Universidade do Minho

Mestrado Integrado em Engenharia Informática

Computação Gráfica

2ª Fase - Transformações Geométricas

Grupo 36

Gonçalo Esteves
.(A85731).

João Araújo
.(A84306).

Mário Real
.(A72620).

Rui Oliveira
.(A83610).

4 de Maio de 2020

Conteúdo

Resumo

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 modificações ao trabalho já realizado, tais como o uso de curvas e superfícies de *Bézier*, curvas de Catmull-Rom e o uso de VBOs para o desenho das primitivas. Adicionou-se também um cometa ao Sistema Solar.

1 Introdução

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

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

2 Gerador

Nesta secção iremos explicar as alterações implementadas no nosso gerador, bem como as novas funcionalidades implementadas.

Novamente, este componente é responsável por gerar os pontos dos triângulos que permitem a construção das diversas figuras geométricas. No entanto, mais uma vez, sofreu alterações com a implementação desta fase, tais como a nova capacidade de realização de *parse* de ficheiros no formato *patch*.

2.1 *Patch*

Esta classe foi criada com o intuito de permitir a realização do *parse* dos ficheiros *patch*, por forma a que sejam armazenados os pontos de controlo.

- **Ficheiro de *Patch***

  Os *patches* de *Bézier* permitem ilustrar, em formato de texto, uma superfície, sendo a função responsável por tal a seguinte:

```
void Patch :: parsePatchFile(string filename);
```

O formato dos ficheiros *patch* é simples e caracterizado por especificações concretas:

- A primeira linha possui o número de *patches*;
- As restantes linhas contêm os 16 índices de cada um dos pontos de controlo que constituem os *patches* da figura;
- Em seguida, segue-se uma linha que contém o número de pontos de controlo necessários para gerar a figura;
- Por fim, aparecem os pontos de controlo. De notar que a ordem deles é importante, uma vez que cada ponto tem um índice associado que é utilizado.

- **Superfícies de *Bézier***

Por forma a definir uma superfíce de *Bézier* utilizamos 16 pontos de controlo que representamos numa matriz 4 por 4. Para isto, 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

$$P = \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_30 & P_{31} & P_{32} & P_{33} \end{bmatrix}$$

- Matrizes de Bézier

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

Tendo por base os valores de a e b, que se encontram entre 0 e 1, torna-se possível obter um ponto da superfície de *Bézier*, através da seguinte fórmula:

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

Deste modo, criamos a função *getPoint*, que permite obter as coordenadas de um ponto.

```
Point* Patch :: getPoint(float ta, float tb, float (*coordX)[4],
    float (*coordY)[4], float (*coordZ)[4])
```

As matrizes argumento referem-se às componentes x, y e z de cada um dos pontos de controlo do *patch* a ser tratado. Para começar, define-se a matriz de *Bézier* e os vetores a e b, multiplicando-se a matriz pelos vetores e guardando os resultados em matrizes auxiliares (am e bm). De seguida, multiplica-se o resultado obtido em am pelos valores das coordenadas de cada um dos pontos de controlo. Por fim, cada componente de um ponto é o produto da multiplicação do resultado obtido da operação anterior por bm.

```
float x = 0.0f, y = 0.0f, z = 0.0f;
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(*coordX,am,amCoordenadaX);
multMatrixVector(*coordY,am,amCoordenadaY);
multMatrixVector(*coordZ,am,amCoordenadaZ);

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

Para além disto, foi necessário elaborar a função *getPatchPoints* que é responsável por determinar quais os vértices dos triângulos. Deste modo, inicialmente são preenchidas três matrizes (coordenadasX, coordenadasY e coordenadasZ) com os pontos de controlo do *patch*, ou seja, cada matriz contém a coordenada x, y ou z do ponto, por forma a que, com o auxílio da função *getPoints* seja possível a geração dos vértices que constituem os triângulos. De realçar que o valor da tecelagem permite definir a porção com que se vai percorrer u e v. Para terminar, os vértices são guardados num vetor de pontos tendo em conta a regra da mão direita por forma a poderem ser posteriormente escritos no ficheiro .3d respetivo.

```
vector<Point> Patch::getPatchPoints(int patch) {
    vector<Point> points;
    vector<int> indexesControlPoints = patches.at(patch);

    float coordenadasX[4][4], coordenadasY[4][4], coordenadasZ
[4][4];
    float u,v,uu,vv;
    float t = 1.0f /(float)tesselation;
    int pos = 0;

    for (int i = 0; i < 4; i++)
    {
        for (int j = 0; j < 4; j++)
        {
            Point controlPoint = controlPoints[
indexesControlPoints[pos]];
            coordenadasX[i][j] = controlPoint.getX();
            coordenadasY[i][j] = controlPoint.getY();
            coordenadasZ[i][j] = controlPoint.getZ();
            pos++;
        }
    }

    for(int i = 0; i < tesselation; i++)
    {
        for (int j = 0; j < tesselation; j++)
        {
            u = (float)i*t;
            v = (float)j*t;
            uu = (float)(i+1)*t;
            vv = (float)(j+1)*t;
            Point *p0,*p1,*p2,*p3;

            p0 = getPoint(u, v, coordenadasX, coordenadasY,
coordenadasZ);
            p1 = getPoint(u, vv, coordenadasX, coordenadasY,
coordenadasZ);
```

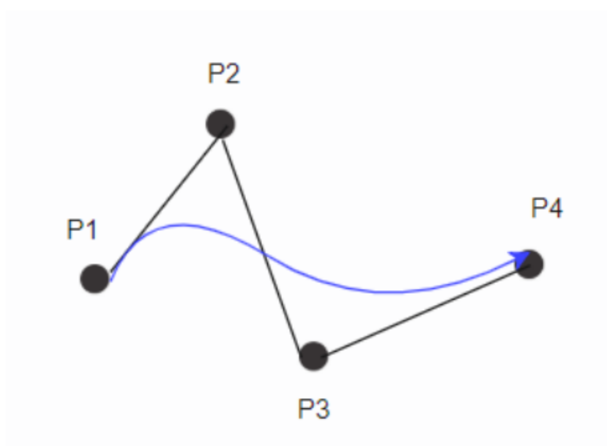
```
            p2 = getPoint(uu, v, coordenadasX, coordenadasY, coordenadasZ);
            p3 = getPoint(uu, vv, coordenadasX, coordenadasY, coordenadasZ);

            points.push_back(*p0); points.push_back(*p2); points.push_back(*p1);
            points.push_back(*p1); points.push_back(*p2); points.push_back(*p3);
        }
    }
    return points;
}
```

- **Curvas de *Bézier***

Para obter estas curvas, necessitamos de recorrer a quatro pontos de controlo (P1, P2, P3 e P4). Estes, utilizados numa função juntamente com um parâmetro t, que varia entre 0 e 1, tornam possível a criação de uma curva. De seguida, poderá observar-se a fórmula que define uma curva de *Bézier*, bem como um exemplo de uma curva.

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



Tendo em vista a obtenção de uma curva mais pormenorizada, deverá recorrer-se ao uso de um maior número de pontos, por forma a definir mais corretamente a curva.

3 Engine

Esta parte do projeto também sofreu alterações, tais como a implementação de VBOs e de curvas de *Catmull-Rom* e a realização de alterações com o intuito de permitir a movimentação dos planetas.

3.1 *Shape*

Uma das alterações introduzidas nesta fase foi a implementação de VBOs, por forma a permitir o desenho das primitivas. O *OpenGL* permite o uso de *Vertex Buffer Objects* por forma a inserir toda a informação dos vértices diretamente na placa gráfica. Assim, começamos por implementar os *vertex buffers*, sendo estes *arrays* capazes de conter os vértices das primitivas a desenhar, bem como uma variável que indica o número de vértices que contém o *buffer*.

```
int numVert;
GLuint buffer[1];
```

Criamos a função *prepareBuffer*, que gera e preenche o *buffer* com os pontos que lhe são passado, guardando, também, na memória da placa gráfica numVert*3 vértices.

```
void Shape::prepareBuffer(vector<Point *> p) {
    int i = 0;
    float* vs = new float[p.size() * 3];

    for(vector<Point*>::const_iterator v_it = p.begin(); v_it != p
    .end(); ++v_it){
        vs[i++] = (*v_it) -> getX();
        vs[i++] = (*v_it) -> getY();
        vs[i++] = (*v_it) -> getZ();
    }

    glGenBuffers(1, buffer);
    glBindBuffer(GL_ARRAY_BUFFER, buffer[0]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(float) * numVert * 3, vs,
   GL_STATIC_DRAW);

    delete [] vs;
}
```

Por forma a desenhar a informação guardada tivemos também de criar a função *draw*. Esta desenha todos os numVert*3 vértices guardados, tornando possível aumentar o desempenho com renderização imediata, sendo visível um aumento do número de *frames* por segundo, em comparação aos esperados na fase anterior do projeto.

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

3.2 *Transformation*

Relativamente à fase anterior, foram adicionadas novas variáveis a esta classe, associadas a transformações novas que foram criadas. Para além disto, adicionaram-se funções e variáveis necessárias à implementação das curvas de *Catmull-Rom*.

De modo a representar as referidas curvas, definimos a utilização da *spline* de Catmull-Rom. Para a definição da curva serão necessários pelo menos quatro pontos. Os pontos extremos juntamente com a tensão t permitem a definição de uma curva. Considerando a tensão 0, podemos obter a matriz M:

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

Definimos também os vetores T e T', sendo que t varia entre 0 e 1.

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

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

Desta feita, será possível obter um sistema solar dinâmico. Como tal, implementamos a função *getGlobalCatmullRomPoint* que permitirá a obtenção de coordenadas dos pontos e as suas derivadas. Esta função recorre à função auxiliar *getCatmullRomPoint*, que utiliza as matrizes e vetores referidos, por forma a gerar os valores de retorno.

```
void Transformation::getGlobalCatmullRomPoint(float gt, float *p,
    float *der) {
    int num = controlPoints.size();
    float t = gt * (float)num;
    int index = floor(t);
    t = t - (float)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, der);
}

void Transformation :: getCatmullRomPoint(float t, int *indexes,
    float *p, float *der) {
    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();
    }

    float a[4][4];
    multMatrixVector(*m, px, a[0]);
    multMatrixVector(*m, py, a[1]);
    multMatrixVector(*m, pz, a[2]);

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

    float Tdev[4] = { 3*T[1] , 2*T[2], 1 , 0};
    multMatrixVector(*a, Tdev, der);
}
```

Tendo em vista a criação das órbitas dos planetas, definimos a função *setCatmullPoints*. Esta cria os pontos da curva a partir dos pontos lidos no ficheiro *XML* e recorrendo à função *getGlobalCatmullRomPoint*, uma vez que esta nos permite obter as coordenadas do próximo ponto da curva para um dado valor t. Deste modo, recorremos a um ciclo que começa em 0 e acaba em 1, usando incrementos de 0.01, por forma a gerar 100 pontos da curva.

```
void Transformation::setCatmullPoints(){
    float ponto[4];
    float der[4];

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

3.3 *Parser*

- ***Rotate***

  Por forma a implementar o movimento de rotação dos planetas, que lhes permite rodarem sobre o seu próprio eixo, foi necessário adicionar a variável time à classe *Transformation*, que indica o tempo que uma figura demora a realizar uma rotação completa sobre o seu próprio eixo. Por forma a utilizar esta variável eficazmente, foi necessário também alterar a função *parseRotate* previamente definida, inserindo o excerto de código a seguir apresentado.

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

- ***Translate***

Foi também necessário capacitar os planetas de um movimento de translação em torno do Sol, usando para isso, também, a variável time. Neste caso, esta variável define o tempo necessário para que uma determinada figura ou grupo percorra a curva definida pelos seus pontos de controlo, contidos no nodo *translate*. Para aplicar este movimento, foi necessário alterar a função *parseTranslate*, adicionando o seguinte pedaço de código.

```
if(element -> Attribute("time")){
    bool d = false;
    vector<Point*> points;

    if(element -> Attribute("derivative"))
    d = stoi(element -> Attribute("derivative")) == 1;

    float time = stof(element -> Attribute("time"));
    time = 1 / (time * 1000);
    element = element -> FirstChildElement("point");

    while (element != nullptr){
        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"));

        points.push_back(new Point (x, y, z));
        element = element -> NextSiblingElement("point");
    }

    group -> addTransformation(new Transformation(time, points,
   d, "translateTime"));
}
```

3.4 *Engine*

Recorrendo ao uso de transformações geométricas, tais como rotações e translações, tornou-se possível dotar os planetas de movimento relativamente a outra figura ou a eles próprios. Desta feita, foram inseridas novas variáveis globais, por forma a capacitar o utilizador da capacidade de parar e retomar o movimento dos planetas:

- **stop** - flag que indica que os planetas deverão ou não estar em movimento;
- **eTime** - tempo decorrido desde que o movimento dos planetas está ativo;
- **cTime** - tempo decorrido desde que o programa foi iniciado.

Por forma a fornecer também informação relativa aos *frames* por segundo, criamos duas outras variáveis globais:

- **frame** - número de frames do último cálculo de *frames* por segundo;
- **timebase** - tempo em que ocorreu o último cálculo de *frames* por segundo.

De modo a obter o número de *frames* por segundo, é necessário calcular a quantidade de *frames* que passaram durante um segundo. Por forma a obter o intervalo de tempo que passou, foi necessário recorrer à função *glutGet(GLUT_ELAPSED_TIME)* que nos indica o tempo, em milissegundos, desde que a aplicação iniciou. Assim, a diferença entre o tempo presente e o tempo guardado na variável timebase permite determinar o tempo decorrido desde o último cálculo de *fps*. O número de *fps* será apresentado no título da janela, juntamente com o título. Para realizar tudo isto, criamos uma função nova, apresentada de seguida.

```
void fps(){
    int time;
    char name[30];

    frame++;
    time = glutGet(GLUT_ELAPSED_TIME);

    if(time - timebase > 1000){
        float f = frame * 1000/(time - timebase);
        timebase = time;
        frame = 0;
        sprintf(name, "Sistema Solar %.2f FPS", f);
        glutSetWindowTitle(name);
    }
}
```

- ***Translation***

Foi necessário definir a função *transApplication*, que irá aplicar as transformações descriminadas nos ficheiros XML, após a sua leitura. Um exemplo de uma das novas transformações, que fornece aos planetas um movimento de translação em torno de uma outra figura, será apresentado seguidamente.

```xml
<translate time="10" >
    <point X="25.0000000" Y="0.000000" Z="0.00000000" />
    <point X="23.0969890" Y="0.000000" Z="9.56708600" />
    <point X="17.6776700" Y="0.000000" Z="17.6776700" />
    <point X="9.56708600" Y="0.000000" Z="23.0969890" />
    <point X="0.00000000" Y="0.000000" Z="25.0000000" />
    <point X="-9.5670860" Y="0.000000" Z="23.0969890" />
    <point X="-17.677670" Y="0.000000" Z="17.6776700" />
    <point X="-23.096989" Y="0.000000" Z="9.56708600" />
    <point X="-25.000000" Y="0.000000" Z="0.00000000" />
    <point X="-23.096989" Y="0.000000" Z="-9.5670860" />
    <point X="-17.677670" Y="0.000000" Z="-17.677670" />
    <point X="-9.5670860" Y="0.000000" Z="-23.096989" />
    <point X="-0.0000000" Y="0.000000" Z="-25.000000" />
    <point X="9.56708600" Y="0.000000" Z="-23.096989" />
    <point X="17.6776700" Y="0.000000" Z="-17.677670" />
    <point X="23.0969890" Y="0.000000" Z="-9.5670860" />
</translate>
```

Como podemos observar, o tempo de translação é passado como parâmetro no ficheiro XML. O objetivo consiste em que, com o decorrer do tempo, o planeta se desloque ao longo da sua curva, aplicando o movimento de translação desejado. Para tal ser possível, recorremos à variável eTime, que guarda o tempo decorrido enquanto que o sistema se encontrava em movimento, e ainda à função *getGlobalCatmullRomPoint*. Nesta fase, optamos por modificar também a forma como as órbitas são criadas, recorrendo agora aos pontos gerados pela fórmula de *Catmull-Rom*. Para além de tudo isto, a função *transApplication* terá também uma condição que será utilizada para definir a trajetória do cometa. Este deverá não só mover-se segundo a sua trajetória, como também manter se na orientação da curva. Deste modo, no caso do cometa e de outras figuras que se pretenda que o movimento seja descrito desta forma, o ficheiro XML deverá conter a informação disposta da seguinte forma:

```
<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>
```

Como é visível neste excerto, agora aparece um parâmetro novo, que é a derivada do ponto da curva. Por forma a que o cometa esteja orientado, é necessário criar uma matriz que irá ser utilizada na função *glMultMatrixf*, com o objetivo de manter o cometa alinhado com a curva.

Posto tudo isto, é possível apresentar o excerto de código da função *transApplication* responsável pelos movimentos de translação.

```
else if (strcmp(type,"translateTime") == 0) {
    float p[4], d[4];
    float dTime = eTime *time;

    t -> getGlobalCatmullRomPoint(dTime, p, d);
    drawOrbits(t);
    glTranslatef(p[0], p[1], p[2]);

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

        glMultMatrixf(matrix);
    }
}
```

- ***Rotation***

  A função *transApplication* também processa os movimentos de rotação de uma figura sobre o seu próprio eixo, que nós designamos no programa como sendo *rotateTime*, após a sua leitura do ficheiro XML. Um exemplo de texto do ficheiro XML será visível de seguida.

```
<rotate time="10" X="0" Y="1" Z="2" />
```

  Novamente, necessitamos de recorrer à variável eTime, por forma a determinar o tempo já decorrido durante a movimentação dos planetas. Recorrendo à multiplicação do valor desta com o valor do ângulo de rotação, iremos obter diferentes ângulos à medida que o tempo passa. Desta feita, torna-se possível dotar a figura de um movimento de rotação sobre o seu próprio eixo.

  Tendo tudo isto em causa, definimos o seguinte excerto de código para o processamento das transformações de movimento de rotação.

```
else if (strcmp(type,"rotateTime") == 0) {
    float aux = eTime * angle;
    glRotatef(aux, x, y, z);
}
```

- **Movimento dos planetas**

  Tal como já foi referido previamente, por forma a dotar o sistema de movimento são utilizados os diferentes tipos de *rotate* e *translate* descritos previamente. Esta movimentação tem por base o decorrer do tempo de vida da aplicação, sendo este o impulsionador do movimento de todas as figuras. A existência ou ausência de movimentação, tal como fora referido, é controlada por uma variável global, que tomará o valor 0 caso o movimento esteja ativo, ou o valor um, caso o movimento esteja inativo. Desta feita, o aumento do tempo de movimento das figuras apenas ocorrerá quando a variável stop estiver com o seu valor a 0.

4 Demonstração

Por forma a gerar os resultados pretendidos, torna-se necessário executar os seguintes comandos, a partir da pasta principal:

- **Gerador**

```
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: Comandos para gerar os ficheiros .3d necessários

O projeto inclui uma pasta designada por *Files3d* onde serão inseridos todos os ficheiros .3d criados, para posteriormente serem lidos pelo *engine*. Nesta fase passa-se a criar também um ficheiro com os pontos para um *teapot*, que será utilizado no cometa, usando por base o ficheiro *patch* fornecido.

- ***Engine***

```
mkdir build && cd build
cmake ..
make
./engine solarsystem.xml
```

Figura 2: Comandos para gerar o Sistema Solar

Passando como parâmetro o nome do ficheiro XML que se pretende ler (este deverá estar guardado na pasta *XML's* incluída no projeto), o *engine* irá ler esse ficheiro, de modo a gerar as figuras pretendidas.

4.1 Menus

Mantivemos o menu criado nas fases anteriores, sem implementar novas atualizações. Este menu aparece quando o utilizador, ao executar o *engine*, passa como argumento *-h* em vez do nome de um ficheiro XML.

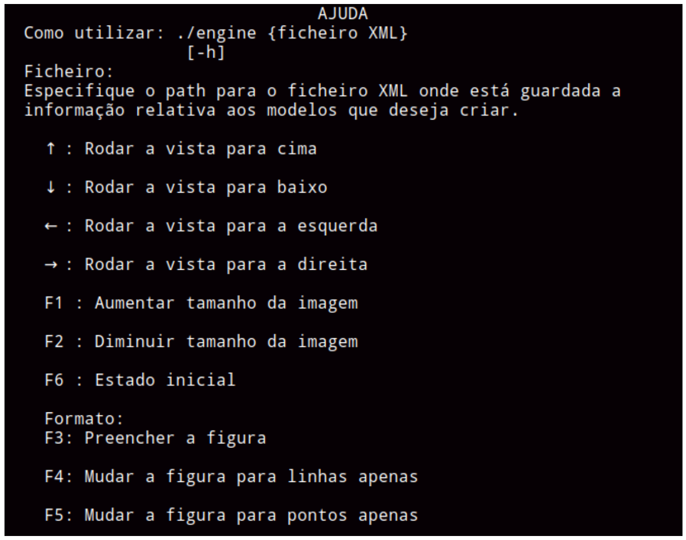


Figura 3: Menu Principal

No entanto, foi necessário atualizar o menu próprio do Sistema Solar, uma vez que removemos a capacidade de focar em cada planeta, permitindo agora ao utilizador pausar ou não o movimento dos planetas. Este menu aparece quando pressionamos o botão direito do rato. Os parâmetros de "Movimento"foram criados de forma *responsive*.

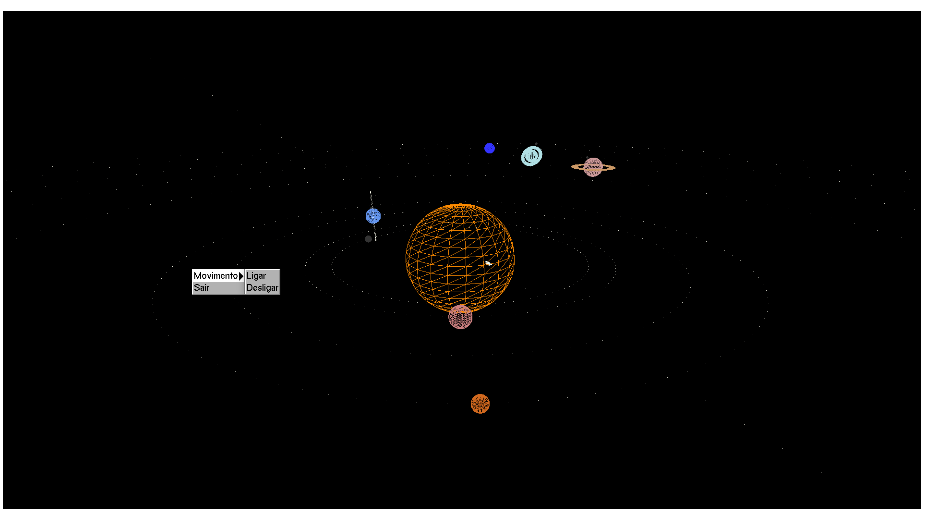


Figura 4: Menu dos Planetas

4.2 Sistema Solar

Tendo em vista a construção do Sistema Solar, tivemos em conta os planetas principais que o constituem (Mercúrio, Vénus, Terra, Marte, Júpiter, Saturno, Urano e Neptuno), a sua disposição, forma e cor. Optamos por não fazer o Sistema Solar à escala, isto é, não respeitamos totalmente as dimensões dos astros nem as distâncias que os separam, uma vez que, caso o fizessemos, ficaríamos com um cenário muito disperso e de difícil observação, acreditando nós não ser este o objetivo do projeto.

No entanto, tivemos em consideração a adição de alguns extras, como por exemplo alguns satélites naturais (a Lua e outros satélites de Júpiter e Saturno) bem como os anéis de Saturno e Urano (estes criados a partir do *torus* gerado). Acrescentamos também o planeta anão Plutão. Nesta fase, optamos por adicionar também um comenta, construído a partir do *teapot* processado do ficheiro *patch* fornecido.

Tendo tudo isto em vista, formulamos um ficheiro XML que permita a construção do Sistema Solar, tendo em atenção tudo o que fora referido previamente. Deste modo, podemos obter o resultado seguidamente apresentado.

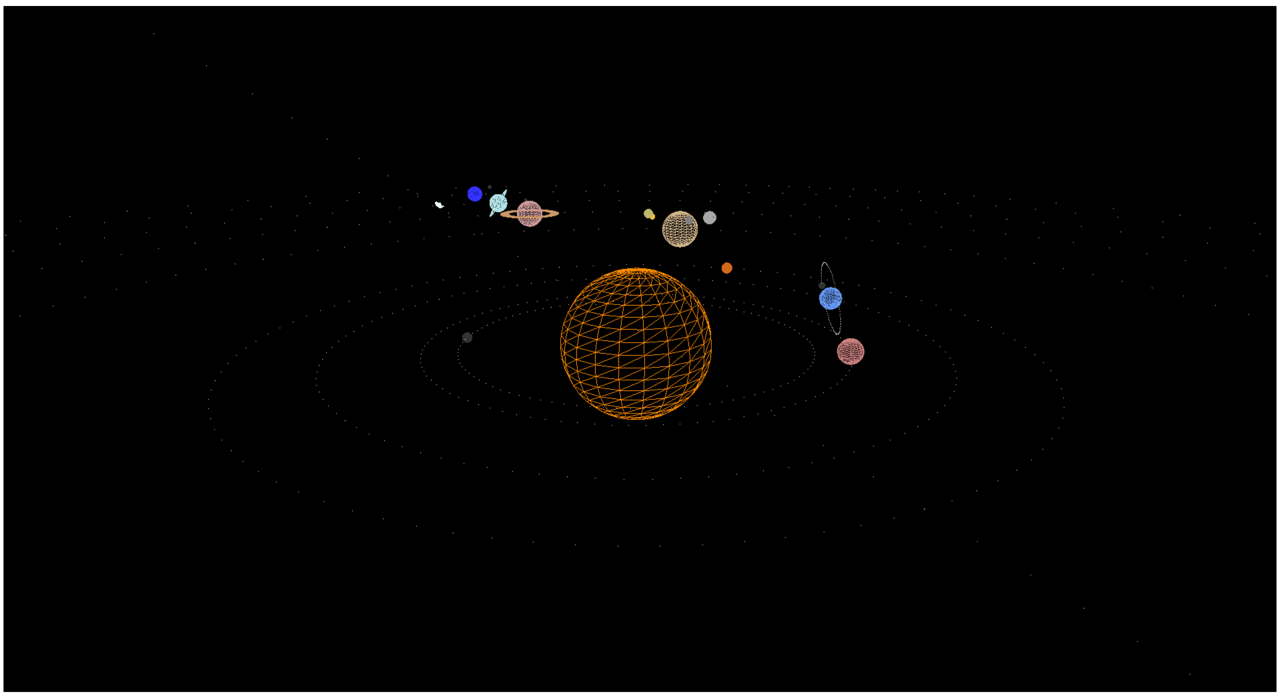


Figura 5: Sistema Solar

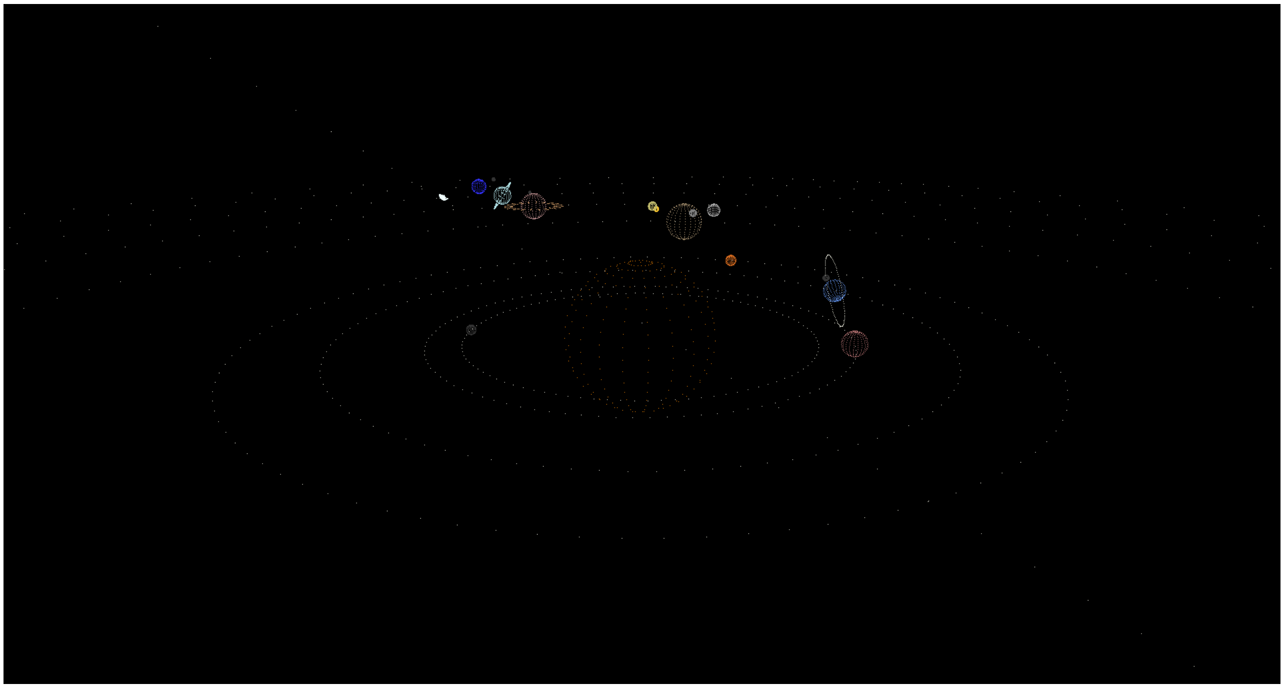


Figura 6: Sistema Solar com pontos

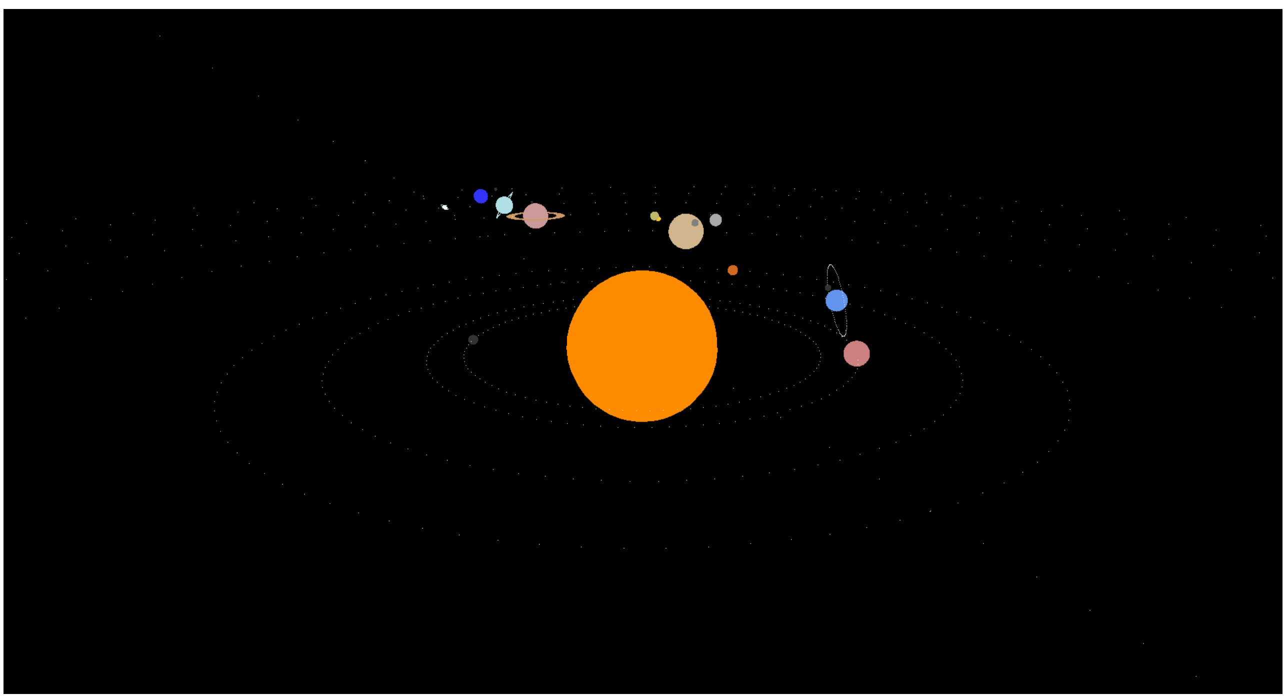


Figura 7: Sistema Solar preenchido

5 Conclusão

Para esta terceira fase podemos afirmar que cumprimos com os objetivos estabelecidos, uma vez que adicionamos as funcionalidades pretendidas, isto é, o uso de curvas e superfícies de Bézier, curvas de Catmull-Rom e o uso de VBOs para o desenho das primitivas. Para além disto, também atualizamos o ficheiro XML responsável por suportar as informações do Sistema Solar, acrescentando as novas transformações agora processáveis, bem como o *parser* que torna capaz o processamento do mesmo.