Universidade do Minho
Escola de Engenharia

Computação Gráfica

Trabalho Prático - Fase 3

2 de Maio de 2021

Grupo 23

Carlos Filipe Coelho Ferreira, A89542
Joel Salgueiro Martins, A89575
José Carlos Leite Magalhães, A85852
Paulo Ricardo Antunes Pereira, A86475




Conteúdo

1 Introdução

No âmbito da unidade curricular de Computação Gráfica foi proposto desenvolver um motor 3D, baseado num cenário gráfico, cujo propósito passa por explorar todas as suas potencialidades e capacidades. Esta exploração é feita através de diversos exemplos visuais e interativos, sendo utilizadas, para esse efeito, todas as ferramentas abordadas nas aulas práticas da disciplina.

O projeto encontra-se dividido em quatro fases, sendo que este relatório, estando as primeiras duas concluídas, versará sobre a terceira etapa. Esta fase incide sobretudo na aplicação de uma série de novos conceitos, como a animação de modelos e a introdução de VBOs, a fim de testar a melhoria de performance relativamente às fases passadas.

Grande parte das estruturas implementadas na etapa anterior sofreram pequenas alterações para, por exemplo, permitir o uso de curvas Catmull Rom para animar os modelos e também para conseguir desenhar modelos através de Bezier Patches.

Mais uma vez todos estes novos requisitos serão aplicados no já complexo Sistema Solar, sendo este a prova gráfica de que tudo está a funcionar como deveria.

2 Motor

2.1 VBO's (*Vertex Buffer Object)*

Um VBO pode ser visto como um *array* residente na memória da placa gráfica. Tirando partido desta componente, permite ao nosso programa garantir níveis de performance bastante elevados pois estamos, por exemplo, a aproveitar o potencial paralelismo disponibilizado pelo GPU.

2.1.1 Implementação

A primeira parte do processo de implementação passou por determinar não só quantos modelos estão presentes no XML, mas também o número de translações animadas, dos quais se pretende desenhar uma trajetória, presentes no mesmo, de modo a criar um vetor com tamanho suficiente para alocar todos estes índices. Depois disto, é feita a criação do VBO, pelo que de seguida se procede à formação de VBO a partir dos pontos carregados, tanto no *.3d* como no XML.

De seguida, é possível verificar a implementação de todo o processo descrito anteriormente:

```
void getTransf(string f) {
    transf = new Transformacao();
    leitorXML *a = new leitorXML(f, transf);
    transf = a->leXML();

    //nr de modelos presentes no XML:
    int nFig = transf->getSizeModelos();
    //nr de translacoesAnimadas:
    int nTranslatedAnimated = transf->getNTranslatedAnimated();

    //O buffer dos VBOs deve ter tamanho para todos os modelos
    //e para todas trajetorias em translacoes animadas:
    GLuint nbuffer = (GLuint)(nFig + nTranslatedAnimated);
    //E feita a alocacao dos indices:
    figures = new GLuint[nbuffer]();
    //Inicializacao do VBO:
    glGenBuffers(nbuffer, figures);

  //Formar os vbo a partir dos pontos carregados nos .3d e xml:
    transf->prepare(figures);

 cout << "Cenario: " << f << " carregado!" << endl << endl <<
    endl;
}
```

Analisando com cuidado o código presente na linha 20, percebe-se que é a classe ***Transformacao*** a responsável por formar os VBO. Esta classe é definida da seguinte forma:

```
class Transformacao {
        vector<Instrucao*> instrucoes;
        vector<Modelo*> models;
        GLuint* buffer;
        int nTranslatedAnimated=0;
//(...)
};
```

Por fim, a função responsável por formar os VBO, tanto para os modelos como para as trajetórias animadas, está presente nesta classe, e é a seguinte:

```
class Transformacao {
//(...)
void prepare(GLuint* figures) {
        this->buffer = figures;
        //Forma VBO para todos os modelos.
        for (Modelo* m : this->models) {
                m->formaVbo(buffer);
        }
        int tbuffer = this->models.size();
                //Forma VBO para todas as trajetorias
                for (Instrucao* i : this->instrucoes) {
        //So forma Vbo se for para desenhar trajetoria:
                  if (i->eParaDesenharTrajetoria()) {
                        i->prepare(buffer, tbuffer);
                                tbuffer++; }
                }
        }//(...)
```

Acrescenta-se que as funções *formaVbo()*, referente à classe ***Modelo***, e a *prepare()*, respetiva à classe ***Instrucao***, serão vistas nas secções seguintes.

2.1.2 Modificação Modelo

A classe ***Modelo***, já definida na fase anterior, sofreu uma pequena alteração, mais concretamente a adição de uma variável, correspondente ao índice do vetor com os respetivos índices do VBO.

```
class Modelo {
        vector<Ponto> pontos;
        string nome;
        GLuint vboID;
//(...)
};
```

De seguida é necessário proceder ao "povoamento"dos VBO's, utilizando os pontos já guardados, para depois serem transferidos para a gráfica, pelo que é feito da seguinte forma:

```
class Modelo {
//(...)
void formaVbo(GLuint* buffer) {
    int size = this->pontos.size(); //total de pontos a guardar
    float* p = new float[size * 3];
    int i = 0;
    for (int j = 0; j < size; j++) {
      // Processo de desdobramento do Ponnto
      p[i] = this->pontos.at(j).x;
      p[i + 1] = this->pontos.at(j).y;
      i += 3;
    }
    // Criar o buffer de VBO na placa grafica
    glBindBuffer(GL_ARRAY_BUFFER, buffer[this->vboID]);
    // Coloca os pontos no buffer especifico
    glBufferData(GL_ARRAY_BUFFER, sizeof(float) * size * 3, p,
       GL_STATIC_DRAW);
}
/(...)
};
```

Por fim, a função responsável pela representação gráfica dos VBO's foi baseada nas práticas da disciplina, e é definida da seguinte forma:

```
void desenhaPontos(GLuint* buffer) {
    //Apontador para o buffer:
    glBindBuffer(GL_ARRAY_BUFFER, buffer[vboID]);
    //Numero de vertices e o seu tipo:
    glVertexPointer(3, GL_FLOAT, 0, 0);
    //Ate acabarem os pontos, sao desenhados triangulos:
    glDrawArrays(GL_TRIANGLES, 0, pontos.size());
}
```

2.2 Curvas Catmull Rom

Para esta fase foi pedida a implementação de translações dinâmicas e, de modo a concretizar isso, foi necessário trabalhar com curvas de Catmull Rom. Então, foi necessário adaptar conteúdos já abordados nas aulas práticas, pelo que, de modo a simplificar, decidiu-se definir uma classe que aglomere os atributos necessários:

```
class TranslateAnimated {
        float timeAnimacao; //tempo total de animacao
        vector<Ponto> trajetoria; //pontos da trajetoria
        int nPontos; //nr total de pontos
```

```
        float pos[3]; //vetor da posicao X, Y e Z
        float deriv[3]; //vetor da derivada no ponto calculado
        float z[3]; //vetor das coordenadas eixo dos  Z
        float up[3] = { 0,1,0 }; //sentido inicial = Eixo Dos Y
        int desenhaTrajetoria = 0;

        GLuint* buffer; // referencia do vetor VBO
        GLuint indice; //indice correspondente ao buffer no VBO
```

De forma a obter os pontos em que o objeto se encontra, em cada *frame*, foram adaptadas duas funções dadas nas aulas práticas da disciplina. Acrescenta-se que, tendo em conta que a sua definição foi fornecida e trabalhada em tempo útil letivo, o grupo não achou necessidade em colocar a sua definição, apenas as seguintes notas:

- Função auxiliar, utiliza 4 pontos da trajetória (passados como argumento), e calcula, para um determinado valor de **t**, a posição e a derivada do objeto em questão:

```
void getCatmullRomPoint(float t, Ponto p0, Ponto p1, Ponto
    p2, Ponto p3, float* pos, float* deriv) {
//(...)
}
```

- Função principal, chama a anterior, e utiliza 4 pontos da trajetória, dependendo do segmento onde se encontra, representado por **tempoRelativo**, guardando em **pos** o ponto calculado e em **deriv** a respetiva derivada:

```
void getGlobalCatmullRomPoint(float tempoRelativo, float*
   pos, float* deriv) {
//(...)
}
```

Por fim, e encarregue de realizar a translação animada, dependendo da posição calculada, está a seguinte função, devidamente documentada:

```
void apply(float timeAtual,bool traj) {
        float matrizRotacao[4][4];
        float tempoRelativo;
        /*
        Por exemplo, se tempoRelativo=4.2/2=2.1
                entao fez 2 voltas e 0.1=10% de outra
        */
        tempoRelativo = timeAtual / this->timeAnimacao;

        //Percentagem de animacao [0,1]
        tempoRelativo -= floor(tempoRelativo);


```

```
13          //Se pretender trajetoria, invoca a responsavel por
               desenhar a mesma
14          if (traj) traceCurve();
15
16          //Usa-se a percentagem da animacao atual
17          getGlobalCatmullRomPoint(tempoRelativo, pos, deriv);
18
19          /*
20          A partir daqui comecam os calculos vetoriais:
21          */
22          //X(i)=P'(T)=deriv
23          normalize(deriv);
24
25          //Z(i)=X(i) x Y(i-1)
26          cross(deriv, up, z);
27          normalize(z);
28
29          //Y(i)=Z(i) x X(i)
30          cross(z, deriv, up);
31          normalize(up);
32
33          //Por o modelo no sitio correto:
34          glTranslatef(pos[0], pos[1], pos[2]);
35          //Matriz de rotacao do objeto:
36          buildRotMatrix(deriv, up, z, *matrizRotacao);
37          //Aplica matriz de rotacao:
38          glMultMatrixf(*matrizRotacao);

39  }
```

2.2.1 Extra - Desenho da trajetória

De modo a fazer ainda mais usufruto das propriedades das curvas Catmull Rom, decidiu-se aplicar de uma forma diferente, com o objetivo de implementar as trajetórias gráficas dos vários elementos do Sistema Solar. Através desta funcionalidade extra, a nossa *demo scence* fica agora mais complexa, mas com um impacto visual bem mais agradável.

O processo é simples. Se o utilizador pretender o desenho das trajetórias, então é ativada uma *flag*. Uma vez sinalizada a vontade, a classe responsável por executar o pedido é a já referida ***TranslateAnimated***, pelo que a sua implementação é semelhante ao já visto nas secções anteriores com a diferença de ser necessário criar um VBO, com o objetivo de desenhar um ciclo de linhas, tendo por base o número de segmentos representados no ficheiro XML.

```
class TranslatedAnimated {
//(...)
void prepare(GLuint* buffer, GLuint tbuffer) {
  if (desenhaTrajetoria != 0) {
          this->buffer = buffer;
          this->indice = tbuffer;
          float* arrayPontos = new float[desenhaTrajetoria * 3];
          float gt;       //Percentagem [0-1]
          int p = 0;
          //desenhaTrajetoria influencia o numero de pontos(
             precisao) da trajetoria a desenhar.
          for (int i = 0; i < this->desenhaTrajetoria; i++) {
           //Conseguir um numero de [0,1]:
           gt = i / (float)desenhaTrajetoria;
           //Calular posicao:
           getGlobalCatmullRomPoint(gt, pos, deriv);
           //Guardar posicao no array de floats:
           arrayPontos[p] = pos[0];
           arrayPontos[p + 1] = pos[1];
           arrayPontos[p + 2] = pos[2];
           p += 3;
           }
           /*
           Criar o buffer de VBO e colocar os pontos nesse
              buffer:
           */
           glBindBuffer(GL_ARRAY_BUFFER, buffer[this->indice]);
           glBufferData(GL_ARRAY_BUFFER, sizeof(float) *
              desenhaTrajetoria * 3, arrayPontos, GL_STATIC_DRAW
              );
        }
 } //(...)
```

Por fim, a função responsável por desenhar a curva é bastante simples:

```
void traceCurve() {
    if (desenhaTrajetoria != 0) {
      //Apontador para o buffer
          glBindBuffer(GL_ARRAY_BUFFER, buffer[this->indice]);
          //Numero de vertices e tipo
          glVertexPointer(3, GL_FLOAT, 0, 0);
          /*
          Como ja tinhamos referido, em vez de triangulos, vai
             ter a flag LINELOOP
          */
          glDrawArrays(GL_LINE_LOOP, 0, desenhaTrajetoria);
                }
        }
```

2.3 Modificação Transformações Geométricas

Tendo em conta as novas exigências, também a classe ***Instrucao*** teve de ser modificada. Decidiu-se, então, adicionar aos atributos já existentes dois novos: um *float* **time** (para rotações animadas) e uma referência a uma classe (**TranslateAnimated**) já vista anteriormente, a fim de facilitar translações animadas. Veremos as suas utilidades de seguida.

```
class Instrucao {
   string nome;
   float x;
   float y;
   float z;
   float angle;
   //util para rotacao animada:
   float time;
   //util para translacoes animadas:
   TranslateAnimated* tAnimated;
   Modelo* model;
   //(...)
};
```

2.3.1 Translação

No caso da translação, além do tipo estático já previamente definido, foi necessário acrescentar a vertente dinâmica.

```
class Instrucao {
//(...)
void apply(GLuint* buffer,float tempo,bool traj) {
//(...)
else if (nome == "Translate")
                glTranslatef(this->x, this->y, this->z);
else if (nome == "TranslateAnimated")
                this->tAnimated->apply(tempo,traj);
//(...)
};
```

Acrescenta-se em forma de nota que a função *apply*, referente à classe ***TranslateAnimated***, encontra-se definida e comentada na página número 7 deste documento. É nela que, se for para desenhar a trajetória, é feita a invocação à função *traceCurve*, responsável por desenhar curva da trajetória.

A curva a desenhar é previamente "preparada", no momento em que é implementada na formação de VBOs, já vista anteriormente, sendo, no momento da criação, invocada a seguinte função:

```
class Instrucao {
//(...)
void prepare(GLuint* buffer,int tbuffer) {//Por os pontos das
   trajetorias nos Vbo para desenhar
                if (this->nome == "TranslateAnimated")
                        tAnimated->prepare(buffer,tbuffer);//
                           buffer e o indice no array de Vbo.
        } //(...)
```

Por fim, e como é possível verificar analisando o excerto de código, é chamada a função *prepare*, referente à classe ***TranslateAnimated***, presente na subsecção anterior.

2.3.2 Rotação

No caso específico da rotação, as alterações não foram substanciais, apenas foi necessário garantir a implementação dos dois tipos possíveis: estática, como na fase anterior, ou dinâmica.

```
class Instrucao {
//(...)
void apply(GLuint* buffer,float tempo,bool traj) {
//(...)
else if (nome == "Rotate")
    //O angulo e definido estaticamente:
        glRotatef(this->angle, this->x, this->y, this->z);

else if (nome == "RotateAnimated")
    //O angulo atual e calculado consoante o tempo:
        glRotatef(((tempo / this->time) * 360), this->x, this->
           y, this->z);
//(...)
};
```

2.4 Interpretação de ficheiros XML

Uma das principais diferenças entre as fases anteriores e a que nos encontramos é o facto de, nesta etapa, ser necessário estender o conceito de translação e rotação de elementos. Então, foi necessário tratar destas duas transformações de forma diferente, criando uma função para cada uma delas.

Assim, o ficheiro XML, terá novos elementos, com novos atributos, seguindo obrigatoriamente o seguinte formato:

```
//(...)
//No caso de ser uma translacao animada:
<translate time=10 >
      <point X=1 Y=0 Z=1 />
      <point X=0.707 Y=0.707 Z=1 />
      <point X=0 Y=1 Z=1 />
      ...
      <point X=-1 Y=0 Z=1 />
</translate>
//(...)
//No caso de ser uma rotacao animada:
<rotate time=10 axisX=0 axisY=1 axisZ=0 />
//(...)
```

Recordemos, então, a classe criada na fase passada, responsável pela leitura e interpretação de ficheiros XML, ***leitorXML***, definida pelo seguinte:

```
class leitorXML {
        string ficheiro;
        Transformacao* transf;
    //(...)
};
```

O processo de leitura e interpretação é feito utilizando várias funções, auxiliadas pela biblioteca *tinyxml*, já usada nas fases anteriores. Entre elas, a função *parseGroup(...)*, responsável por verificar que tipo de elemento está a ser lido, bem como proceder ao processamento adequado.

Todos os tipos de elementos das fases anteriores são mantidos integralmente, bem como o seu tratamento. Aos elementos **translate** e **rotate**, foi adicionada, além do que já tinham associado, uma nova vertente - animações.

Então, a já referida *parseGroup(...)*, levou as seguintes adições:

```
void parseGrupo(XMLElement* group) {
 transf->addPush();
 for(XMLElement* elem = group->FirstChildElement(); elem;
  elem = elem->NextSiblingElement()){
  string type = elem->Value();

  if (type.compare("translate") == 0) {
    if (elem->FindAttribute("time")) { //Translacao animada
          parseAnimatedTranslate(elem);}
     else{ //Translacao normal
     transf->addTranslate(elem->FloatAttribute("x"), elem->
        FloatAttribute("y"), elem->FloatAttribute("z"));}
     }

   else if (type.compare("rotate") == 0) {
```

```
15          if (elem->FindAttribute("time")) {//Rotacao animada
16          transf->addRotateAnimated(elem->FloatAttribute("x"),
               elem->FloatAttribute("y"),elem->FloatAttribute("z"),
               elem->FloatAttribute("time"));}
17          else{ //Rotacao normal
18          transf->addRotate(elem->FloatAttribute("x"), elem->
               FloatAttribute("y"), elem->FloatAttribute("z"), elem
               ->FloatAttribute("angle")); }
19          }
20      //(...)
21  }
```

Então, analisando o código anterior, conclui-se que, se o elemento for do tipo **rotate**, então este pode seguir dois caminhos:

- Se a *keyword* "**time**"estiver presente na parcela XML correspondente, então estamos perante uma rotação animada. Neste caso, então é atualizada, além das coordenadas, a variável **time**, vista anteriormente.

```
1  void addRotateAnimated(float x, float y, float z,float
      time) {
2           Instrucao* i = new Instrucao();
3           i->rotateAnimated(x, y, z,time);
4           instrucoes.push_back(i);
5  }
```

- Caso contrário, então estamos perante uma rotação simples, em conformidade com o já implementado na fase anterior.

Se estivermos perante uma transformação **translate**, então:

- Se a condição presente na linha 8 for *True*, então estamos perante uma translação animada e, para tal, é invocada a função *parseAnimatedTranslate*, que se pode definir da seguinte forma:

```
1  void parseAnimatedTranslate(XMLElement* translate) {
2    vector<float> trajetoria;
3    int desenhaTrajetoria = 0;
4    if (translate->FindAttribute("trace")) {
5    //Verifica se e ou nao para desenhar trajetoria
6    desenhaTrajetoria = translate->Int64Attribute("trace");
7                   }
8    for(...){ //itera sobre o ficheiro XML
9           //Efetua o parsing dos pontos:
10          trajetoria.push_back(ponto->FloatAttribute("x"));
11          trajetoria.push_back(ponto->FloatAttribute("y"));
12          trajetoria.push_back(ponto->FloatAttribute("z"));
```

```
13                  }
14          transf->addTranslateAnimated(trajetoria, translate
               ->FloatAttribute("time"),desenhaTrajetoria);
15          }
```

Por fim, temos então a função *addTranslateAnimated*, que faz a inserção da transformação na estrutura, bem como sinalizar se é, ou não, para desenhar a trajetória:

```
1  void addTranslateAnimated(vector<float> trajetoria, float
       time,int desenhaTrajetoria) {
2          Instrucao* i = new Instrucao();
3          i->translateAnimated(trajetoria,time,
               desenhaTrajetoria);
4          instrucoes.push_back(i);
5          if (desenhaTrajetoria!=0)
6          //Se for para desenhar a trajetoria:
7          this->nTranslatedAnimated++;
8          }
```

- Caso contrário, então estamos perante uma translação simples, em conformidade com o já implementado na fase anterior.

2.4.1 Funcionalidade extra

De modo a satisfazer o requisito das trajetórias gráficas, foi necessário adicionar um novo elemento ao ficheiro XML, denominado *trace*. Este indica o número segmentos da linha de trajetória, podendo ser visto um exemplo de seguida:

```
1  //(...)
2  <translate time="10" trace="100">
3      <point x="-30.0" y="0" z="0"/>
4      <point x="0.0" y="0" z="-30.0"/>
5      <point x="30.0" y="0" z="0"/>
6      <point x="0" y="0" z="30.0"/>
7      </translate>
8  //(...)
```

Por exemplo, no segmento acima, estamos perante uma translação animada com a duração de 10 segundos, e uma trajetória visual definida por 100 segmentos.

3 Gerador

O gerador desta fase sofreu uma pequena adição ao já implementado anteriormente, pois, tendo em conta a introdução de curvas de Bezier, foi necessário criar toda uma estrutura de modo a poder gerar os pontos para a construção das referidas curvas.

Acrescenta-se, em forma de nota, que o gerador é responsável apenas pela interpretação dos comandos do utilizador. No caso de pretender gerar Bezier, então é inicializada uma nova classe, ***Bezier***, criada para facilitar toda a organização envolvente.

3.1 Curvas de Bezier

3.1.1 Leitura e interpretação do ficheiro patch

Em primeiro lugar foi necessário interpretar o formato dos ficheiros patch. Estes são divididos essencialmente em quatro partes, pelo que cada uma destas tem uma função própria, como é possível verificar no excerto seguinte:

```
/* Parte 1 -> Numero de patches */
2
/* Parte 2 -> Indices do patch n. Cada linha tem 16 indices */
0, 1, 2, 3, 4, 5 (...) // n=1
3, 16, 17, 18, 7 (...) // n=2
/* Parte 3 -> Numero de control points */
28
/* Parte 4 -> Control points x y z */
1.4, 0, 2.4 // Control point 0
1.4, -0.784, 2.4 // Control point 1
(...)
-1.5, -0.84, 2.4 // Control point 27
```

Tendo em conta esta definição, então decidiu-se que a estrutura responsável por guardar a informação do ficheiro seria definida da seguinte forma:

```
struct DadosPatch {
    int numPatch;  //representa o numero de patches
    int nrPontos;  //representa o numero de pontos
    vector<vector<int>> indices; //representa os indices
    vector<Ponto> pontos; //armazena os varios pontos
};

class Bezier {
    DadosPatch dados;
    //(...)
};
```

Desta forma, definida a estrutura e tendo analisado o formato de um patch, foi necessário implementar uma função dedicada à leitura e interpretação do patch. Esta comporta-se de forma sequencial, estando de acordo com as várias partes da leitura já enumeradas.

A primeira parte, referente ao número de patches, é obtida de forma simples, pois este está presente na primeira linha:

```
class Bezier {
        DadosPatch dados;

public:
 //(...)
 void carregaPatch(string inputF, string outputF, int tess) {
    string linha;
    ifstream readFile(inputF);
    //(...)
    getline(readFile, linha);
    dados.numPatch = stoi(linha);
    //(...)
```

A segunda parte corresponde à informação dos índices. Tenha-se em atenção que, referente a esta parte, serão lidas $numPatch$ linhas. Posteriormente, cada uma destas será "partida", obtendo todos os 16 índices correspondentes, de modo a guardar na estrutura criada para o efeito, já falada anteriormente.

No fim desta ação, ficaremos com um $vector$ com $numPatch$ posições, pelo que em cada uma destas posições estará um novo $vector$, com 16 posições.

```
//(...)
for (int i = 0; i < dados.numPatch; i++) {
        dados.indices.push_back({});
        getline(readFile, linha);
        char* arrayLinha = strdup(linha.c_str());
        char* point = strtok(arrayLinha, ", ");

        for (int j = 0; point != NULL; j++) {
              dados.indices.at(i).push_back(stoi(point));
              point = strtok(NULL, ", ");
        } //(...)
}
```

A parte seguinte foca-se no número de *control points*. Visto ser apenas uma linha, e respeitando o já falado formato do ficheiro, a obtenção deste dado é feita de forma sequencial às partes anteriores, da seguinte forma:

```
//(...)
   getline(readFile, linha);
   dados.nrPontos= stoi(linha);
//(...)
```

Por último, é necessário guardar todos os $nrPontos$ pontos. Para cada um destes *control points*, é lida a linha, efetuado o respetivo *parsing*, preenchida a estrutura *Ponto* e, por fim, procede-se à inserção desta última no $vector$ criado para este efeito.

```
//(...)
for (int i = 0; i < dados.nrPontos; i++) {
                Ponto p;
                getline(readFile, linha);
                char* arrayLinha = strdup(linha.c_str());
                char* point;

                point = strtok(arrayLinha, ", ");
                p.x = stof(point);

                point = strtok(NULL, ", ");
                p.y = stof(point);

                point = strtok(NULL, "\n ");
                p.z = stof(point);

                dados.pontos.push_back(p);

            } //(...)
```

De seguida, é requisito obrigatório que esta informação até agora recolhida seja convertida num ficheiro *.3d*, extensão já utilizada nas outras duas fases deste projeto. Para isso, aproveitou-se a mesma função de leitura, passando agora o objetivo pela escrita.

A primeira etapa deste processo passa por iterar cada patch, posteriormente iterará sobre uma "grelha", cujo tamanho dependerá do valor de $tess$, a variável de *input* referente ao valor *tesselation*.

```
//(...)
    ofstream escreveFicheiro(outputF);
    //(...)
    float u1, v1, u2, v2;
    float p0[3], p1[3], p2[3], p3[3];
    if (escreveFicheiro.is_open()) {
       for (int a = 0; a < dados.numPatch; a++) {
           for (int u = 0; u < tess; u++) {
               for (int v = 0; v < tess; v++) {
                   u1 = (float)u / tess;
                   v1 = (float)v / tess;
```

```
                    u2 = (float)(u + 1) / tess;
                    v2 = (float)(v + 1) / tess;
                //(...)
```

De seguida, para cada uma das "grelhas", vão ser gerados 4 pontos de cada vez, necessários para construir o modelo Bezier, sendo passados, separadamente, por argumento, à função que efetua esta construção:

```
//(...)
    calculaPontoBezier(u1, v1, a, p0);
    calculaPontoBezier(u1, v2, a, p1);
    calculaPontoBezier(u2, v2, a, p2);
    calculaPontoBezier(u2, v1, a, p3);
//(...)
```

Por último, foi necessário efetuar a escrita no ficheiro *.3d*. Tendo em conta que os 4 pontos representam um quadrado, e consequentemente 2 triângulos, estes são escritos no ficheiro da seguinte forma:

```
   /*   0------- 1
        |        |
        |        |
        3--------2   */

    //Triangulo 012:
    escreveFicheiro << p0[0] << " " << p0[1] << " " << p0[2] <<
        endl;
    escreveFicheiro << p1[0] << " " << p1[1] << " " << p1[2] <<
        endl;
    escreveFicheiro << p2[0] << " " << p2[1] << " " << p2[2] <<
        endl;

    //Triangulo 023:
    escreveFicheiro << p0[0] << " " << p0[1] << " " << p0[2] <<
        endl;
    escreveFicheiro << p2[0] << " " << p2[1] << " " << p2[2] <<
        endl;
    escreveFicheiro << p3[0] << " " << p3[1] << " " << p3[2] <<
        endl;
//(...)
```

3.1.2 Implementação da fórmula matemática

De modo a implementar no nosso projeto as curvas de Bezier, foi necessário recorrer à sua fórmula matemática, abordada nas aulas teóricas, apresentada de seguida:

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

Analisando a equação, foi necessário proceder ao cálculo de cada um dos componentes desta, baseado em fórmulas dadas nas aulas:

```
void calculaPontoBezier(float u,float v,int patch,float r[3]) {
    /* vetor u: */
    float vetorU[1][4] = { {powf(u,3),powf(u,2),u,1} };

    /* vetor v: */
    float vetorV[4][1] = { {powf(v,3)},{powf(v,2)},{v},{1} };

    /* matriz M: */
    float matBezier[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} };

    /* matriz M_T (transposta) */
    float matBezierTrans[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} };
//(...)
```

De seguida, é criada a matriz corresponde ao patch, com os 16 pontos:

```
//(...)
float pX[4][4]; float pY[4][4]; float pZ[4][4];

  for (int i = 0; i < 4; i++) {
     for (int j = 0; j < 4; j++) {
        pX[i][j] = dados.pontos.at(dados.indices.at(patch).at(i
            * 4 + j)).x;
        pY[i][j] = dados.pontos.at(dados.indices.at(patch).at(i
            * 4 + j)).y;
        pZ[i][j] = dados.pontos.at(dados.indices.at(patch).at(i
            * 4 + j)).z;
            }
        }
//(...)
```

Por fim, é necessário iniciar as diversas multiplicações de matrizes. Estando já estas previamente inicializadas, o processo pode ser resumido nos "excertos"de pseudo-código apresentados de seguida.

Acrescenta-se, em forma de nota, que as funções de responsáveis pela multiplicação (propriamente dita) de matrizes são uma implementação fiel dos processos algébricos, estando omitidas deste relatório por essa razão.

Então, seguindo (1), tem-se a seguinte sequência de multiplicações:

- É aplicada a operação $res = vetorU * matBezier$:

```
//(...)
    // (1x4)(4x4) = 1x4
    multMatrix(vetorU, matBezier, res);
```

- É aplicada a operação $resX = res * pX$ para a coordenada X, sendo análogo para as coordendas Y e Z:

```
     // (1x4)(4x4) = 1x4
     multMatrix(res, pX, resX);
     multMatrix(res, pY, resY);
     multMatrix(res, pZ, resZ);
```

- É aplicada a operação $tX = resX * matBezierTrans$ para a coordenada X, sendo análogo para as coordendas Y e Z:

```
    // (1x4)(4x4) = 1x4
     multMatrix(resX, matBezierTrans, tX);
     multMatrix(resY, matBezierTrans, tY);
     multMatrix(resZ, matBezierTrans, tZ);
```

- Por fim, é aplicada a operação $x = tX * vetorV$ para a coordenada X, sendo análogo para as coordendas Y e Z, obtendo assim as 3 coordenadas do ponto a desenhar:

```
     // (1x4)(4x1) = 1x1
     multMatrix2(tX, vetorV, &x);
     multMatrix2(tY, vetorV, &y);
     multMatrix2(tZ, vetorV, &z);
```

- Para finalizar, é necessário colocar os pontos em questão no vetor responsável por dar *return*:

```
//(...)
     r[0] = x; r[1] = y; r[2] = z;
//(...)
```

3.2 Código de teste

De modo a poder testar esta nova funcionalidade, o método de invocação terá de seguir o formato explícito no seguinte comando:

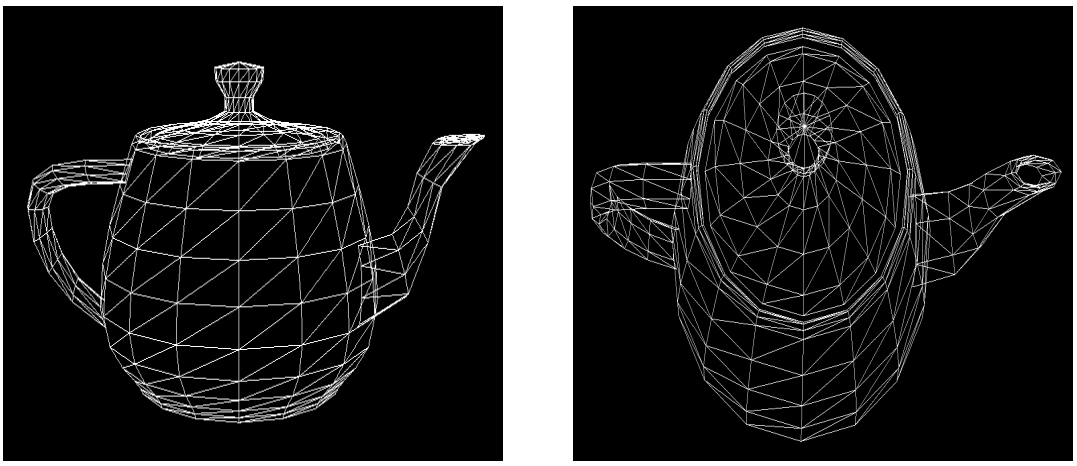
```
./gerador bezier <nomepatch>.patch <tesselacao> <output>.3d
```

4 Execução

4.1 *Teapot* - Bezier Patches

Conforme pretendido pelo enunciado, foi gerada a representação gráfica do *teapot* sendo utilizado, para a sua construção, o ficheiro patch disponibilizado pela equipa docente.

O resultado é o seguinte:



(a) Vista lateral (b) Vista de cima

Figura 1: *Teapot*

4.2 Sistema Solar

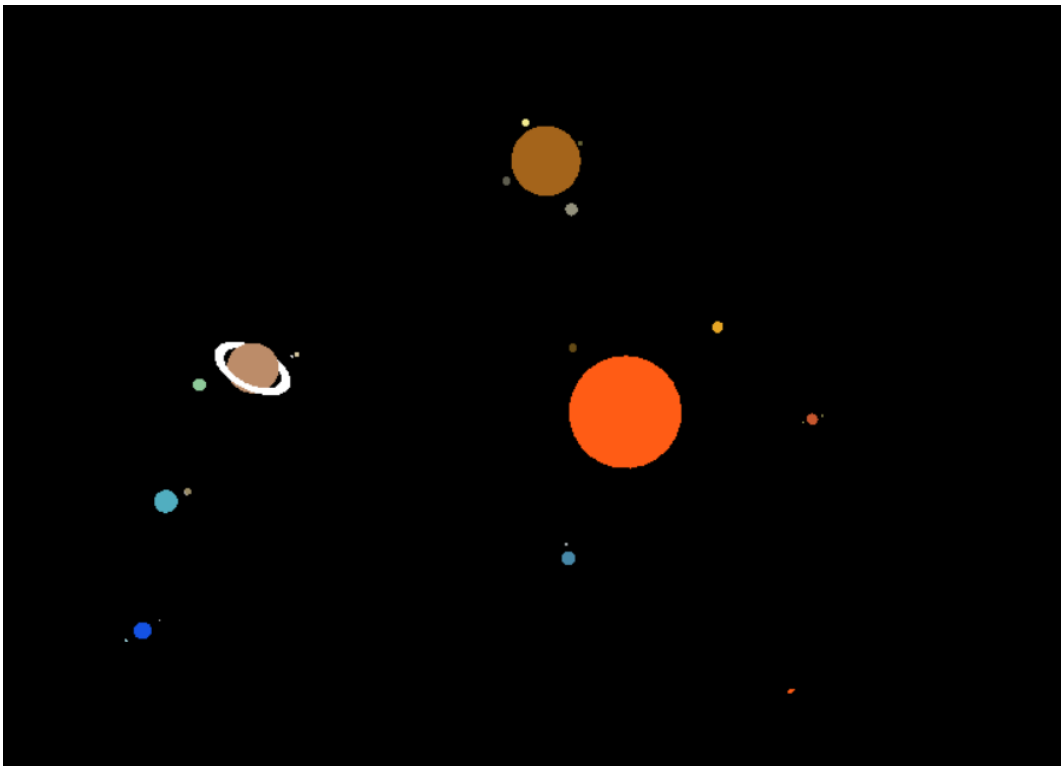


Figura 2: Sistema Solar sem trajetória gráfica desenhada

Premindo a tecla **t**, o utilizador tem a opção de visualizar as trajetórias gráficas dos vários elementos do Sistema Solar.

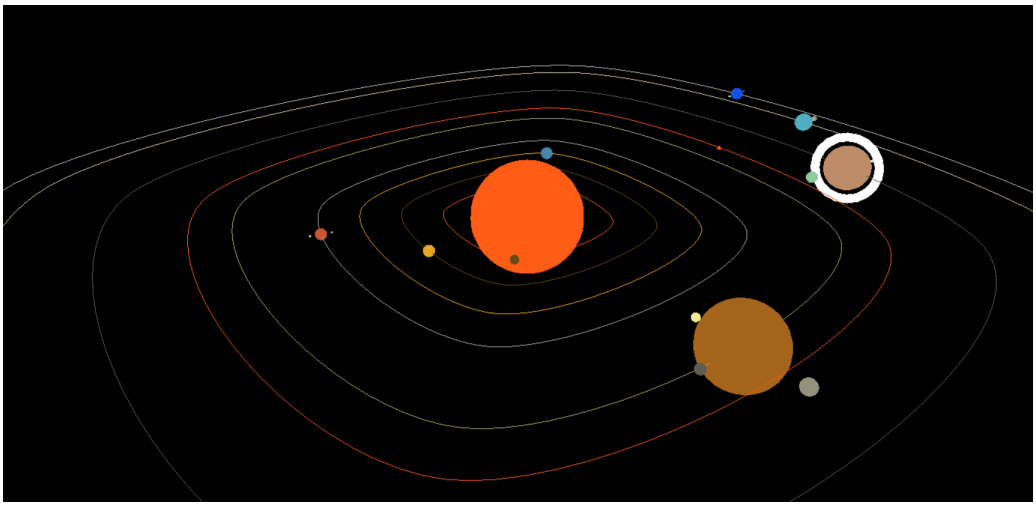


Figura 3: Sistema Solar com trajetórias representadas - 1

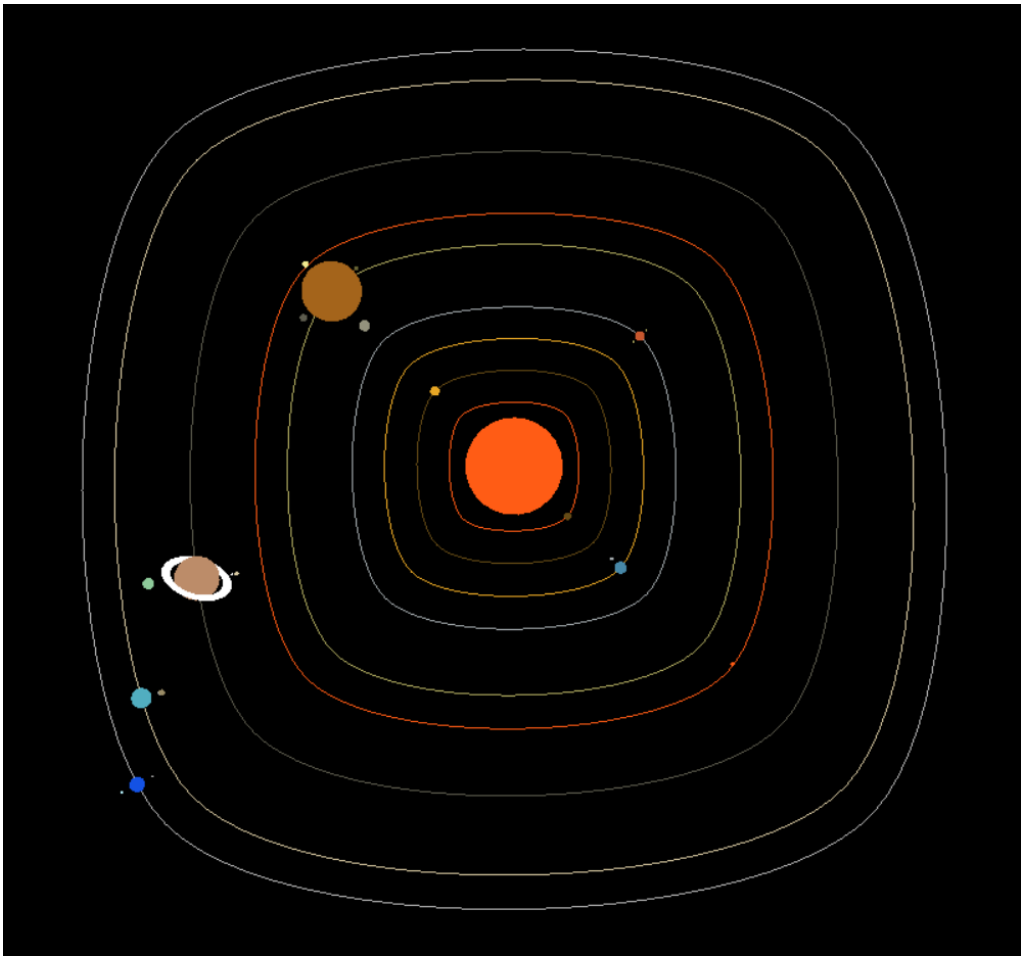


Figura 4: Sistema Solar com trajetórias representadas - 2

5 Conclusão

Quanto à parte pedagógica, concluímos que este projeto foi muito enriquecedor para o nosso coletivo, pois este trabalho permitiu que melhorássemos as nossas competências a nível prático relacionadas com a unidade curricular de Computação Gráfica.

Esta fase do trabalho permitiu a que chegássemos a várias conclusões distintas, pois lidámos com vários conceitos. Por um lado, foi possível obter uma prova clara da eficiência e utilidade dos VBOs. Por outro lado, toda a matemática e linha de pensamento por detrás da construção das várias curvas, para as animações, ficou bem assimilado, não tendo o grupo agora dificuldade em lidar com esta nova forma mais complexa e diversificada de representar modelos.

Consideramos, no entanto, que todos os objetivos delineados para esta fase foram cumpridos, traduzindo-se num maior aprofundamento na área da Computação Gráfica.

Em suma, o esforço foi grande com o intuito de garantir boas soluções para o enunciado proposto deixando, assim, uma janela aberta e curiosa para a quarta e última fase.