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

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 Catmul-Rom. Para isso, definimos uma classe que abstrai todo este comportamento.

```
class CatmullRom {
  int points; // total de pontos da trajetoria
  vector<Point> segPoints; // pontos da trajetoria

  float gt; // tempo entre 0 e 1 para calcular a posicao num
     segmento
  float animationTime; // tempo total de animacao

  float pos[3]; // vetor que da a posicao nos eixos x,y,z
  float deriv[3]; // vetor que indica derivada no ponto
     calculado
  float z[3]; // vetor que indica as coordenadas do eixo z
  float up[3] = { 0,1,0 }; // vetor para indicar o sentido
     que    para "cima" por defeito

  int segments; // numero total de segmentos
  float* rgb; // cor da trajetoria (caso seja o caso de ser
     desenhada)
  GLuint vboID; // indice no buffer do VBO
  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*:

```
void animatedTranslate(float elapsed_time) {
  float m[4][4], m_transpose[4][4];
  this->gt = elapsed_time / this->animationTime; // elapsed
      time
  this->gt -= floor(gt); // has to be between 0 and 1

  getGlobalCatmullRomPoint(gt, pos, deriv);

  normalize(deriv); // processos de normalizacao e calculo
     vetorial
  cross(deriv, up, z);
  normalize(z);
  cross(z, deriv, up);
  normalize(up);
  glTranslatef(pos[0], pos[1], pos[2]); // translacao para
     a posicao correta
  buildRotMatrix(deriv, up, z, *m); // construir a matriz
     de rotacao
  transpose(*m, *m_transpose); // calcular a transposta
  glMultMatrixf(*m_transpose); // multiplicar matriz atual
     (viewmodel) com as transformacoes finais
}
```

2.2 VBO's

Os VBO são uma ótima medida de garantirmos uma elevada perfomance 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**.

```
void getTransformations(string f) {
```

```
    GLuint nFig = 0;
    transformations = new vector<Transformations*>();

    xmlReader(f, transformations, &nFig); // nFig da nos o
       numero de models dentro do ficheiro XML

    figures = new GLuint[nFig](); // alocacao de espacos para
        os indices relativos a cada vboID

    glGenBuffers(nFig, figures); // criacao do VBO

    for (Transformations* t : *transformations) {
      t->addReferenceBuffer(figures); // adicionar a
         referencia para o VBO para posteriormente desenhar a
          respetiva fatia de cada transformacao
      t->start();  // inicializar a "fatia" respetiva do VBO
    }

    cout << "Scene: " << f << " loaded!" << endl;
  }
```

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.

```
  void start() {
    for (Model* m : this->models) {
      m->prepareModel(this->buffer); // mover os pontos do
         modelo para a placa grafica
    }

    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)

    for (Transformations* t : this->subgroups) {
        t->start(); // recursivamente aplicar aos subgrupos
    }
  }
```

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.

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

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

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

2.2.2 CatmullRom (Trajetória Gráfica)

Muito semelhante aos models anteriores, a diferença é que os pontos são retirados usando as primitidas 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.

```
  void prepareCurve() {
    float* loopPoints = new float[segments * 3];
    int j = 0;
    float t;
    for (int i = 0; i < this->segments; i++) {
      t = i / (float) segments;
      getGlobalCatmullRomPoint(t, pos, deriv);
      loopPoints[j] = pos[0];
      loopPoints[j + 1] = pos[1];
      loopPoints[j + 2] = pos[2];
      j += 3;
    }

    glBindBuffer(GL_ARRAY_BUFFER, buffer[vboID]); // // criar
        o buffer de VBO na placa grafica
    glBufferData(GL_ARRAY_BUFFER, sizeof(float) * segments *
       3, loopPoints, GL_STATIC_DRAW); // colocar os pontos
       nesse buffer
  }

  void traceCurve() {
    glPushMatrix(); // nao alterar as outras transformacoes
    glColor3f(this->rgb[0], this->rgb[1], this->rgb[2]); //
       // cor escolhida para a curva
    glBindBuffer(GL_ARRAY_BUFFER, buffer[vboID]); // criar
       apontador para o buffer
    glVertexPointer(3, GL_FLOAT, 0, 0); // quantos vertices e
        que tipo (neste caso floats) sao
    glDrawArrays(GL_LINE_LOOP, 0, segments); // agora em vez
       de triangulos e um line-loop
    glPopMatrix(); // voltar a matriz de transformacoes
       antiga
  }
};
```

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.

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

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.

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

2.3.2 Translação

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

```
class Translate {
public:
  bool animated; // se e dinamica ou nao
  bool traced; // se o utilizador pretende mostrar
     graficamente a trajetoria
  bool toggledTrace; // para ativar/desativar a trajetoria
  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.

```
void parseRotate(XMLElement* rotate, Transformations*
    transforms) {
  x,y,z = Atributo x,y,z do elemento XML;
  se existir atributo "time":
    new Rotate(x,y,z,valor desse atributo,true);
  se nao
    new Rotate(valor do atributo "angle", x, y, z);
  }
  transforms->addRotate(r);
}

void parseTranslate(XMLElement* translate, Transformations*
    transforms, unsigned int* nFig) {
  se tiver o atributo "time":
    float time = valor desse atributo;
    int seg;
    bool traced = false;
    se tiver o atributo "traced":
      seg = o valor desse atributo;
      traced = true;
    se nao:
      seg = 0;
    vector<Point> controlPoints;
    para todos os pontos dentro do elemento translate:
      inseri-los em controlPoints;
    }
    se (traced):
      float* rgb = new float[3]();
      se tiver os atributos "r", "g" e "b":
        rgb[0] = valor do atributo "r" / 255.0f;
        rgb[1] = valor do atributo "g"  / 255.0f;
        rgb[2] = valor do atributo "b"  / 255.0f;
      se nao:
        rgb[2] = rgb[1] = rgb[0] = 1.0f;
        transforms->addTranslate(new Translate(
            controlPoints, time, seg, *nFig, rgb));
        (*nFig)++;
      se nao:
      transforms->addTranslate(new Translate(controlPoints,
          time, seg));
  se nao:
    transforms->addTranslate(new Translate(valor do
       atributo "x", valor do atributo "y", valor do
       atributo "y"));
}
```

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 indíces de pontos de controlo. Cada linha contém 16 indíces, e cada um representa um ponto de controlo desse patch (Passo 2). A linha seguinte, que será 1+**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 contêm 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
         control points
3     //cada linha tera 16 indices
4     //para guardar a info usamos uma matriz -> indicesCP[
         numPatch][16]
5     indicesCP = new int* [numPatch];
6     for (int i = 0; i < numPatch; i++) {
7       indicesCP[i] = new int[16];
8       getline(readFile, linha);
9       char* arrayLinha = strdup(linha.c_str());
10      char* point = strtok(arrayLinha, ",␣");
```

```
      for (int j = 0; point != NULL; j++) {
        indicesCP[i][j] = stoi(point);
        point = strtok(NULL, ",␣");

    }}
```

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

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

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.

```
    if (writeFile.is_open()) {

      float incremento = 1.0 / tesselation;
      float coord0[3], coord1[3], coord2[3], coord3[3];

      /*pontos    0 . ---- . 3
                    |      |
                    |      |
                  1 . ---- . 2
      */

      // 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,
                 coord0);
21            bezierPatches(u, v1, indicesCP[i], valuesCP,
                 coord1);
22            bezierPatches(u1, v1, indicesCP[i], valuesCP,
                 coord2);
23            bezierPatches(u1, v, indicesCP[i], valuesCP,
                 coord3);
24
25
26            writeFile << coord0[0] << "␣" << coord0[1] << "␣"
                   << coord0[2] << endl;
27            writeFile << coord1[0] << "␣" << coord1[1] << "␣"
                   << coord1[2] << endl;
28            writeFile << coord2[0] << "␣" << coord2[1] << "␣"
                   << coord2[2] << endl;
29
30            writeFile << coord0[0] << "␣" << coord0[1] << "␣"
                   << coord0[2] << endl;
31            writeFile << coord2[0] << "␣" << coord2[1] << "␣"
                   << coord2[2] << endl;
32            writeFile << coord3[0] << "␣" << coord3[1] << "␣"
                   << coord3[2] << endl;
33          }
34        }
35      }
36    }
```

3.1.2 Fórmula de Bezier

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

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

Calculamos então todas essas componentes.

```
float U[1][4] = { { pow(u,3) , pow(u,2) ,u , 1 } };

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 MT[4][4];
transpose(*M, *MT);

float V[4][1] = { {pow(v,3)}, {pow(v,2)}, {v}, {1} };

for (int i = 0; i < 4; i++) {
  for (int j = 0; j < 4; j++) {
    cpX[i][j] = valuesCP[indicesCP[i * 4 + j]][0];
    cpY[i][j] = valuesCP[indicesCP[i * 4 + j]][1];;
    cpZ[i][j] = valuesCP[indicesCP[i * 4 + j]][2];;
  }
}
```

Por fim, fazemos as multiplicações entre as matrizes.

```
// aXc * cXb
//1X4 * 4X4  a=1;c=4;b=4
multMatrix(*U, *M, *res, 1, 4, 4);

//1X4 * 4X4  a=1;c=4;b=4
multMatrix(*res, *cpX, *resX, 1, 4, 4);
multMatrix(*res, *cpY, *resY, 1, 4, 4);
multMatrix(*res, *cpZ, *resZ, 1, 4, 4);

//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

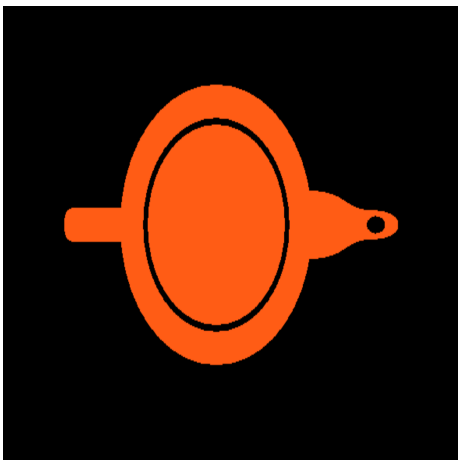
Foi agora adicionado um novo comando ao gerador:

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

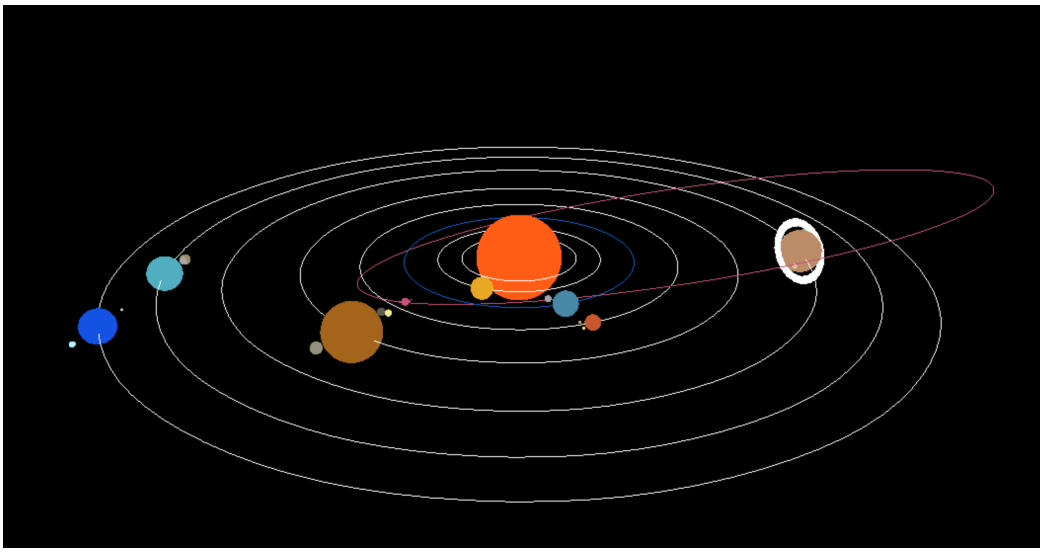
Que permite, agora, a criação de uma figura utilizando patches de Bezier. Os parâmetros são o ficheiro de *input* (em formato patch tal como o fornecido), o ficheiro de *output*, que será em formato 3d tal como os que temos trabalhado até agora e a tesselação, que indica a "qualidade"do desenho. Quanto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

4 Resultado

4.1 Teapot (Bezier Patches)



4.2 Sistema Solar



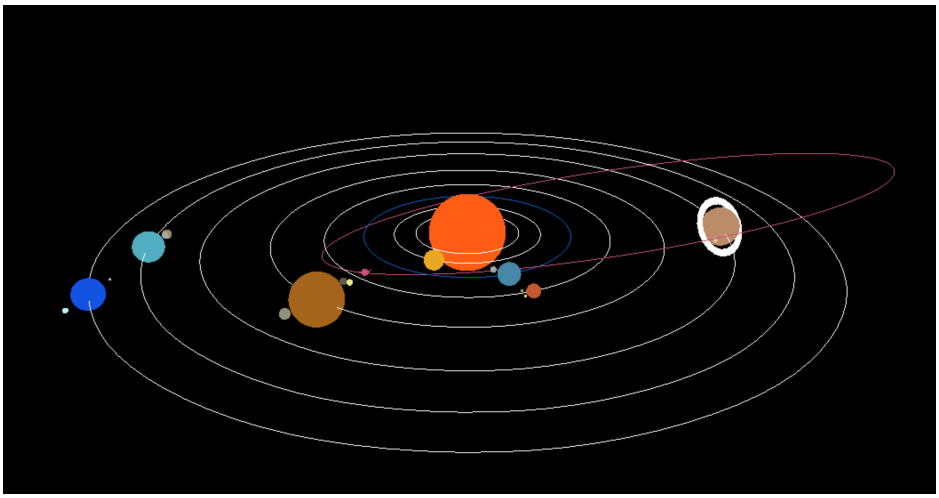
5 Extras

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.

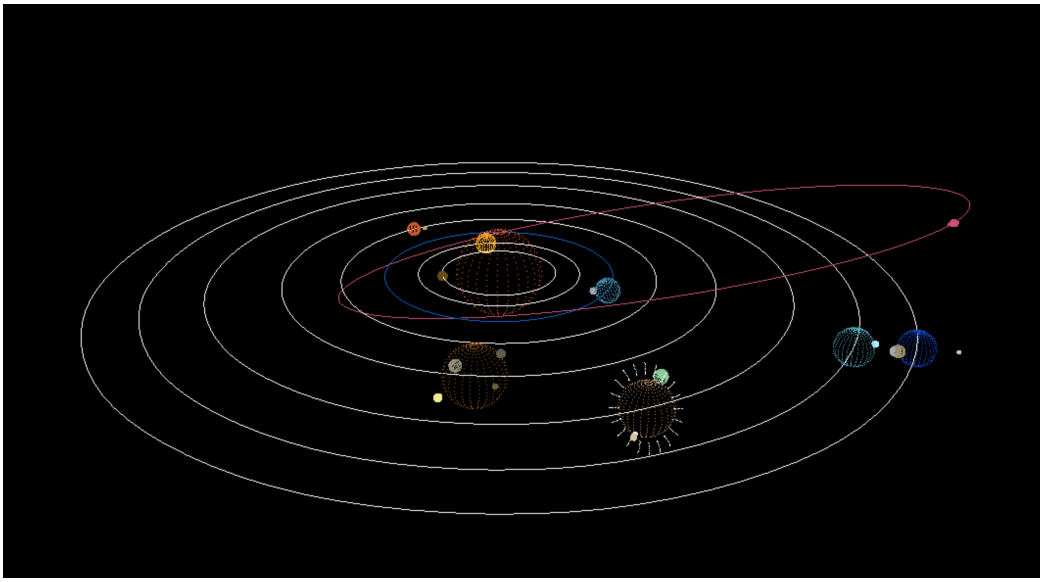


Figura 1: Tecla 'f'

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

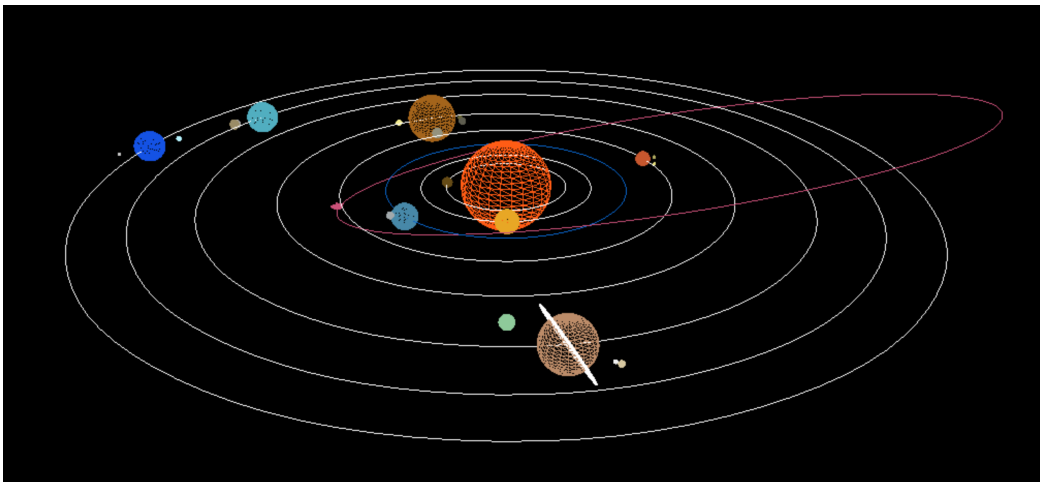


Figura 2: Tecla 'd'

A tecla **"s"** mostra as figuras preenchidas.

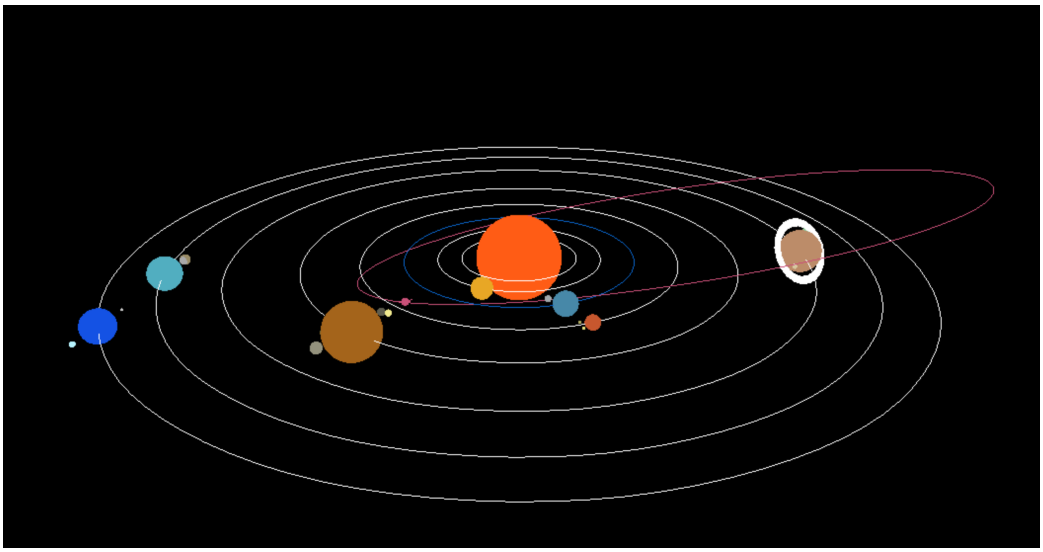


Figura 3: Tecla 's'

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

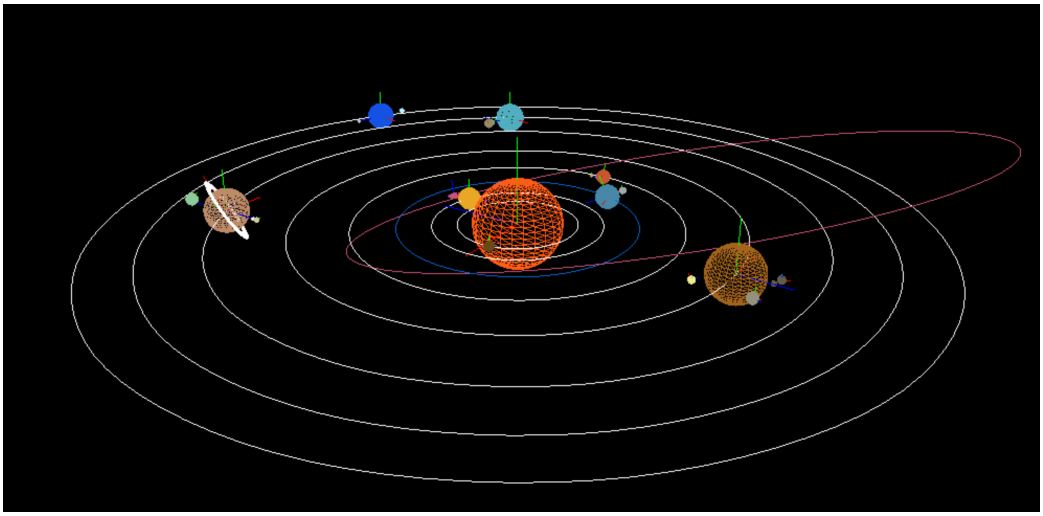


Figura 4: Tecla 'a'

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

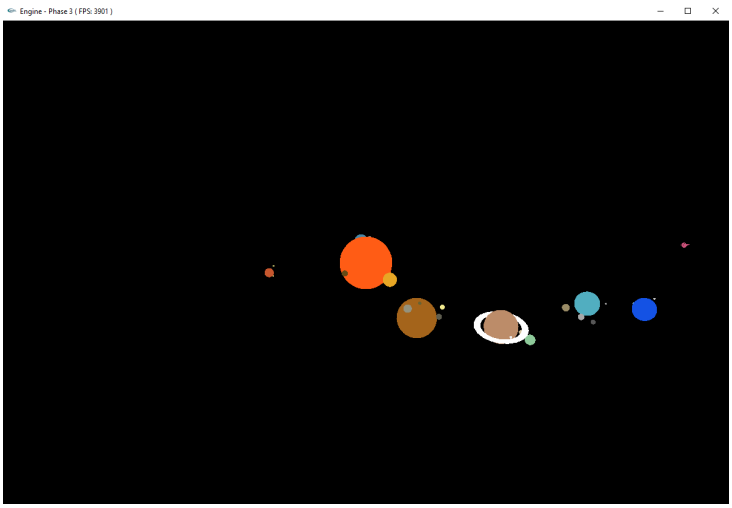


Figura 5: Tecla 't'

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

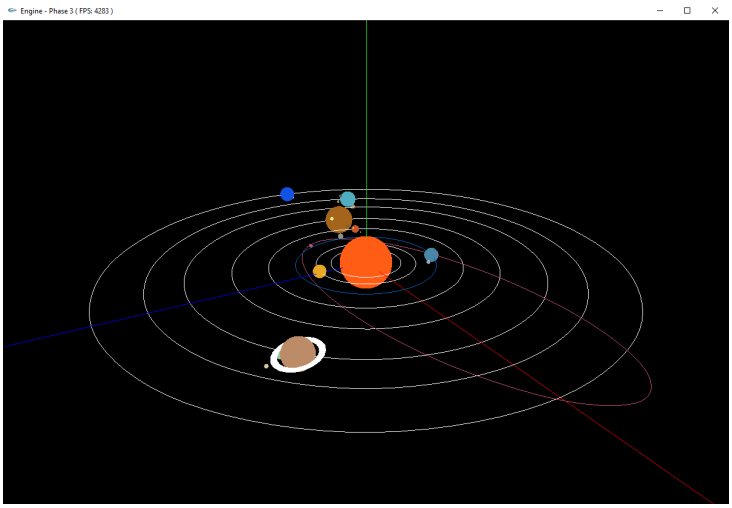


Figura 6: Tecla 'c'

A tecla **"p"** desativa rotação do sistema solar (caso esteja ativada), assim como volta ativar a rotação, caso esteja desativada.

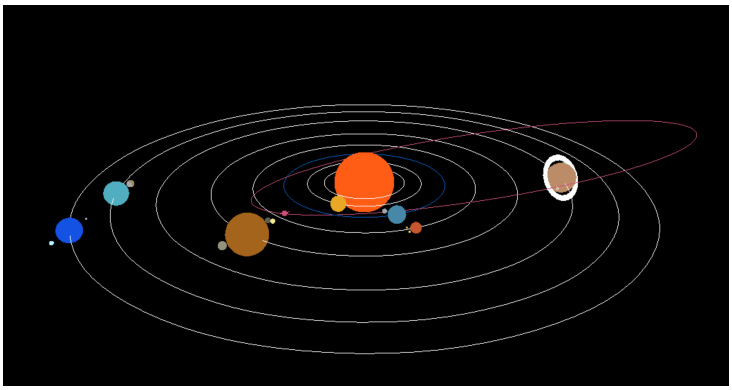


Figura 7: Tecla 'p'

6 Conclusão

Em suma, nesta terceira fase do trabalho, o grupo considera que já aparenta ter um conhecimento da matéria mais aprodundado, 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 superficies 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.