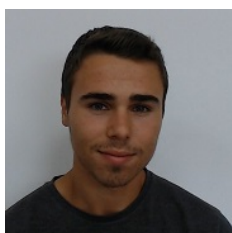


Computação Gráfica

Trabalho Prático (Parte 3)

Luís Miguel Ramos (A83930)
Válter Carvalho (A84464)

4 de Maio de 2020



Grupo nº 26
Mestrado Integrado em Engenharia Informática
Universidade do Minho

Conteúdo

1	Introdução	2
2	Motor	3
2.1	Catmull-Rom	3
2.2	VBO's	4
2.2.1	Model	6
2.2.2	CatmullRom (Trajetória Gráfica)	7
2.3	Alterações nas Transformações Geométricas	8
2.3.1	Rotação	8
2.3.2	Translação	8
2.4	Alterações no XMLReader	9
3	Gerador	11
3.1	Bezier	11
3.1.1	Leitura do ficheiro patch	11
3.1.2	Fórmula de Bezier	14
3.2	Execução	15
4	Resultado	16
4.1	Teapot (Bezier Patches)	16
4.2	Sistema Solar	16
5	Extras	17
5.1	Trace	17
5.2	Atalhos	18
6	Conclusão	22

1 Introdução

Concluída a segunda fase deste projeto, demos inicio à terceira fase que incide mais sobre a animação dos modelos, curvas cúbicas e desempenho

Os modelos agora têm a possibilidade de ter translações e rotações animadas, fazendo uso de curvas de Catmull-Rom para o primeiros.

Para um maior eficiência devido à quantidade enorme de triângulos que se vão desenhar a cada iteração, são agora utilizados VBO's.

Também nesta fase, é agora permitido desenhar um novo tipo de modelo utilizando Bezier Patches.

2 Motor

2.1 Catmull-Rom

Para podermos efetuar uma translação dinâmica, tivemos de re-implementar algo que foi trabalhado nas aulas práticas, que são as curvas de Catmull-Rom. Para isso, definimos uma classe que abstrai todo este comportamento.

```
1 class CatmullRom {
2     int points; // total de pontos da trajetoria
3     vector<Point> segPoints; // pontos da trajetoria
4
5     float gt; // tempo entre 0 e 1 para calcular a posicao num
        segmento
6     float animationTime; // tempo total de animacao
7
8     float pos[3]; // vetor que da a posicao nos eixos x,y,z
9     float deriv[3]; // vetor que indica derivada no ponto
        calculado
10    float z[3]; // vetor que indica as coordenadas do eixo z
11    float up[3] = { 0,1,0 }; // vetor para indicar o sentido
        que para "cima" por defeito
12
13    int segments; // numero total de segmentos
14    float* rgb; // cor da trajetoria (caso seja o caso de ser
        desenhada)
15    GLuint vboID; // indice no buffer do VBO
16    GLuint* buffer; // referencia para o vetor de VBO
```

Há duas funções fundamentais para o comportamento desta transformação, que são as que efetivamente calculam os pontos em que o objeto se encontra em todos os *frames*. Estas funções são:

- *void getGlobalCatmullRomPoint(float gt, float* pos, float* deriv)*: utiliza 4 pontos da trajetória consoante o segmento que se encontra (dado por **gt**), guardando no vetor **pos** o ponto calculado e em **deriv** a sua derivada.
- *void getCatmullRomPoint(float t, Point p0, Point p1, Point p2, Point p3, float* pos, float* deriv)*: função auxiliar da anterior que utiliza os 4 pontos da trajetória calculados na anterior para efetivamente proceder ao cálculo (em determinado instante) da posição e da derivada do objeto.

Agora é, portanto, realizada a translação animada consoante a posição calculada (e respetiva rotação usando os vetores normais que são calculados) através da função *animatedTranslate*:

```

1 void animatedTranslate(float elapsed_time) {
2     float m[4][4], m_transpose[4][4];
3     this->gt = elapsed_time / this->animationTime; // elapsed
        time
4     this->gt -= floor(gt); // has to be between 0 and 1
5
6     getGlobalCatmullRomPoint(gt, pos, deriv);
7
8     normalize(deriv); // processos de normalizacao e calculo
        vetorial
9     cross(deriv, up, z);
10    normalize(z);
11    cross(z, deriv, up);
12    normalize(up);
13    glTranslatef(pos[0], pos[1], pos[2]); // translacao para
        a posicao correta
14    buildRotMatrix(deriv, up, z, *m); // construir a matriz
        de rotacao
15    transpose(*m, *m_transpose); // calcular a transposta
16    glMultMatrixf(*m_transpose); // multiplicar matriz atual
        (viewmodel) com as transformacoes finais
17 }

```

2.2 VBO's

Os VBO são uma ótima medida de garantirmos uma elevada performance no nosso programa tirando partido duma componente já especializada em efetuar estas operações custosas muito rapidamente, que é a placa gráfica. O CPU demora muito tempo a processar todos os pontos necessários para desenhar uma figura e, escalando isto para um sistema solar, temos possivelmente milhões de triângulos a serem desenhados em cada frame, o que se torna custoso quanto mais complexo for.

Para criarmos o VBO, a nossa implementação foi descobrir quantos **model** existem no ficheiro XML para identificarmos com um único **vboID** e, assim, conseguirmos criar um vetor que guarde os índices relativos a cada um desses **vboID**. Todo este processo é gerido no leitor de XML e registados no **main.cpp**.

```

1 void getTransformations(string f) {

```

```

2   GLuint nFig = 0;
3   transformations = new vector<Transformations*>();
4
5   xmlReader(f, transformations, &nFig); // nFig da nos o
        numero de models dentro do ficheiro XML
6
7   figures = new GLuint[nFig](); // alocao de espacos para
        os indices relativos a cada vboID
8
9   glGenBuffers(nFig, figures); // criacao do VBO
10
11  for (Transformations* t : *transformations) {
12      t->addReferenceBuffer(figures); // adicionar a
        referencia para o VBO para posteriormente desenhar a
        respetiva fatia de cada transformacao
13      t->start(); // inicializar a "fatia" respetiva do VBO
14  }
15
16  cout << "Scene:␣" << f << "␣loaded!" << endl;
17  }

```

A função *start()* dentro da classe *Transformations* inicializa o processo de criação dos buffers respetivas do VBO associados a cada **vboID** de cada model pertencente ao ficheiro de XML.

```

1  void start() {
2      for (Model* m : this->models) {
3          m->prepareModel(this->buffer); // mover os pontos do
        modelo para a placa grafica
4      }
5
6      if(this->translate) this->translate->prepareTranslate(
        this->buffer); // caso seja animada precisa do VBO
        para as suas trajetorias graficas (ver abaixo na
        seccao das translacoes)
7
8      for (Transformations* t : this->subgroups) {
9          t->start(); // recursivamente aplicar aos subgrupos
10     }
11 }

```

2.2.1 Model

Esta classe foi refeita para ter em conta os VBO, isto é, apenas guarda os pontos para serem transferidos posteriormente para a placa gráfica.

```
1  class Model {
2  vector<Point> points; // pontos retirados do ficheiro .3d
3  GLuint vboID; // indice do vetor com indices do VBO
4  (...)

1  class Model {
2  public:
3  (...)
4  void prepareModel(GLuint* buffer) {
5      int size = this->points.size(); // total de pontos a
        guardar na placa grafica
6      float* p = new float[size * 3]; // desdobrar Point em 3
        floats
7      int i = 0;
8      for (int j = 0; j < size; j++) {
9          p[i] = this->points.at(j).x;
10         p[i + 1] = this->points.at(j).y; // processo de
            desdobramento
11         p[i + 2] = this->points.at(j).z;
12         i += 3;
13     }
14     glBindBuffer(GL_ARRAY_BUFFER, buffer[vboID]); // criar o
        buffer de VBO na placa grafica
15     glBufferData(GL_ARRAY_BUFFER, sizeof(float) * size * 3, p
        , GL_STATIC_DRAW); // colocar os pontos nesse buffer
16 }

17
18 void drawModel(float red, float green, float blue, GLuint*
    buffer) {
19     glColor3f(red, green, blue); // cor escolhida para o
        modelo (visto que nao ha texturas para ja)
20     glBindBuffer(GL_ARRAY_BUFFER, buffer[vboID]); // criar
        apontador para o buffer
21     glVertexPointer(3, GL_FLOAT, 0, 0); // quantos vertices e
        que tipo (neste caso floats) sao
22     glDrawArrays(GL_TRIANGLES, 0, points.size()); // desenhar
        triangulos ate esgotar os pontos
23 }
24 };
```

2.2.2 CatmullRom (Trajetória Gráfica)

Muito semelhante aos modelos anteriores, a diferença é que os pontos são retirados usando as primitivas da curva de Catmull-Rom em função do número de segmentos na curva e agora são desenhados usando um line-loop, tendo em conta o seu número de segmentos.

```
1 void prepareCurve() {
2     float* loopPoints = new float[segments * 3];
3     int j = 0;
4     float t;
5     for (int i = 0; i < this->segments; i++) {
6         t = i / (float) segments;
7         getGlobalCatmullRomPoint(t, pos, deriv);
8         loopPoints[j] = pos[0];
9         loopPoints[j + 1] = pos[1];
10        loopPoints[j + 2] = pos[2];
11        j += 3;
12    }
13
14    glBindBuffer(GL_ARRAY_BUFFER, buffer[vboID]); // // criar
        // o buffer de VBO na placa grafica
15    glBufferData(GL_ARRAY_BUFFER, sizeof(float) * segments *
        3, loopPoints, GL_STATIC_DRAW); // colocar os pontos
        // nesse buffer
16 }
17
18 void traceCurve() {
19     glPushMatrix(); // nao alterar as outras transformacoes
20     glColor3f(this->rgb[0], this->rgb[1], this->rgb[2]); //
        // cor escolhida para a curva
21     glBindBuffer(GL_ARRAY_BUFFER, buffer[vboID]); // criar
        // apontador para o buffer
22     glVertexAttribPointer(3, GL_FLOAT, 0, 0); // quantos vertices e
        // que tipo (neste caso floats) sao
23     glDrawArrays(GL_LINE_LOOP, 0, segments); // agora em vez
        // de triangulos e um line-loop
24     glPopMatrix(); // voltar a matriz de transformacoes
        // antiga
25 }
26 };
```


2.3 Alterações nas Transformações Geométricas

2.3.1 Rotação

A rotação teve umas ligeiras alterações, pois é agora pretendido dar ao utilizador a oportunidade de escolher que pretende uma rotação que seja dinâmica ou estática.

```
1 class Rotate {
2 public:
3     bool animated; // se e ou nao dinamica
4     float time; // tempo para rodar 360 graus sobre determinado
        eixo
5     float ang, x, y, z; // angulo de rotacao (se for estatica)
        e o eixo de rotacao
6     (...)
```

Efetivamente a única mudança a efetuar no método *apply()* que tínhamos anteriormente para esta classe, é pô-lo a ter em conta esta possibilidade ser estática ou dinâmica, ou seja, se contabilizamos ou não o ângulo atual consoante o tempo ou apenas o ângulo definido estaticamente.

```
1     void apply(float elapsed_time) {
2         if (this->animated) { // se for animada, calcular o
            angulo de acordo com o tempo que passou
3             this->ang = (elapsed_time / this->time) * 360; //
                angulo atual de rotacao
4         }
5         glRotatef(this->ang, this->x, this->y, this->z); // faz
            rotacao consoante seja ou nao animada
6     }
```

2.3.2 Translação

Quanto às translações, receberam também várias alterações para poderem ser dinâmicas (utilizando para isso curvas de Catmull-Rom) ou estáticas.

```
1 class Translate {
2 public:
3     bool animated; // se e dinamica ou nao
4     bool traced; // se o utilizador pretende mostrar
        graficamente a trajetoria
5     bool toggledTrace; // para ativar/desativar a trajetoria
6     float x, y, z; // caso seja uma translacao estatica usam-se
        estes pontos
```

```

7  CatmullRom* cr; // caso seja dinamica o comportamento esta
   definido dentro deste objeto
8  (...)

```

Agora, é necessário preparar os VBO's relativos às trajetórias animadas (caso seja o caso) para posterior desenho no ecrã.

```

1  void prepareTranslate(GLuint* b) { // recebe o vetor de
   indices do VBO para saber o seu
2  if (this->traced) { // se for uma trajetoria grafica
3      this->cr->addReferenceBuffer(b); // adicionar o vetor
   como referencia para mais tarde
4      this->cr->prepareCurve(); // preparar o VBO dentro do
   objeto de Catmull-Rom
5  }
6  }

```

Por fim é necessário o método que efetivamente aplica esta translação.

```

1  void apply(float elapsed_time) {
2      if (!this->animated) { // se nao for dinamica
3          glTranslatef(this->x, this->y, this->z); // translacao
   usando os pontos estaticos
4      }
5      else if (this->cr->isValid()) { // se for uma
   transformacao dinamica valida (pontos de controlo >=
   4)
6          if (this->traced && this->toggledTrace) { // se for uma
   trajetoria grafica e estiver ativada
7              cr->traceCurve(); // desenhar essa trajetoria
8          }
9          cr->animatedTranslate(elapsed_time); // fazer o
   translate dinamico pela curva de Catmull-Rom
10     }
11     else {
12         cout << "ERROR. Minimum of 4 points required. Aborting
   this transformation..." << endl;
13     }
14 }

```

2.4 Alterações no XMLReader

Essencialmente foram adicionadas funções que tratam individualmente agora as rotações e as translações, *parseRotate* e *parseTranslate*. Devido à verbosidade do código específico do *tinyxml2*, será adicionada aqui no relatório em pseudo-código.

```

1 void parseRotate(XMLElement* rotate, Transformations*
  transforms) {
2   x,y,z = Atributo x,y,z do elemento XML;
3   se existir atributo "time":
4     new Rotate(x,y,z,valor desse atributo,true);
5   se nao
6     new Rotate(valor do atributo "angle", x, y, z);
7   }
8   transforms->addRotate(r);
9 }
10
11 void parseTranslate(XMLElement* translate, Transformations*
  transforms, unsigned int* nFig) {
12   se tiver o atributo "time":
13     float time = valor desse atributo;
14     int seg;
15     bool traced = false;
16     se tiver o atributo "traced":
17       seg = o valor desse atributo;
18       traced = true;
19     se nao:
20       seg = 0;
21     vector<Point> controlPoints;
22     para todos os pontos dentro do elemento translate:
23       inseri-los em controlPoints;
24   }
25   se (traced):
26     float* rgb = new float[3]();
27     se tiver os atributos "r", "g" e "b":
28       rgb[0] = valor do atributo "r" / 255.0f;
29       rgb[1] = valor do atributo "g" / 255.0f;
30       rgb[2] = valor do atributo "b" / 255.0f;
31     se nao:
32       rgb[2] = rgb[1] = rgb[0] = 1.0f;
33     transforms->addTranslate(new Translate(
      controlPoints, time, seg, *nFig, rgb));
34     (*nFig)++;
35     se nao:
36       transforms->addTranslate(new Translate(controlPoints,
        time, seg));
37   se nao:
38     transforms->addTranslate(new Translate(valor do
      atributo "x", valor do atributo "y", valor do
      atributo "y"));
39 }

```

3 Gerador

3.1 Bezier

3.1.1 Leitura do ficheiro patch

Os ficheiros patches seguem sempre o seguinte padrão. A primeira linha indica o número de patches (**numPatches**) (Passo 1). As restantes **numPatches** linhas correspondem a índices de pontos de controlo. Cada linha contém 16 índices, e cada um representa um ponto de controlo desse patch (Passo 2). A linha seguinte, que será $1 + \text{numPatches} + 1$, irá conter o número de pontos de controlo (**numCP**) (Passo 3). As seguintes **numCP** linhas correspondem às coordenadas de cada ponto de controlo. Cada linha contém 3 valores, que correspondem às coordenadas x,y e z (Passo 4).

Percebendo então o funcionamento de um ficheiro patch, passou-se à fase de implementação. Começou-se pela criação de estrutura de dados de modo a guardar toda a informação do ficheiro.

```
1     int numPatch = 0, numCP = 0, posicao = 0;
2     string linha;
3     int** indicesCP;
4     float** valuesCP;
```

De seguida fez-se a implementação dos 4 passos necessários à leitura.

```
1     //Passo 1
2     // primeira linha contem o numero de patches
3     getline(readFile, linha);
4     numPatch = stoi(linha);

1     //Passo 2
2     //as seguintes 'numPatch' linhas contem informa o dos
3     //control points
4     //cada linha tera 16 indices
5     //para guardar a info usamos uma matriz -> indicesCP[
6     //numPatch][16]
7     indicesCP = new int* [numPatch];
8     for (int i = 0; i < numPatch; i++) {
9         indicesCP[i] = new int[16];
10        getline(readFile, linha);
11        char* arrayLinha = strdup(linha.c_str());
12        char* point = strtok(arrayLinha, ",");
```

```

11     for (int j = 0; point != NULL; j++) {
12         indicesCP[i][j] = stoi(point);
13         point = strtok(NULL, ",");
14     }
15 }

1 //Passo 3
2 // esta linha contem o numero de control points
3 getline(readFile, linha);
4 numCP = stoi(linha);

1 //Passo 4
2 //as seguintes 'numCP' linhas cont m as coordenadas dos
   control points
3 //cada linha ter 3 coordenadas
4 //para guardar a info usamos uma matriz -> valuesCP[numCP
   ][3]
5 valuesCP = new float* [numCP];
6 for (int i = 0; i < numCP; i++) {
7     valuesCP[i] = new float[3];
8     getline(readFile, linha);
9     char* arrayLinha = strdup(linha.c_str());
10    char* point = strtok(arrayLinha, ",");
11    for (int j = 0; point != NULL; j++) {
12        valuesCP[i][j] = stof(point);
13        point = strtok(NULL, ",");
14    }
15 }

```

Por fim, o objetivo foi transformar essa informação em pontos e guardar esses pontos num ficheiro com o formato "teapot.3d" sendo assim possível a construção do teapot. Para isto recorremos à fórmula de Bezier.

```

1 if (writeFile.is_open()) {
2
3     float incremento = 1.0 / tessellation;
4     float coord0[3], coord1[3], coord2[3], coord3[3];
5
6     /*pontos    0 . ---- . 3
7                |         |
8                |         |
9                1 . ---- . 2
10    */
11
12    // v-> horizontal / u-> vertical

```

```

13
14     for (int i = 0; i < numPatch; i++) {
15         for (float u = 0; u < 1; u += incremento) {
16             for (float v = 0; v < 1; v += incremento) {
17                 float u1 = u + incremento;
18                 float v1 = v + incremento;
19
20                 bezierPatches(u, v, indicesCP[i], valuesCP,
21                               coord0);
22                 bezierPatches(u, v1, indicesCP[i], valuesCP,
23                               coord1);
24                 bezierPatches(u1, v1, indicesCP[i], valuesCP,
25                               coord2);
26                 bezierPatches(u1, v, indicesCP[i], valuesCP,
27                               coord3);
28
29                 writeFile << coord0[0] << " " << coord0[1] << " "
30                               << coord0[2] << endl;
31                 writeFile << coord1[0] << " " << coord1[1] << " "
32                               << coord1[2] << endl;
33                 writeFile << coord2[0] << " " << coord2[1] << " "
34                               << coord2[2] << endl;
35                 writeFile << coord3[0] << " " << coord3[1] << " "
36                               << coord3[2] << endl;
37             }
38         }
39     }

```

3.1.2 Fórmula de Bezier

A fórmula de Bezier é composta por vários componentes, como é visível na figura abaixo.

$$B(u,v) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} M \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} M^T \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix}$$

Calculamos então todas essas componentes.

```
1 float U[1][4] = { { pow(u,3) , pow(u,2) ,u , 1 } };
2
3 float M[4][4] = { {-1.0f, 3.0f, -3.0f, 1.0f},
4                  { 3.0f, -6.0f, 3.0f, 0.0f},
5                  {-3.0f, 3.0f, 0.0f, 0.0f},
6                  { 1.0f, 0.0f, 0.0f, 0.0f} };
7
8 float MT[4][4];
9 transpose(*M, *MT);
10
11 float V[4][1] = { {pow(v,3)}, {pow(v,2)}, {v}, {1} };
12
13 for (int i = 0; i < 4; i++) {
14     for (int j = 0; j < 4; j++) {
15         cpX[i][j] = valuesCP[indicesCP[i * 4 + j]][0];
16         cpY[i][j] = valuesCP[indicesCP[i * 4 + j]][1];
17         cpZ[i][j] = valuesCP[indicesCP[i * 4 + j]][2];
18     }
19 }
```

Por fim, fazemos as multiplicações entre as matrizes.

```
1 // aXc * cXb
2 //1X4 * 4X4 a=1;c=4;b=4
3 multMatrix(*U, *M, *res, 1, 4, 4);
4
5 //1X4 * 4X4 a=1;c=4;b=4
6 multMatrix(*res, *cpX, *resX, 1, 4, 4);
7 multMatrix(*res, *cpY, *resY, 1, 4, 4);
8 multMatrix(*res, *cpZ, *resZ, 1, 4, 4);
9
10 //1X4 * 4X4 a=1;c=4;b=4
```

```

11  multMatrix(*resX, *MT, *resTX, 1, 4, 4);
12  multMatrix(*resY, *MT, *resTY, 1, 4, 4);
13  multMatrix(*resZ, *MT, *resTZ, 1, 4, 4);
14
15  //1X4 * 4X1  a=1;c=4;b=1
16  multMatrix(*resTX, *V, x, 1, 1, 4);
17  multMatrix(*resTY, *V, y, 1, 1, 4);
18  multMatrix(*resTZ, *V, z, 1, 1, 4);
19
20  coord[0] = x[0]; coord[1] = y[0]; coord[2] = z[0];

```

3.2 Execução

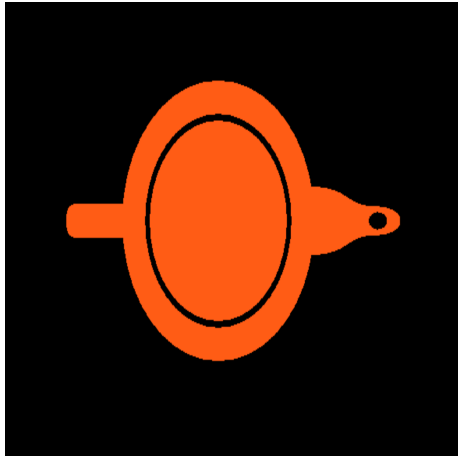
Foi agora adicionado um novo comando ao gerador:

- **generator bezier <InputFile>.patch <outputFile>.3d <tessellation>**

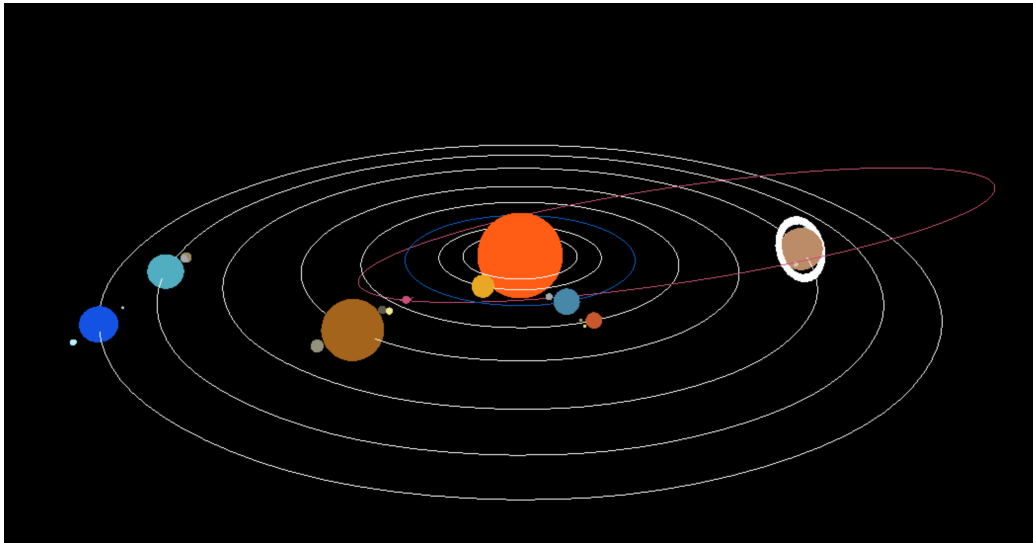
Que permite, agora, a criação de uma figura utilizando patches de Bezier. Os parâmetros são o ficheiro de *input* (em formato patch tal como o fornecido), o ficheiro de *output*, que será em formato 3d tal como os que temos trabalhado até agora e a tesselação, que indica a "qualidade" do desenho. Quanto maior o nível de tesselação, melhores os resultados graficamente.

4 Resultado

4.1 Teapot (Bezier Patches)



4.2 Sistema Solar



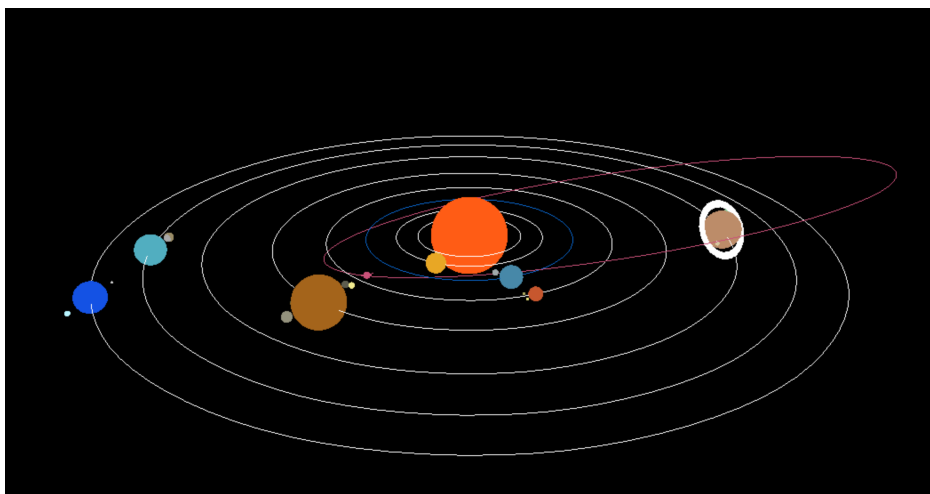
5 Extras

5.1 Trace

Ao nosso projeto foi adicionado um extra denominado de **trace**, especificado no ficheiro de XML. Este serve para demonstrar o movimento efetuado por cada planeta e é opcional, porém terá de ter uma cor associada. O valor deste atributo indica o número de segmentos da linha de trajetória.

Por exemplo, `<translate time="1" trace="100" r="255" g="255" b="255"> (...)</translate>` indica uma translação animada com: duração de 1 segundo, uma trajetória visual definida por 100 segmentos (quantos mais segmentos melhor será a qualidade do desenho da trajetória) e colorida a branco.

É possível observar a movimentação dos planetas usando este mesmo comando na figura abaixo.



5.2 Atalhos

Nesta secção iremos abordar alguns dos atalhos que foram criados neste projeto de modo a melhorar a experiência do utilizador.

A tecla "f" apenas mostra os pontos das figuras desenhadas.

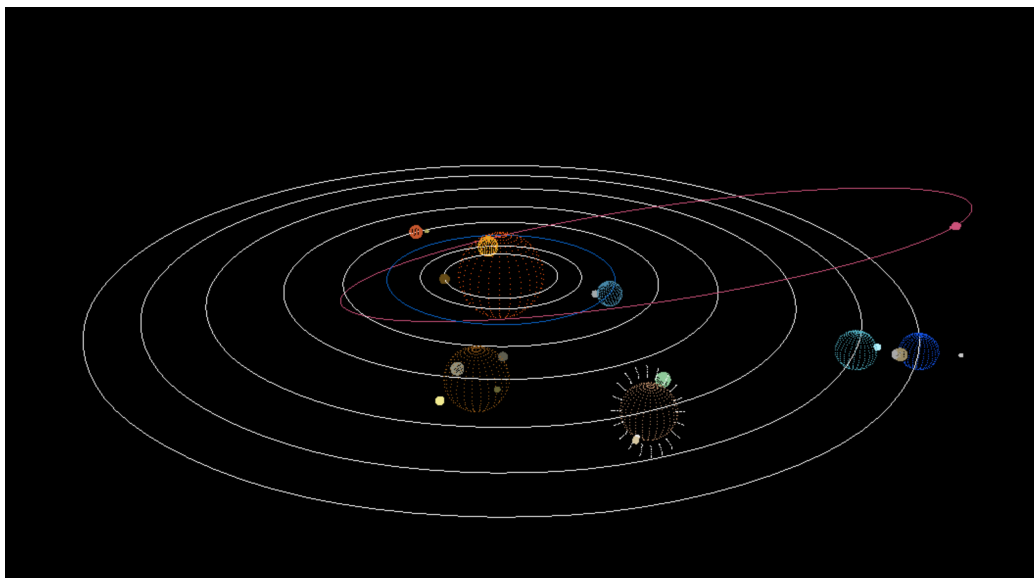


Figura 1: Tecla 'f'

A tecla "d" apenas mostra os triângulos das figuras (não preenchidos).

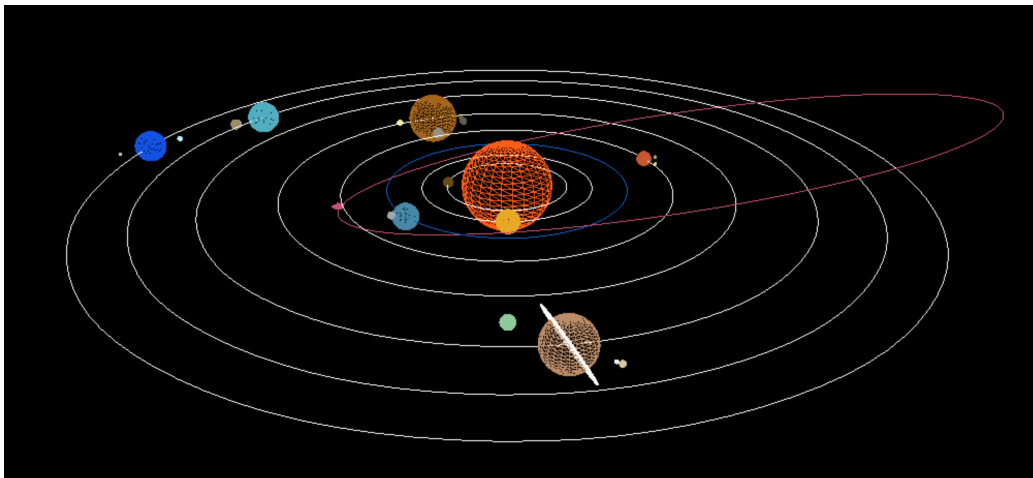


Figura 2: Tecla 'd'

A tecla "s" mostra as figuras preenchidas.

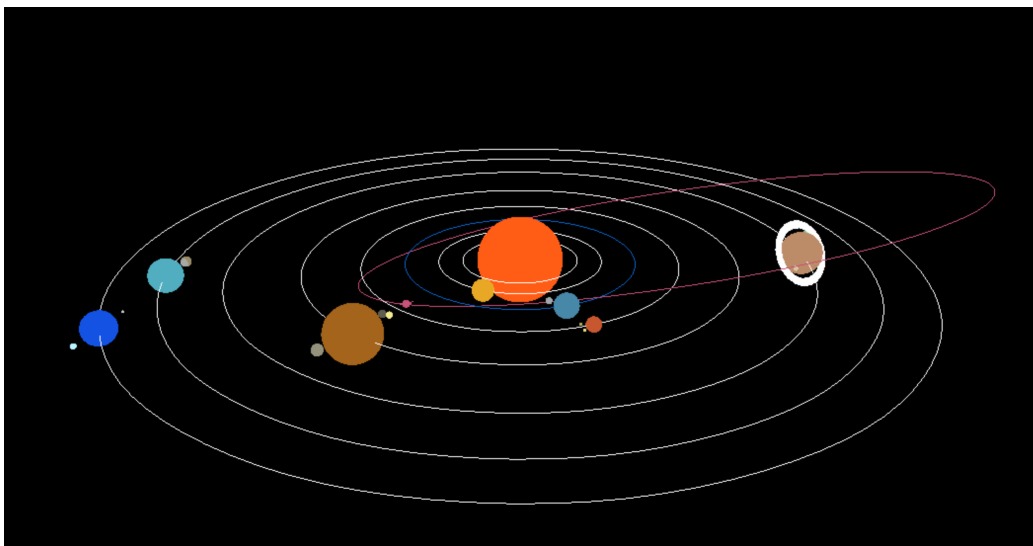


Figura 3: Tecla 's'

A tecla "a" ativa os referenciais relativos a cada figura individual, assim como desativa caso já estejam ativados.

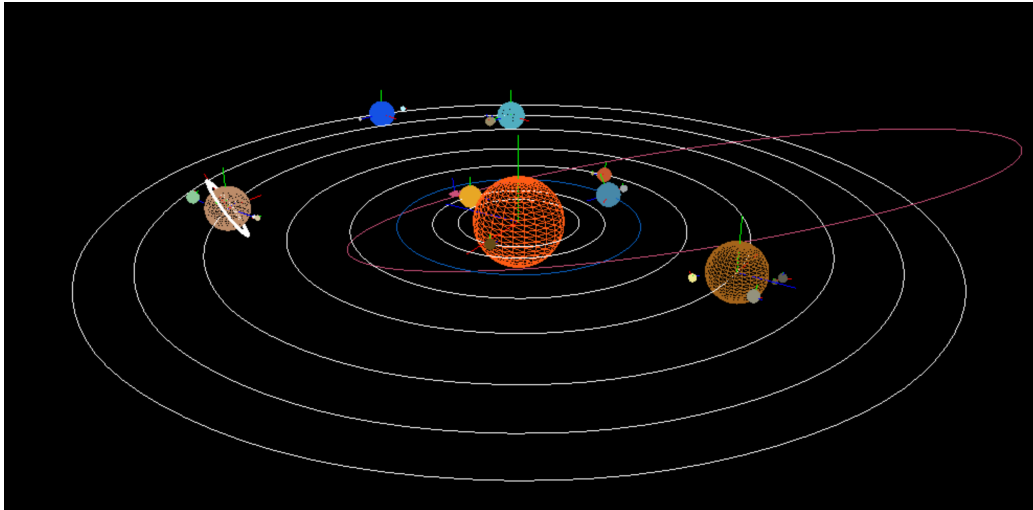


Figura 4: Tecla 'a'

A tecla "t" ativa o desenho das trajetórias (caso sejam o caso), assim como desativa caso já estejam ativadas.

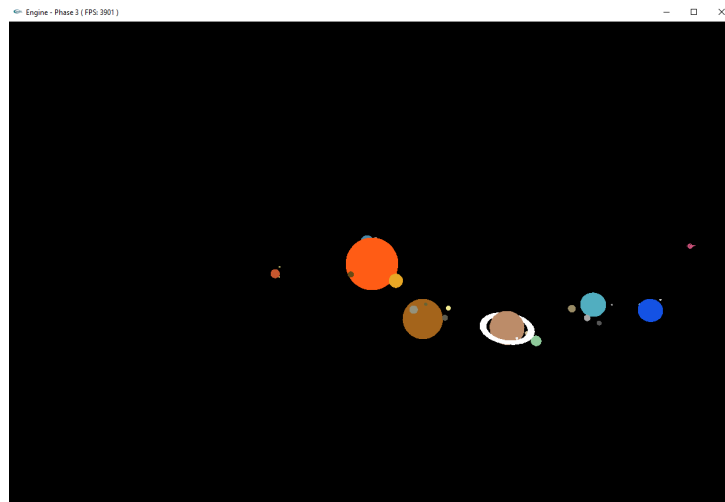


Figura 5: Tecla 't'

A tecla ”c” ativa o sistema de eixos central, assim como desativa caso já esteja ativado.

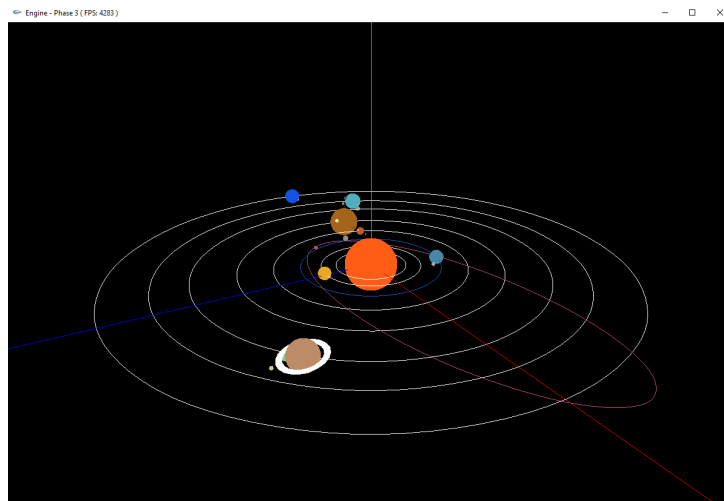


Figura 6: Tecla 'c'

A tecla ”p” tem uma funcionalidade semelhante a um botão de pausa, ou seja, desativa qualquer animação (translação e rotação) até que volte a ser clicada.

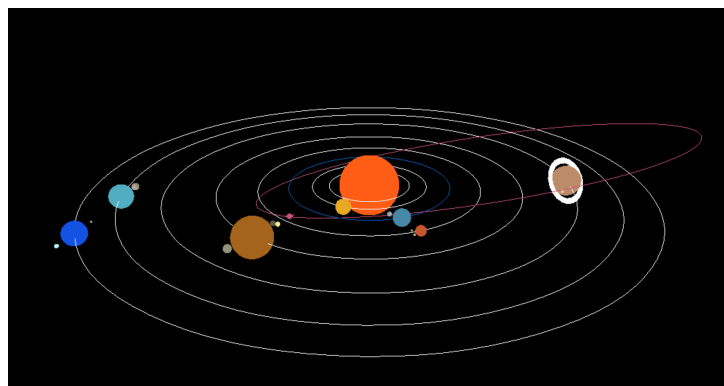


Figura 7: Tecla 'p'

6 Conclusão

Em suma, nesta terceira fase do trabalho, o grupo considera que já aparenta ter um conhecimento da matéria mais aprofundado, tendo em conta que tudo o que foi pedido no enunciado foi conseguido, e que ainda deu tempo para adicionar uns extras.

Foi dada a oportunidade de perceber o quão útil é o uso de VBOs, e como funcionavam os modelos dinâmicos. Também adquirimos mais conhecimentos à cerca de superfícies cúbicas.

Por fim, terminada esta terceira fase, o objetivo será começar já a trabalhar na ultima fase do projeto, de modo a tornar o modelo mais realista e mais apelativo, e continuando assim aprofundar o conhecimento sobre Computação Gráfica.