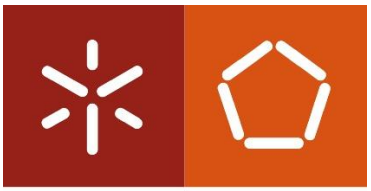


Universidade do Minho

Mestrado Integrado de Engenharia Informática

Computação Gráfica (2ºSemestre - 2019/20)

Relatório CG - TP3

Grupo Nº 33

A85813 António Alexandre Carvalho Lindo

A85400 Nuno Azevedo Alves da Cunha

A85919 Pedro Dias Parente

Braga

4 de Maio de 2020

Conteúdo

Lista de Figuras

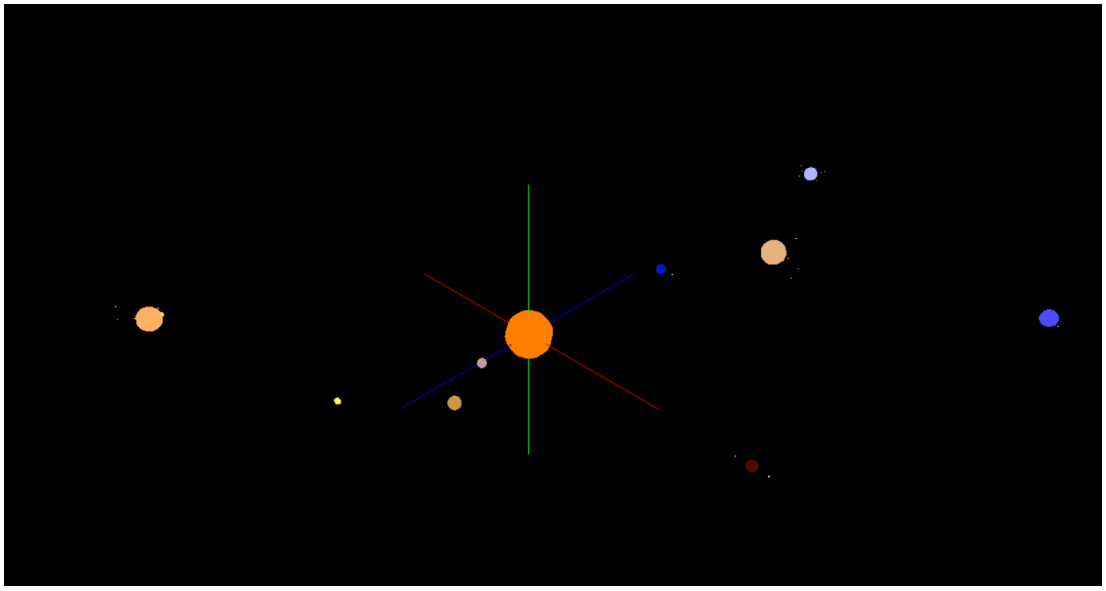
Capítulo 1

Introdução

Este relatório tem como objetivo descrever a implementação da 3ª fase do trabalho prático de CG.

A meta desta fase do Trabalho consistiu em implementar um sistema solar dinamico incluindo o movimento dos planetas e das suas luas. Foi necessário ter em conta a utilizção de VBOs, extenção do formato XML usado e a inclusão de um cometa formado usando bezier patches gerados pelo nosso programa gerador.

Figura 1.1: Sistema solar



Capítulo 2

Generator

2.1 Leitura do ficheiro

Para esta fase foi preciso fazer uma mudança ao Generator, pois tinhamos que implementar uma nova funcionalidade: Gerar ***bezierPatches*** a partir da leitura de um ficheiro.
Os argumentos para a nova função são: O nome do ficheiro lido, o nome do ficheiro onde se vai escrever os vértices dos triângulos, o tesselation level e os valores rgb.
Os ficheiros lidos têm que ter a seguinte estrutura:

- Uma primeira linha para indicar quantas patches vai ter a figura gerada;
- Tantas linhas quanto patches para indicar a ordem dos control points para cada patch (em cada linha/patch vai ter os control points referentes a esse patch e a ordem deles);
- Uma linha para indicar quantos control points existem;
- Uma linha para cada control point, com as coordenadas de cada um.

Criamos então uma função que lê este ficheiro e guarda a informação necessária.

Primeiro faz uma leitura da segunda linha para saber quantos control points há por patch para tornar a função mais dinâmica (em vez de assumir que são sempre 16).

De seguida começa a ler o ficheiro desde o princípio, guarda então o número de patches, guarda a ordem de todas as patches num vector pointsOrder, guarda quantos control points há ao todo e guarda as coordenadas de todos os control points em 3 vectors distintos: pointsX, pointsY e pointsZ.

2.2 Cálculo dos pontos que formam o Bezier Patch

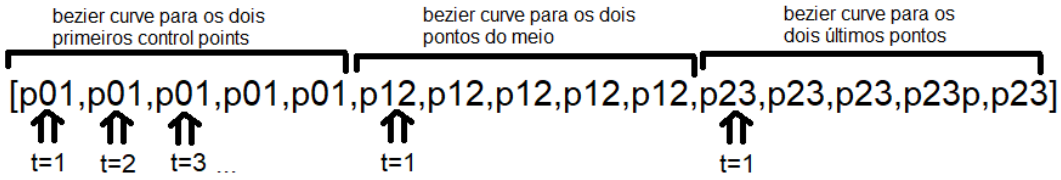
Nesta fase da função pegamos nas coordenadas dos nossos vectors pointsX, pointsY e pointsZ pela ordem que aparecem no vector pointsOrder e calculamos os pontos resultantes de usarmos a fórmula para bezier curves. (Se o nosso tesselation level for, por exemplo, 4 e tivermos 16 control points por patch calculamos a fórmula para t = {0,0.25,0.5,0.75,1}) e guardamos esses pontos em vectors com tamanho:

tamanho = (tesselation level + 1) x (raiz(número de control points por patch) - 1)

No nosso caso o tamanho seria então igual a 5 x 3 = 15

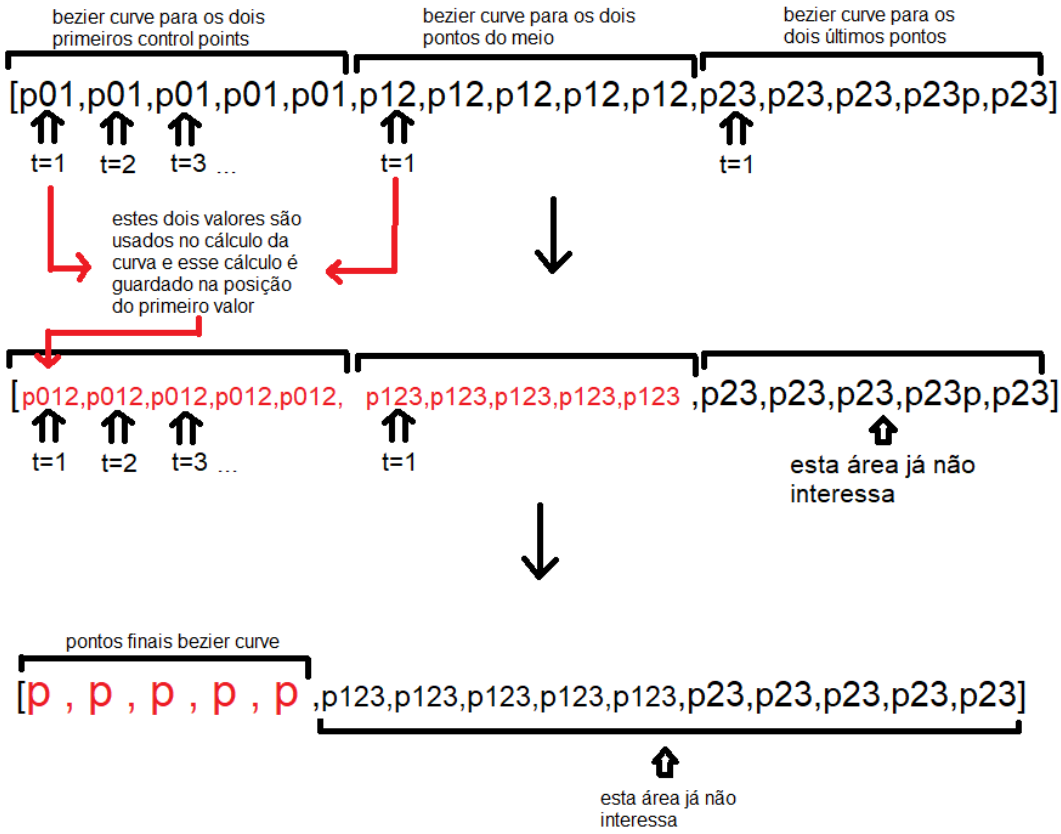
O seguinte esquema ajuda a perceber o mecanismo:

Figura 2.1: Bezier Curve - primeira iteração



A razão por qual queremos isto é para na seguinte iteração de calcularmos as curvas de bezier podermos aceder facilmente aos valores calculados na iteração anterior.

Figura 2.2: Bezier Curve - segunda e terceira iteração



A este ponto da função temos então os pontos calculados para as bezier curves individuais de cada

patch. Porém, para termos os pontos para os bezier patches, temos de pegar nos pontos das bezier curves e aplicar o mesmo procedimento que aplicamos aos control points iniciais.

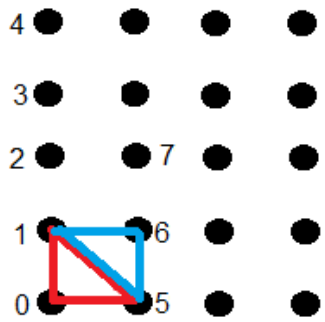
No fim disto tudo, guardamos os pontos finais, por ordem, em 3 vectors: o **bezPatchX**, **bezPatchY** e **bezPatchZ**.

2.3 Guardar os vértices dos triângulos num ficheiro

Tendo os pontos guardados por ordem em vectors torna então esta parte bastante simples, apenas temos de ter em conta quais são pontos a utilizar para cada triângulo.

O seguinte esquema irá facilitar a perceber o método que decidimos utilizar:

Figura 2.3: Bezier Patch - Guardar os vértices



Para o primeiro "quadrado"(2 triângulos), os vértices vão então ser guardados com os pontos 0,5 e 1 para o primeiro triângulo, e 5, 6 e 1 para o segundo. Este processo vai ser repetido para os pontos 1,2,6 e 7 e assim sucessivamente até percorrer a patch toda.

Capítulo 3

Engine

3.1 VBOs

Nesta 3º fase do trabalho prático, utilizamos VBOs para o desenho dos triângulos. Os VBOs são configurados e inicializados na função drawModel:

Figura 3.1: drawModel

```
void drawModel(Model* model) {
    vector<float> modelVertex = model->getVertices();
    glColor3f(model -> getR(), model -> getG(), model -> getB());
    glEnableClientState(GL_VERTEX_ARRAY);
    glBindBuffer(GL_ARRAY_BUFFER, modelsBuf[currentModel]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(float) * model->getVertices().size(), &modelVertex[0], GL_STATIC_DRAW);
    glVertexPointer(3, GL_FLOAT, 0, 0);
    glDrawArrays(GL_TRIANGLES, 0, model->getVertices().size()/3);
}
```

Como podemos observar, os vértices são colocados no vetor modelVertex, sendo depois iniciado o VBO. O vetor é depois copiado para o VBO na função glBufferData. Através da função glVertexPointer dizemos que um vértice é constituído por 3 floats e por fim desenhamos o modelo através de glDrawArrays. Através da observação dos resultados finais concluímos que houve uma enorme melhoria de performance através da utilização de VBOs.

3.2 Extenção do formato XML

Ao formato XML utilizado nas fases anteriores foram acrescentadas a rotação e translação animadas sendo estas mutuamente exclusivas com as suas versões estáticas.

Figura 3.2: Extensões ao XML

```
<translate time='5'>
    <point X='40' Y='00' Z='00'/>
    <point X='00' Y='00' Z='-40'/>
    <point X='-100' Y='00' Z='00'/>
    <point X='-150' Y='00' Z='150'/>
    <point X='00' Y='00' Z='100'/>
  </translate>
  <rotate time='2.5' X='0' Y='1' Z='0'/>
```

- A translação é descrita por uma curva de Catmull-rom que deverá ser completada no tempo indicado.
- A rotação deve descrever uma volta completa (360^{o}) no tempo indicado

De modo a acomodação da nova informação a receber foram criadas estruturas de dados para representar cada uma das transformações em memória

3.3 Animação

A animação foi tratado no programa na função **applyTranformation()**, segue-se o snippet relevante

Figura 3.3: drawModel

```
c = g->getCatmull();
if (c != NULL) {
    if (showOrbit) {
        drawCurve(c);
    }
    getGlobalCatmullRomPoint(time,c, pos, deriv);
    glTranslatef(pos[0], pos[1], pos[2]);
    currentModel++;
}

rotA = g->getRotationAnimation();
if (rotA != NULL) {
    glRotatef(getRotationAngle(rotA), rotA->getX(), rotA->getY(), rotA->getZ());
}
```

3.3.1 Translação

A translação é obtida de acordo com a posição calculada na curva de acordo com o tempo decorrido na função **getGlobalCatmullRomPoint()** e **getCatmullRomPoint()**

Figura 3.4: drawModel

```
c = g->getCatmull();
if (c != NULL) {
    if (showOrbit) {
        drawCurve(c);
    }
    getGlobalCatmullRomPoint(time,c, pos, deriv);
    glTranslatef(pos[0], pos[1], pos[2]);
    currentModel++;
}

rotA = g->getRotationAnimation();
if (rotA != NULL) {
    glRotatef(getRotationAngle(rotA), rotA->getX(), rotA->getY(), rotA->getZ());
}
```

Figura 3.5: drawModel

```
c = g->getCatmull();
if (c != NULL) {
    if (showOrbit) {
        drawCurve(c);
    }
    getGlobalCatmullRomPoint(time,c, pos, deriv);
    glTranslatef(pos[0], pos[1], pos[2]);
    currentModel++;
}

rotA = g->getRotationAnimation();
if (rotA != NULL) {
    glRotatef(getRotationAngle(rotA), rotA->getX(), rotA->getY(), rotA->getZ());
}
```