente à etapa anterior, não foi exposta uma parte importante da nossa solução para o parser. O desenvolvimento de tal ferramenta segundo os critérios que escolhemos seguir, levou-nos a ter de no final do parser, colocar uma espécie de sinal que nos permitisse saber se o ficheiro tinha ou não terminado.

```
<?xml version="1.0" ?>
<imagem>
        <grupo prof=1>
            <translacao X=10 Y=0 Z=0 />
            <rotacao angulo=45 eixoX=1 eixoY=0 eixoZ=0 />
            <modelos>
                <modelo ficheiro="cone.3d" colorX=1 colorY=0 colorZ=1 />
            </modelos>
            <grupo prof=2>
                <translacao X=-5 Y=0 Z=10 />
                <modelos>
                    <modelo ficheiro="cone2" colorX=0 colorY=1 colorZ=1 />
                    <modelo ficheiro="esfera.3d" colorX=1 colorY=1 colorZ=0 />
                </modelos>
                <grupo prof=2>
                    <translacao X=-5 Y=0 Z=-10 />
                    <modelos>
                        <modelo ficheiro="paralel.3d" colorX=0 colorY=0 colorZ=1 />
                    </modelos>
                </grupo>
                <grupo prof=-1>
            </grupo>
        </grupo>
</imagem>
```

Figura 1: **Exemplo de um ficheiro xml, onde está assinalada a marca de fecho extraordinária**.

Como podemos ver na figura 1, existe uma marca extra no final do ficheiro.

3 Implementação de VBO's

Nesta fase, um dos objectivos era a implementação de VBOs para cada primitiva, por isso, o nosso grupo converteu todas as primitivas geométricas para VBOs. Foram, então, criados os arrays para armazenar os vértices para cada primitiva.

3.1 Formas

Para convertermos as nossas primitivas em VBOs foi necessário alterar a leitura do ficheiro de pontos. Sendo assim, foi criado o método **criarVBO**, que dado o nome de um ficheiro, lê o seu conteúdo e armazena todos os pontos num array para posteriormente ser criado/desenhado a VBO.

```
void Forma::criarVBO(string filename) {
    std::fstream file(filename, std::ios_base::in); |
    file >> nome;
    int size;
    file >> size;
    float* vertexB;

    vertexB = (float*)malloc((size) * sizeof(float));
    int indice = 0;
    glEnableClientState(GL_VERTEX_ARRAY);
    float x, y, z;
    while (file >> x >> y >> z) {
        vertexB[indice++] = x;
        vertexB[indice++] = y;
        vertexB[indice++] = z;
    }
    file.close();
    vertexCount = size;

    glGenBuffers(1, buffers);
    glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(float) * size, vertexB, GL_STATIC_DRAW);

    free(vertexB);
}
```

Figura 2: **Função criarVBO**

Como se pode observar pela Figura 2, este método está definido na classe Forma e permite criar uma VBO para todas as nossas primitivas geométricas, exeptuando o Cone que como veremos mais à frente nos obrigou a uma abordagem diferente.

Com a método de criação da VBO definido restou-nos desenvolver um método para desenhar a VBO criada. Este método, **drawVBO**, definido na classe Forma, além de desenhar a VBO, aplica-lhe todas as transformações previamente definidas, desenhando assim a primitiva no seu lugar correto.

Este método permite-nos desenhar a VBO de qualquer primitiva exceptuando do Círculo, e do Cone.

```
void Forma::drawVBO() {
    applyTransforms();
    glColor3f(color.x, color.y, color.z);

    glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);
    glVertexPointer(3, GL_FLOAT, 0, 0);
    glDrawArrays(GL_TRIANGLES, 0, vertexCount);

    glPopMatrix();
}
```

Figura 3: **Função drawVBO**

Na Figura 13 pode-se observar uma esfera desenhada à custa de uma VBO.

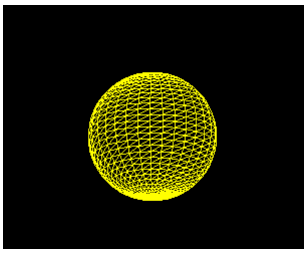

Figura 4: **Esfera desenhada através da VBO**

3.2 Círculo

Para a primitiva do Círculo, como este é desenhado à custa de **GL_TRIANGLE_FAN**, procedemos à redefinição da função **drawVBO** nesta classe, para que os Círculos fossem correctamente desenhados a partir da VBO criada. Na Figura 5 pode-se observar a função redefinida.

```
void Circulo::drawVBO() {
    applyTransforms();
    glColor3f(color.x, color.y, color.z);

    glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);
    glVertexPointer(3, GL_FLOAT, 0, 0);
    glDrawArrays(GL_TRIANGLE_FAN, 0, vertexCount);

    glPopMatrix();
}
```

Figura 5: **Função drawVBO (Círculo)**

3.3 Cone

Para a primitiva do Cone, tivemos de proceder de maneira diferente, a nossa abordagem passou por redefinir a função que cria os ficheiros com os pontos,

para que esta além de escrever os pontos no ficheiro, escreva em seguida os indices dos pontos.

```
void Cone::criarVBO(string filename) {
    std::fstream file(filename, std::ios_base::in);
    file >> nome;
    int size;
    file >> size;
    float* vertexB;

    vertexB = (float*)malloc((size)* sizeof(float));
    int indice = 0;
    glEnableClientState(GL_VERTEX_ARRAY);
    float x, y, z;
    int i = 0;
    cout << size;
    while ( i < size) {
        file >> x >> y >> z;
        vertexB[indice++] = x;
        vertexB[indice++] = y;
        vertexB[indice++] = z;
        i += 3;
    }

    i = indice = 0;
    file >> vertexCount;
    aIndex = (int*)malloc(sizeof(int)*vertexCount);
    cout << vertexCount;
    while (i < vertexCount) {
        file >> x >> y >> z;
        aIndex[indice++] = x;
        aIndex[indice++] = y;
        aIndex[indice++] = z;
        i += 3;
    }
    file.close();

    glGenBuffers(1, buffers);
    glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(float)*size, vertexB, GL_STATIC_DRAW);

    free(vertexB);
}
```

Figura 6: **Função criarVBO (Cone)**

Para que a VBO da nossa primitiva Cone seja corretamente desenhada, é necessário que os seus pontos sejam desenhados por uma ordem específica, é para isso que se utiliza o array de indices.

Em seguida redefinimos a função **criarVBO** para que esta, além de ler os pontos da figura e os armazenar num array, leia também os indices e os armazene num array de indices, para depois se proceder ao desenhar da primitiva. Foi por isso necessário a redefinição da função **drawVBO** nesta classe, para que os Cones fossem correctamente desenhados a partir da VBO criada. Na Figura 7 pode-se observar a função redefinida.

4 Teapot com Patches de Bézier

Foi necessário, tambem, criar um Teapot a partir de Patches de Bézier. Será aqui descrito tal processo.

```
void Cone::drawVBO() {
    applyTransforms();
    glColor3f(color.x, color.y, color.z);
    glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);
    glVertexPointer(3, GL_FLOAT, 0, 0);
    glDrawElements(GL_TRIANGLES, vertexCount, GL_UNSIGNED_INT, aIndex);
    glPopMatrix();
}
```

Figura 7: **Função drawVBO (Cone)**

Leitura do teapot.Patch

Neste passo é feita a leitura do ficheiro .patch segundo o formato estabelecido na página da BlackBoard da Unidade Curricular de Computação Gráfica. Á medida que o ficheiro vai sendo lido, são guardados os patches numa estrutura de dados.

```
if (ifile.is_open()) {
    ifile >> np; getline(ifile, line);

    for (i = 0; i < np && getline(ifile, line); i++) {
        Teapot pa = Teapot::Teapot();
        for (j = 0; j < 16; j++) {
            pos = line.find(",");
            token = line.substr(0, pos);
            ind = atof(token.c_str());
            line.erase(0, pos + 1);

            pa.adicionaIndice(ind);
        }
        patchs.push_back(pa);
    }
    ifile >> nv; getline(ifile, line);
    for (i = 0; i < nv && getline(ifile, line); i++) {
        for (j = 0; j < 3; j++) {
            pos = line.find(",");
            token = line.substr(0, pos);
            n = atof(token.c_str());
            line.erase(0, pos + 1);

            ponto[j] = n;
        }
        vertices.push_back(Ponto3D::Ponto3D(ponto[0], ponto[1], ponto[2]));
    }
    ifile.close();
}
```

Figura 8: Função que lê e armazena os patches de Bézier

Conversão para ficheiro .3d

Por fim, é lido a estrutura de dados na qual estão os Patch e, para cada um, são feitos os cálculos necessários (algoritmo de De Castlejau, etc.), obtendo-se os pontos finais do Teapot.

```
void initSupBezier(int tess, string nameFile) {
    ofstream file;
    file.open(nameFile);

    file << "TEAPOT\n";
    int num = patchs.size();

    file << (num*tess*tess * 6 * 3) << endl;

    for (int i = 0; i < num; i++)
        patchBezier(tess, i, file);
    file.close();
}
```

Figura 9: Primeira função a ser executada que percorre os patches todos

```
void patchBezier(int tess, int ip, ofstream& file) {
    float inc = 1.0 / tess, u, v, u2, v2;

    for (int i = 0; i < tess; i++) {
        for (int j = 0; j < tess; j++) {
            u = i*inc;
            v = j*inc;
            u2 = (i + 1)*inc;
            v2 = (j + 1)*inc;

            Ponto3D p0 = bezier(u, v, patchs[ip].getPatch());
            Ponto3D p1 = bezier(u, v2, patchs[ip].getPatch());
            Ponto3D p2 = bezier(u2, v, patchs[ip].getPatch());
            Ponto3D p3 = bezier(u2, v2, patchs[ip].getPatch());

            escreveTriangulos(p0, p2, p3, file);
            escreveTriangulos(p0, p3, p1, file);
        }
    }
}
```

Figura 10: Função que trata de cada Patch

```
Ponto3D bezier(float u, float v, vector<int> pat) {
    float bz[4][3], res[4][3];
    int i, j = 0, k = 0;

    for (i = 0; i < 16; i++) {
        bz[j][0] = vertices[pat[i]].x;
        bz[j][1] = vertices[pat[i]].y;
        bz[j][2] = vertices[pat[i]].z;

        j++;

        if (j % 4 == 0) {
            Ponto3D p = bernstein(u, bz[0], bz[1], bz[2], bz[3]);
            res[k][0] = p.x;
            res[k][1] = p.y;
            res[k][2] = p.z;

            k++;
            j = 0;
        }
    }
    return bernstein(v, res[0], res[1], res[2], res[3]);
}
```

Figura 11: Cálculos efectuados para cada Patch

5 Órbitas dos planetas

Foi necessário também redefinir as órbitas dos planetas, utilizando a função da elipse:

$$\frac{y^2}{a} + \frac{x^2}{b} = 1 \tag{1}$$

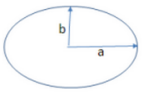

Figura 12: Elipse

Foram escolhidos, para o Mercúrio, os seguintes valores:

$$a = 42, b = 30 \tag{2}$$

A partir destes valores, foi sendo incrementado a distância entre o actual planeta e o próxima, deste modo obtendo os valores de todos os planetas.

Planeta	a	b
Mercúrio	42	30
Vénus	49	37
Terra	57	45
Marte	x	y
Júpiter	91	79
Saturno	124	112
Urano	159	147
Neptuno	199	187

Posteriormente, para cada planeta, foram introduzidos esses valores numa máquina de calcular e obteram-se pontos da elipse.

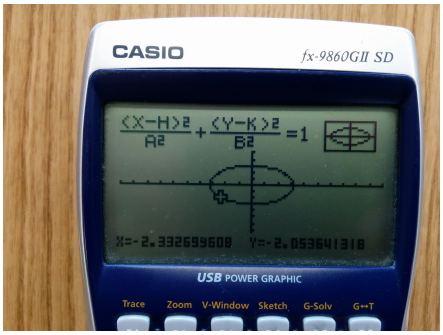

Figura 13: Exemplo da obtenção dum ponto

6 Curvas Catmull-Rom

Para animar os sistema solar foi nesta fase pedido que se implementassem funções que desenhassem curvas para definir as translações dos planetas.

```
void getCatmullRomPoint(float t, int *indices, float *res, float p[][3]) {
    int i;
    float res_aux[4];
    // catmull-rom matrix
    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 } };

    // Calcular o ponto res = T * M * P
    // sendo Pi = p[indices[i]]

    //Sem derivada
    for (i = 0; i<4; i++)
        res_aux[i] = pow(t, 3) * m[0][i] + pow(t, 2) * m[1][i] + t * m[2][i] + m[3][i];

    //Calculo do RES
    for (i = 0; i<3; i++){
        res[i] = res_aux[0] * p[indices[0]][i] + res_aux[1] * p[indices[1]][i] + res_aux[2] * p[indices[2]][i] + res_aux[3] * p[indices
    }
}

// Dado o t global, retorna o ponto na curva
void getGlobalCatmullRomPoint(float gt, float *res, float p[][3], int pc) {

    float t = gt * pc; // this is the real global t
    int index = floor(t);  // which segment
    t = t - index; // where within  the segment

    // indices store the points
    int indices[4];
    indices[0] = (index + pc - 1) % pc; indices[1] = (indices[0] + 1) % pc;
    indices[2] = (indices[1] + 1) % pc; indices[3] = (indices[2] + 1) % pc;

    getCatmullRomPoint(t, indices, res, p);
}
```

Figura 14: funcões que calculam o ponto/posição de uma forma na curva, **getGlobalCatmullRomPoint(float gt, float *res, float p[][3], int pc)** e **getGlobalCatmullRomPoint(float gt, float *res, float p[][3], int pc)**.

Na figura 14, podemos observar as funções que efetuam cálculos matriciais para obter um **ponto em função do tempo**, os parâmetros têm o seguinte significado:

- **float gt**: é o tempo global, vai incrementando à medida que é executado o programa por forma a tentar obter movimentos constantes de uma dada forma na curva;
- **float * res**: é onde se guarda o ponto resultado;
- **float p[][3]**: é uma matriz com os pontos que definem uma dada curva (no caso de um planeta serão os pontos que definem a sua trajetória elíptica);
- **pc**: é o número de pontos da matrizt **p**.

Também foi definida uma função para desenhar a curva em si, para termos uma melhor noção da **distribuição/ocupação do espaço**...
Na figura 15, podemos observar que os argumentos passados são apenas os pontos que definem a curva e o total de pontos da curva.

```
void renderCatmullRomCurve(float p[][3], int pc) {
    int i;
    glBegin(GL_LINE_LOOP);
    for (i = 0; i < pc; i++){
        glVertex3f(p[i][0],p[i][1],p[i][2]);
    }
    glEnd();
}
```

Figura 15: **void renderCatmullRomCurve(float p[][3], int pc)**.

6.1 Escala redefenida

Para tornar o nosso sistema solar o mais realista possível, foi necessário redefinir as escalas com base nos tempos reais. As escalas redefenidas foram as de translação dos planetas, em torno do sol, e foi também necessário redefinir as escalas de rotação dos planetas. Nas figuras abaixo podemos observar as respectivas tabelas.

	Dias Solares (DS)	Escala redefinida (LOG(DS;2)*2)
Mercúrio	87.969	12.88589
Vénus	224	15.61471
Terra	365	17.02351
Marte	686.98	18.84413
Júpiter	4330 (11 anos e 315 dias)	24.1603
Saturno	10765 (29 anos e 6 meses)	26.78812
Urano	30664 (84 anos e 4 dias)	29.80852
Neptuno	60225 (165 anos)	31,75615
Lua	28	9.61471

A escala utilizada é logaritmo de base 2 dos DS a multiplicar por 2.

Figura 16: **Escala de translação**

Como se pode observar pela Figura 16, nesta tabela, na segunda coluna temos o tempo real em dias solares e na terceira e ultima coluna temos os valores utilizados por nós, que resultam da aplicação de uma escala logaritmica de base 2 multiplicada por 2 sobre os valores reais apresentados na segunda coluna.

	Horas (H)	Escala redefinida (LOG(H;2)*5)
Mercúrio	1408:00	52.30
Vénus	5832:00	62.55
Terra	23:56	22.92
Marte	24:37	22.92
Júpiter	9:56	16.61
Saturno	10:15	16.61
Urano	12:14	17.92
Neptuno	16:07	20
Lua	648:00	46.70

A escala utilizada é logaritmo de base 2 das H a multiplicar por 5.

Figura 17: **Escala de rotação**

Como se pode observar pela Figura 17, nesta tabela, na segunda coluna temos o tempo real em horas e na terceira e ultima coluna temos os valores utilizados por nós, que resultam da aplicação de uma escala logaritmica de base 2 multiplicada por 5 sobre os valores reais apresentados na segunda coluna.

7 Parser XML

Nesta etapa foram também feitas alterações ao *parser* de modo a aceitar novo tipo de informação, nomeadamente:

- A translação passa a aceitar um conjunto de pontos que definem a trajetória;
- A translação passa também a receber um parâmetro tempo;
- Uma rotação recebe tal como uma translação um parâmetro tempo;

```
else if (strcmp(name, "translacao") == 0){
    TransformsWrapper tw;
    tw.nome = "TRANSLATE";

    tw.translacao.b = 0;
    tw.translacao.tempo = atoi(element->Attribute("tempo")); // Tempo da translação
    tw.translacao.first.x = tw.translacao.first.y = tw.translacao.first.z = 0;

    TiXmlElement * pontos;
    TiXmlElement * aux;
    pontos = element;

    vector<Ponto3D> vp = vector<Ponto3D>();
    aux = pontos->FirstChildElement("ponto");
    while (aux){
        Ponto3D paux;
        paux.x = atof(aux->Attribute("X"));
        paux.y = atof(aux->Attribute("Y"));
        paux.z = atof(aux->Attribute("Z"));
        vp.push_back(paux);

        aux = aux->NextSiblingElement("ponto");
    }

    tw.translacao.pontos = vp;
    transforms_atual.push_back(tw);
```

Figura 18: **uma translação**

.

8 Atualização da estrutura de dados

Por forma a suporta novas funcionalidades, nomeadamente as referidas na secção anterior alteramos as estruturas de dados existentes e criámos uma nova para criar o **Teapot**.

```
// Uma rotação 3D é uma extensão de um ponto 3D com o acréscimo de um ângulo
class Rotacao3D : public Ponto3D {
 public:
    Rotacao3D();
    float tempo; // Tempo para se complete a rotação (em segundos)
    float ang; // Ângulo
    float timebase;
    void printR() const; // debug
    void rotate();
};

class Translacao3D {
 public:
    Ponto3D first; // Valores para efetuar primeiro translate
    float tempo; // Tempo que leva a ser completada a translação (em segundos)
    float b; // Ponto da curva
    float timebase;
    vector<Ponto3D> pontos; // Pontos que definem a curva da translação
    void printT();
    void translate();
};
```

Figura 19: **uma translação**

.

Como podemos observar na figura 19, são acrescentadas algumas variáveis que guardam valores com o tempo para ambas as classes, no caso da classe translação existe também um vector de pontos para definir a trajetória da translação.

8.1 Problema de acessos a memória

Como são problemas característicos da linguagem C, o grupo teve dificuldade conseguir posteriormente aceder aos valores de tempo para atualizá-los o que não nos permitia efetuar as translações. Como solução provisória foram definidos arrays globais acedidos em **formas.cpp** por forma a conseguirmos guardar esses valores de tempo, atualizá-los e posteriormente reutilizá-los.

20 estas variáveis são inicializadas a partir do chamamento da função **init(int nformas)**.

```
// Variáveis globais para armazenamento de tempos e ângulos relativos a animações
int N; // número total de figuras a desenhar numa cena

float *global_b; // Array de tempos globais para cada figura
int global_index; // Índice da figura atual que se está a desenhar

float *global_timeb;
float *global_timer;
float *global_ang;
```

Figura 20: **variáveis globais**

.

```
void init(int nformas)
{
    N = nformas;

    global_b = (float*)malloc(N * sizeof(float));
    global_timeb = (float*)malloc(N * sizeof(float));
    global_ang = (float*)malloc(N * sizeof(float));
    global_timer = (float*)malloc(N * sizeof(float));

    global_index = 0;

    for (int i = 0; i < N; i++){
        global_b[i] = 0;
        global_timeb[i] = 0;
        global_ang[i] = 360; // Ângulo de translação de um planeta
        global_timer[i] = 0;
    }

    cout << "NUMERO DE PLANETAS: " << N << "\n";
}
```

Figura 21: **função init()**

.

9 Resultado final

Finalmente terminámos esta nossa 3ª etapa, e nesta secção apresentamos os resultados finais.

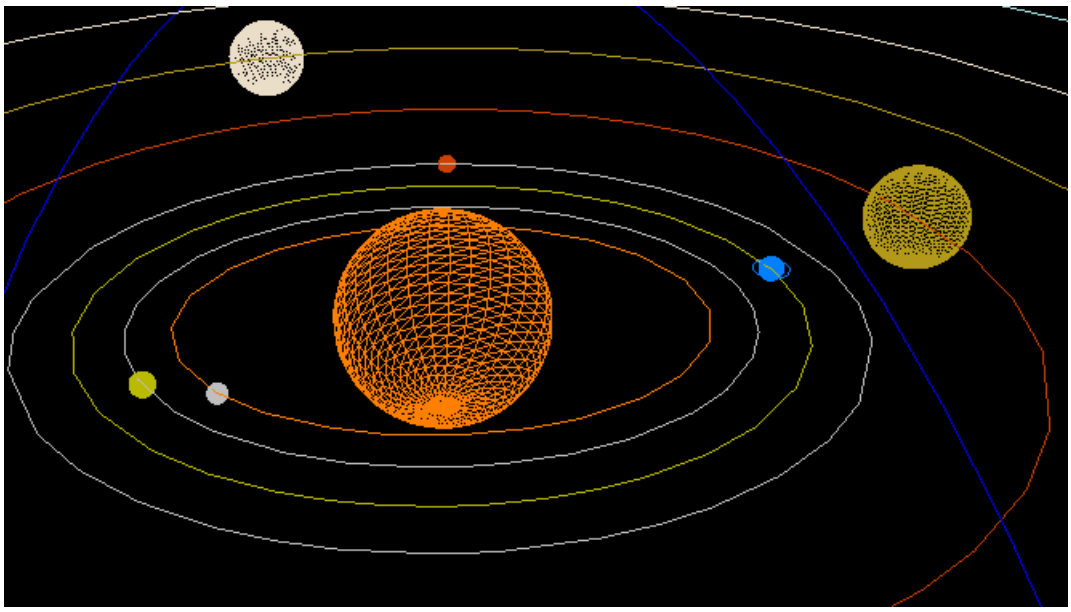

Figura 22: Exmplo 1

.

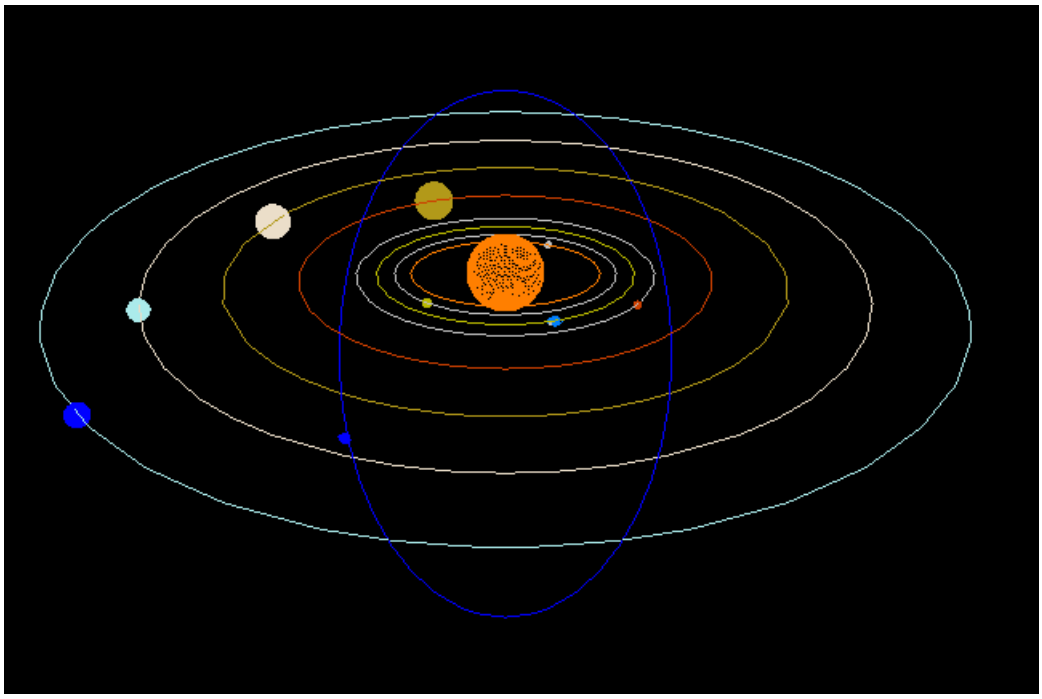

Figura 23: Exemplo 2

.

10 Conclusão

Nesta terceira fase consolidamos os conhecimentos obtidos nas aulas práticas relativamente ao manuseamento de VBOs, como ferramentas de optimização de desenho de gráficos. Consolidamos também o desenho de curvas e superfícies.

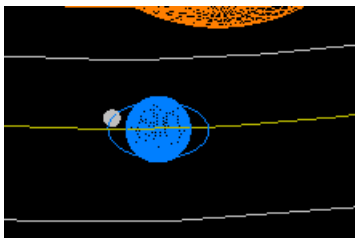

Figura 24: A Terra e a Lua

.