Universidade do Minho

Mestrado Integrado em Engenharia Informática

Computação Gráfica

Fase 3 - Curves, Cubic Surfaces and VBOs

Grupo 32

César Henriques - A64321

João Gomes - A74033

Rafael Magalhães - A75377

Tiago Fraga - A74092

2 de Maio de 2017

Conteúdo

1 Introdução

Nesta terceira fase do trabalho prático da unidade curricular de Computação Gráfica, foi nos pedido para alterar o código de modo a que o nosso desenho do sistema solar fosse realizado com *VBO's*. Foi nos pedido ainda que o nosso sistema solar fosse dinâmico e para tal utilizamos a curva de *Catmull-Rom's* de modo a definir as trajectorias dos planetas. Para alem disso, o gerador deverá ser capaz de, utilizando curvas de bezier, ler um ficheiro com os indices e pontos de controlos dos patches.

2 Gerador

Vamos começar pelas alterações feitas no gerador. De forma a ser possível ler o ficheiro patch com as informações para desenhar as várias superfícies de Bezier, foram adicionadas duas funções.

Trabalhamos apenas com superfícies de ordem (3,3), sendo então necessários **(3+1) x (3+1) = 16** pontos de controlo por superfície.

A função *generateModelBezier* trata de fazer o parsing do ficheiro patch e de preencher o vector **patches** e **control_points**. Após o parsing, a função contém uma panóplia de for loops, aninhados, que funcionam da seguinte forma:

- O ciclo *for* mais exterior, vai tratar de percorrer todos os patches;
- O segundo ciclo, para cada patch, vai preencher um vector **patch_cpoints** que vai conter todos os pontos de controlo relativo ao patch a gerar;
- Os próximos dois ciclos têm o propósito de gerar, a cada iteração mais interior, 6 pontos de Bezier de forma a que sejam formados triângulos. No total, o ciclo é percorrido de maneira a gerar uma superfície de acordo com a tecelagem pretendida;

```
int generateModelBezier(const char* inputFile, int tessellation
, const char* outputFile) {
...
  //A variavel tess representa o incremento entre dois pontos
de acordo com a tecelagem pretendida
  for(int i = 0 ; i < tessellation + 1; i++){
    for (int j = 0; j < tessellation + 1; j++) {
      float u = i*tess;
      float v = j * tess;
      Point generated_bezier_point;
      generated_bezier_point = bezierMatrixFormula(
patch_cpoints,u,v);
      bezier_points.push_back(generated_bezier_point);
      generated_bezier_point = bezierMatrixFormula(
patch_cpoints, u+tess, v);
                ...
    }
  }
```

Listing 1: Transformations

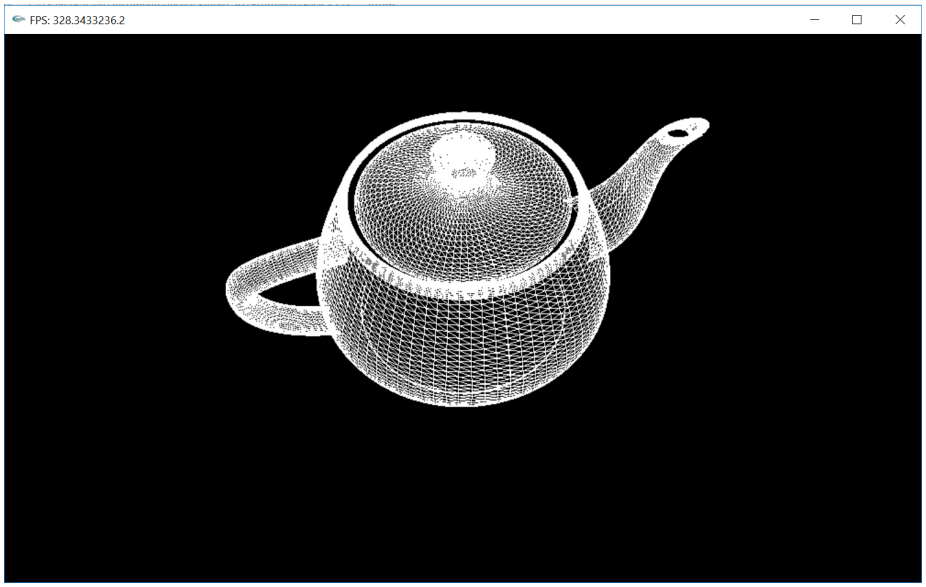
Falta então referir a segunda função, mencionada no primeiro parágrafo desta secção. A função *bezierMatrixFormula* trata de interpolar um ponto. Para efetuar este calculo, é utilizada a formula matricial que se encontra no formulário facultado na plataforma Blackboard, tendo em atenção que se devem utilizar as 3 coordenadas X, Y e Z.

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

Após um ponto ser calculado, ainda na função *generateModelBezier* é inserido num vector **bezier_points** que contem todos os pontos gerados. No fim, esse vector é passado como parâmetro para a função que trata de criar um ficheiro pronto a ser lido pelo motor.
Usando o seguinte comando é então possivel gerar modelos a partir de ficheiros patch:

```
$Generator.exe bezier <patch_file> <tesselation> <output_file>
```

Para o ficheiro teapot.patch é gerado o seguinte modelo:



Ainda nas alterações feitas no programa Gerador, foi adicionada uma funcionalidade que torna o gerador capaz de criar, numa circunferência de raio definido como argumento ao programa, pontos de controlo para uma curva gerada a partir do algoritmo Catmull-Rom. Tem também a particularidade de ser possível alterar o aspeto da circunferência de forma a se tornar numa elipse. Para utilizar, basta usar o seguinte comando:

```
$Generator.exe curve <raio> <tamanho_x> <tamanho_z> <output_file>
```

Os argumentos **tamanho_x** e **tamanho_z** são um valor entre [0..1] de maneira a que a circunferência adote a forma desejada. Exemplificando, com os argumentos raio=1.5, tamanho_x=1, tamanho_z=1 e output_file=points.xml é criado o ficheiro points.xml com o seguinte conteúdo:

```
<point x="1.500000" y="0.000000" z="-0.000000"/>
<point x="1.060660" y="0.000000" z="-1.060660"/>
<point x="-0.000000" y="0.000000" z="-1.500000"/>
<point x="-1.060660" y="0.000000" z="-1.060660"/>
<point x="-1.500000" y="0.000000" z="0.000000"/>
<point x="-1.060660" y="0.000000" z="1.060660"/>
<point x="-0.000000" y="0.000000" z="1.500000"/>
<point x="1.060660" y="0.000000" z="1.060661"/>
```

Agora, é só copiar e inserir no config.xml. É perceptível que esta alteração apenas visa a facilitar a criação do config file.

3 Motor

3.1 Processo de Leitura

No processo de parsing do XML, era agora necessário conseguir lidar com as alterações pretendidas.

- **Processamento dos Translates**

  Aparece agora o caso em que, no XML, o elemento *translate* pode conter um atributo time e, nesse caso, como filhos desse elemento devem ser inseridos quatro ou mais pontos de controlo para desenhar uma curva que o modelo terá de percorrer (no determinado tempo). Para resolver este problema, foi criada uma função *getTranslatePoints* que preenche um vector de floats, passado como parâmetro, com as coordenadas dos pontos de controlo.

```
int getTranslatePoints(vector<float> &cPoints, XMLElement*
    translateElem) {
    float x, y, z;
    ...
    while(pointElem != nullptr){
        x = 0; y = 0; z = 0;
        if (strcmp(pointElem->Name(), "point") == 0 ...
        ...
            while (attribute != nullptr) {
                if (strcmp((attribute->Name()), "X") == 0 ...
                    x = stof(attribute->Value());
                else if ...

                attribute = attribute->Next();
               }
        }
        //Assim garantimos que as coordenadas sao inseridas
    pela ordem correta.
        cPoints.push_back(x);
        cPoints.push_back(y);
        cPoints.push_back(z);
        pointElem = pointElem->NextSiblingElement();
    }
    //garante que a catmull tem no minimo quatro pontos de
    controlo.
    if ((cPoints.size() / 3) < 4) {
        cout << "Translate doesn't have enough control points.
    " << endl;
        return 0;
    }
    return 1;
}
```

Listing 2: Função de parsing de pontos de controlo

Este método é invocado na função *storeTranslate*, já criada na fase anterior, com a diferença que agora tem de satisfazer novas alterações ao construtor da Classe Translate.

```
int storeTranslate(Group &store_group, XMLElement * store_elem
    ) {
  int with_time = 0; //variavel discutida mais a frente
  float time = 0;
  float x = 0, y = 0, z = 0;
  vector<float> controlPoints; //vetor com pontos de controlo
    a ser preenchido
    ...
  while (attribute != nullptr) {
      if (strcmp((attribute->Name()), "X") == 0 ...
          x = stof(attribute->Value());
      else if ...
        ...
      else if (strcmp((attribute->Name()), "time") == 0...) {
      with_time = 1;
      time = stof(attribute->Value());
      if (!(getTranslatePoints(controlPoints, store_elem)))
        return 0;
    }
    attribute = attribute->Next();
  }
  Translate *translate = new Translate(x,y,z,controlPoints,
    time, with_time);
  store_group.transformations.push_back(translate);
  return 1;
}
```

Listing 3: Função de parsing de elementos Translate

A função *storeRotate* foi também alterada mas de forma menos significativa.

```
int storeRotate(Group &store_group, XMLElement * store_elem) {
  int with_time = 0;
  float time = 0, angle = 0, x = 0, y = 0, z = 0;
    ...

  while (attribute != nullptr) {
    if (strcmp((attribute->Name()), "X") == 0 ...
      x = stof(attribute->Value());
    else if ...
        ...
    else if (strcmp((attribute->Name()), "time") == 0 ... {
      time = stof(attribute->Value());
      with_time = 1;
    }
    attribute = attribute->Next();
  }
  Rotate *rotate = new Rotate(angle,x,y,z,time,with_time);
  store_group.transformations.push_back(rotate);
  return 1;
}
```

Listing 4: Função de parsing de elementos Rotate

- **PrimitiveMap**

  Como forma de não carregar a mesma primitiva mais do que uma vez, recorremos à estrutura Map para inserir de modo único as primitivas. Estas são mapeadas pelo nome do ficheiro respetivo. O map é declarado de forma global e é preenchido durante o parsing, na função loadModels.

- **Método *loadModels***

  De modo a que o motor fosse capaz de desenhar em *non-immediate-mode* tivemos de, em primeiro lugar, incluir a biblioteca Glew. Em segundo lugar, utilizar o metodo *glEnableClientState(GL_VERTEX_ARRAY)* na função main. Em termos de estruturas, alterar a classe Primitive que será discutida mais à frente, e em termos de parse alterar a função loadModels.

  Esta ultima, passa agora a incluir métodos para inicializar e preencher Vertex Buffer Objects que serão armazenados e posteriormente utilizados.

```
int loadModels(Group &group, vector<string>& list) {
  ...
    for (auto const& fileName : list) {
      //Se a primitiva ainda nao tiver sido inserida
        if (PrimitiveMap.find(fileName) == PrimitiveMap.end())
    {
            vector<float> vertex_vec;
            ...
            //preenche o vertex_vec com as coordenadas dos
    pontos do modelo
            ...
            //inicializa e preenche a VBO respetiva
            GLuint buffer;
            glGenBuffers(1, &buffer);
            glBindBuffer(GL_ARRAY_BUFFER, buffer);
            glBufferData(GL_ARRAY_BUFFER, sizeof(float) *
    vertex_vec.size() * 3, &(vertex_vec[0]), GL_STATIC_DRAW);

            //cria o objeto Primitive, onde guarda a VBO
            Primitive primitive(vertex_vec.size(), buffer);

            //insere a primitiva no Map
            PrimitiveMap.insert(pair<string, Primitive>(
    fileName, primitive));

            cout << "Loaded Primitive: " << fileName << endl;
        }
        group.models.push_back(fileName);
    }
    return 1;
}
```

Listing 5: Load

3.2 Classes

- **Classe *Primitive* :**

  Nesta classe adicionamos as variaveis *GLuint buffer* que guarda o VBO e ainda o *int* com o numero de vertices no buffer.

```
class Primitive {
public:
  // Numero de Vertices
    int vertexNumber;
    // buffer relativo as VBO's
    GLuint buffer;
public:

    .
    .
    .
}
```

Listing 6: Nova Classe Primitive

- **Alterações às classes de transformações :**

  As classes Translate e Rotate foram alteradas de forma a conseguirem lidar com a dependência (ou não) do tempo decorrido desde o glutInit(). Para tal, a classe Transformation recebeu mais um **float time** onde é guardado o tempo.

  Ambas as classes receberam uma variável **with_time** que serve como flag, com o propósito de os construtores saberem como lidar nos diferentes casos.
  No caso do Translate, a classe tem também um vector com os pontos de controlo necessários para a Catmul-Rom. O step é quanto o grupo se deve mover num milissegundo. Se multiplicarmos o step pelo *GLUT_ELAPSED_TIME* obtemos o ponto onde nos encontramos na curva. A derivada (tangente da curva no ponto) foi ignorada neste caso, porque não queremos que o modelo mantenha a orientação da curva.

```
class Translate : public Transformation {
public:
    int with_time;
    float step;
    vector<float> controlPoints;
public:
  ...
Translate::Translate(float x, float y, float z, vector<float>
    points, float time, int with_time){
    Translate::with_time = with_time;
    if (with_time == 1) {
        Translate::controlPoints = points;
        Translate::time = time;
        Translate::step = 1 / (time * 1000);
  }
    ...
}
void Translate::transform() {
    if(with_time == 0)
        glTranslatef(x, y, z);
    else {
```

```
        float res[3];
22      float deriv[3];
        //caso showTrajectories seja ativo, faz o render das
    curvas
24      if(showTrajectories)
          renderCatmullRomCurve(controlPoints);
26      elapsed_time = glutGet(GLUT_ELAPSED_TIME);
        getGlobalCatmullRomPoint(elapsed_time*step, controlPoints,
    res, deriv);
28      glTranslatef(res[0], res[1], res[2]);
        }
30  }
```

Listing 7: Translate

No Rotate, basta agora salientar que, o construtor varia na forma como guarda o ângulo e na forma como executa o método transform() dependendo do valor do with_time.

```
    class Rotate : public Transformation {
2   public:
        int with_time;
4       ...
    Rotate::Rotate(float angle, float x, float y, float z, float
        time, int with_time) {
6       Rotate::with_time = with_time;
        Rotate::time = time;
8       if (with_time == 0)
            Rotate::angle = angle;
10      else
            Rotate::angle = 360 / (time*1000);
12
        ...
14  }
    void Rotate::transform() {
16      if (with_time == 0)
            glRotatef(angle, x, y, z);
18      else{
            elapsed_time = glutGet(GLUT_ELAPSED_TIME);
20          glRotatef(elapsed_time*angle, x, y, z);
        }
22  }
```

Listing 8: Rotate

- **Funções para a realização da curva de *Catmull-Rom***

  Como o problema do calculo dos pontos foi discutido e resolvido nas aulas práticas, não vamos dedicar, para este relatório, muito texto às funções que tratam dos cálculos.

4 Ficheiro XML

Nesta secção apresentamos as alterações feitas no ficheiro *XML*, de modo a que o nosso sistema solar fosse dinamico. O esquema do nosso sistema solar mantém-se: uma hierarquia onde o Sol é o topo e todos os outros planetas são criados em função dele. As escalas foram mantidas e adicionamos apenas novos *translates*, que simulam a translação dos planetas usando a noção de tempo e os pontos gerados pela funcionalidade extra implementada por nós no gerador, e *rotates* que simulam o movimento de rotação dos planetas, também recorrendo à noção de tempo. Seguimos então com uma pequena extração do atual config.xml.

```
<scene>
  <group>
    <translate x="-50.5"/>
    <!--Sol-->
    <scale x="5.5" y="5.5" z="5.5"/>
    <models>
      <model file="sphere.3d"/>
    </models>
          <!--Mercurio-->
    <group>
      <translate time="10.8797">
        <point x="3.500000" y="0.000000" z="-0.000000"/>
        <point x="2.474874" y="0.000000" z="-2.474874"/>
        <point x="-0.000000" y="0.000000" z="-3.500000"/>
        <point x="-2.474874" y="0.000000" z="-2.474873"/>
        <point x="-3.500000" y="0.000000" z="0.000000"/>
        <point x="-2.474874" y="0.000000" z="2.474873"/>
        <point x="-0.000001" y="0.000000" z="3.500000"/>
        <point x="2.474873" y="0.000000" z="2.474875"/>
      </translate>
      <scale x="0.15" y="0.15" z="0.15"/>
      <rotate time="0.785" x="0" y="1" z="0"/>
      <models>
        <model file="sphere.3d"/>
      </models>
    </group>
          ...
```

Listing 9: XML

5 Cometa

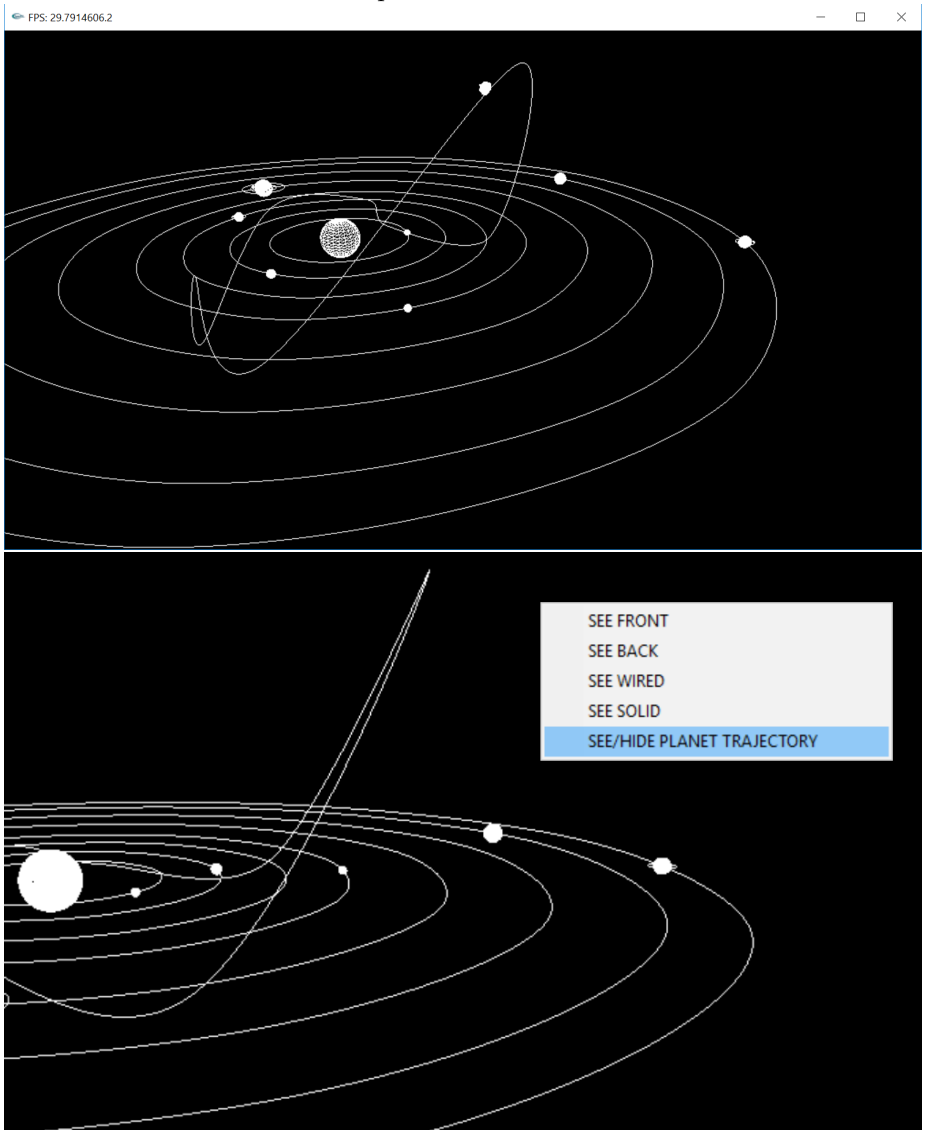
Nesta fase do trabalho prático além de realizarmos as orbitas dos planetas, criamos ainda um cometa que apresenta uma orbita um pouco diferente. O modelo para o cometa é o modelo gerado a partir do ficheiro patch do teapot.

```
<!-- Cometa -->
      <group>
        <translate time="80">
          <point x="12.000000" y="9.000000" z="-0.000000"/>
          <point x="5.303301" y="0.000000" z="-5.303301"/>
          <point x="-0.000001" y="0.000000" z="-3.000000"/>
          <point x="-5.303302" y="0.000000" z="-5.303300"/>
          <point x="-7.500000" y="0.000000" z="0.000000"/>
          <point x="-7.303302" y="-7.000000" z="5.303300"/>
          <point x="-0.000002" y="0.000000" z="7.500000"/>
          <point x="9.303298" y="0.000000" z="9.303303"/>
        </translate>
        <scale x="0.10" y="0.10" z="0.10"/>
        <rotate time="10" x="0.5" y="1"/>
        <models>
          <model file="teapot.3d"/>
        </models>
      </group>
```

Listing 10: XML

6 Demo

Por fim, incluimos uma print do nosso sistema solar com todas as alterações, com o cometa e mostramos também uma nova entrada no menu que permite mostrar ou esconder as orbitas dos planetas.



7 Conclusão

A elaboração desta terceira fase foi um bom modo de aprofundarmos os conhecimentos obtidos nas aulas ao nível do desenho de figuras com *VBO's*, bem como a implementação de curvas de *Catmull-Rom*.

As principais dificuldades encontradas foram sobretudo compreender o funcionamento das superficies de Bezier, mas mais complicado foi conseguir encontrar e corrigir os problemas de calculo encontrados após perceber como estas funcionavam.

Finalizando, o trabalho cumpre todos os requisitos pedidos, pretendendo para uma próxima fase dividir o código fonte em mais ficheiros de forma a manter tudo mais organizado.