

Universidade do Minho

COMPUTAÇÃO GRÁFICA

MESTRADO INTEGRADO EM ENGENHARIA INFORMÁTICA

Fase 3

CURVAS, SUPERFÍCIES CÚBICAS E VBOs

GRUPO 19 - PL2

A85227 João Pedro Rodrigues Azevedo

A85729 Paulo Jorge da Silva Araújo

A83719 Pedro Filipe Costa Machado

A89983 Paulo Jorge Moreira Lima

Braga
Maio 2020

Conteúdo

1	Introdução	2
1.1	Contextualização	2
1.2	Objetivos para a fase 3	2
2	Atualizações ao gerador	3
2.1	Estrutura do ficheiro <i>Patch</i>	3
2.2	Geração dos vértices com os <i>patches</i> de Bezier	3
2.3	Execução do novo modelo	7
3	Atualizações ao Engine	8
3.1	Novas formas de configuração em xml	8
3.2	Curvas de <i>Catmull-rom</i> na definição das trajetórias	8
3.2.1	Condições de interpolação da curva	9
3.2.2	Cálculo do ponto de <i>Catmull-Rom</i>	9
3.3	Rotações com especificação de tempo	11
3.4	Desenho dos modelos com VBOs	12
3.5	Adaptações às estruturas de dados	13
4	Sistema Solar e órbitas dos planetas	14
5	Conclusão	15

Capítulo 1

Introdução

1.1 Contextualização

Este relatório é relativo ao trabalho prático da UC de **Computação Gráfica** e tem como objetivo final desenvolver um *engine* para a resolução de modelos **3D**, mostrando o seu potencial através da apresentação de um ficheiro de configuração **xml** capaz de reproduzir um sistema solar, na fase 4, com texturas, luzes e o movimento dos planetas pelas suas órbitas e em torno do seu próprio eixo.

1.2 Objetivos para a fase 3

Nesta terceira fase finalizamos o desenvolvimento do *core* do *engine*, na medida em que a estrutura do Sistema Solar com movimento dos planetas e inclusão de cometas pode ser apresentada, restando-nos, na fase 4 e última, adicionar texturas e luzes à nossa *scene*.

Posto isto, estabelecemos um conjunto de tarefas que indiciram tanto na atualização do **generator** como no **engine** para suportar os requisitos indicados no enunciado:

- Atualizações à aplicação **generator**:
 1. Efetuar o *parsing* de ficheiros de modelos baseados em *patches* e pontos de controlo com a formatação indicada no ficheiro de apoio, armazenando os dados lidos em estruturas apropriadas.
 2. Partindo dos dados da tarefa 1, efetuar a geração dos vértices (não repetidos)¹ utilizando as fórmulas das superfícies e curvas de *Bezier*.
- Atualizações à aplicação **engine**:
 1. Alterar o processamento dos vértices até agora utilizado, passando do modo imediato a VBOs (sem índices) para os modelos criados na fase 1.
 2. Tendo por base o ficheiro gerado pelo *patch* no *generator* apresentar os modelos utilizando, desta vez, VBOs com índices.
 3. Expandir o processamento do ficheiro de configuração *xml* de modo a suportar um novo tipo de *translates* e *rotates*, explicadas nos próximos capítulos.
 4. Adicionar os conhecimentos em curvas de *catmull-rom* na expansão da primitiva *translate* para o movimento dos modelos tendo por base um conjunto de pontos e o tempo total.

¹Para posterior utilização de VBOs com índices.

Capítulo 2

Atualizações ao gerador

2.1 Estrutura do ficheiro *Patch*

O ficheiro *patch* segue o seguinte formato adotado pelo documento de apoio fornecido:

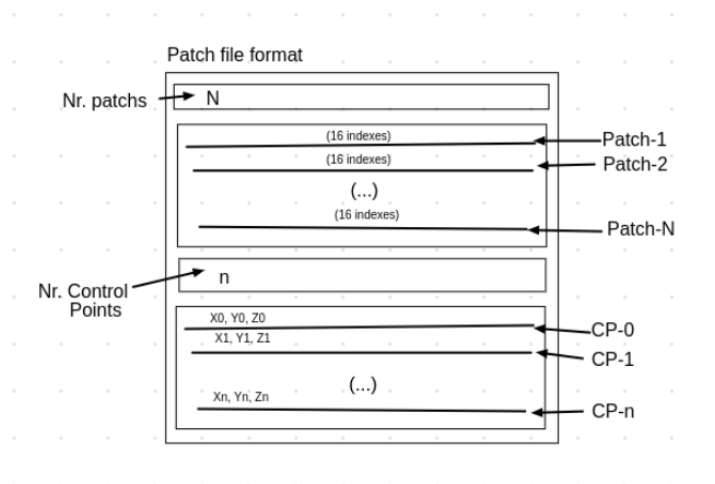


Figura 2.1: Formato do ficheiro *patch*.

As metodologias adotadas para *parsing* são irrelevantes para o relatório em si, mas destacamos as informações necessárias para passar à seguinte tarefa, ou seja, a geração do modelo em si. Temos então os seguintes dados:

- Um vetor com todos os índices (podendo depois saltar de 16 em 16 para percorrer os *patches*);
- Um vetor com todos os pontos de controlo;
- O nível de tesselação, fornecido como argumento ao programa e o próprio ficheiro *output*.

2.2 Geração dos vértices com os *patches* de Bezier

Nesta secção pretende-se apresentar o algoritmo adotado tendo em conta os dados obtidos na secção anterior na criação de superfícies de Bezier. Posto isto, temos um conjunto de índices em cada *patch* que mapeiam 16 **pontos de controlo** (4x4).

O nível de tesselação surge no sentido de "suavizar" as superfícies, pelo que as divide em (*tessellation* * *tessellation*) pontos intermédios. Um exemplo prático da estrutura real de uma superfície, que será posteriormente adotada para o *teapot* é a figura seguinte:

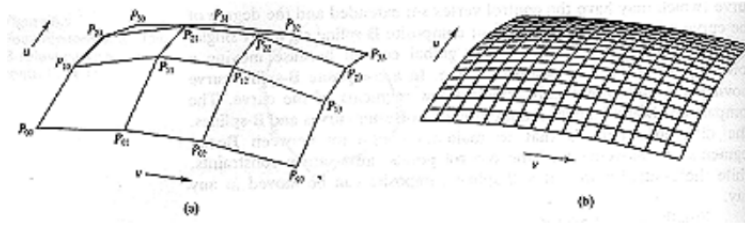


Figura 2.2: Superfície de Bezier.

Os pontos P_{ij} apresentados representam os pontos de controlo no *patch* original e as variáveis u e v tomam valores entre 0 e o valor de tesselação.

Começando a analisar, em forma de pseudocódigo, o algoritmo adotado, temos o seguinte fluxo:

```
algoritmo "gerar_modelo_bezier"
Inicio
    para <i = 0> até <total_indices t> [i+=16]

        patch_atual = i / 16

        Pij = buscar_vertices ( patch_atual, indices[], pontosControlo[] )

        (...)

    fim para

    (...)
Fim
```

Nesta primeira parte, obtemos os pontos de controlo do *patch* atual e armazenamos os mesmos numa matriz P_{ij} ¹.

Estes pontos de controlo, juntamente com o nível de tesselação, vão nos permitir calcular os diferentes pontos de bezier, variando os valores de u e v , para isso:

```
algoritmo "gerar_modelo_bezier"
Inicio
    para <i> até <total_indices t> [i+=16]

        //buscar Pij
        (...)

        para <v = 0> até <nivel_tesselacao>
            para <u = 0> até <nivel_tesselacao>

                //ambos entre 0 e 1
                delta_u = u / nivel_tesselacao
                delta_v = v / nivel_tesselacao
                //calcular ponto recorrendo a bezier e aos deltas
                PontoBezier = calcula_bezier( delta_u, delta_v, Pij )

                //escrever no ficheiro
                escrever PontoBezier
```

¹Na verdade, no código c++, guardamos separadamente em 3 vetores que compõem a matriz, P_{ij-X} , P_{ij-Y} e P_{ij-Z} .

De notar que a variação de u e $v \in [0..1]$ pois ambos os valores variam entre 0 e o nível de tesselação. Estes valores e o conjunto de pontos de controlo calculados anteriormente servirão para obter os pontos recorrendo às fórmulas de *Bezier*.

O cálculo propriamente dito será descrito mais à frente no algoritmo, aproveitando a continuidade do pseudocódigo apresentado para explicar o passo seguinte, a obtenção dos índices na formação dos triângulos dos modelos.

Sendo assim, na segunda parte do algoritmo passamos ao processamento dos índices:

algoritmo "gerar_modelo_bezier"

Início

```
//calcula dos pontos de bezier pelo método já apresentado acima
(...)

//os ciclos de cima considerao "<=" portanto nivel_tesselacao + 1
verticesInPatch = (nivel_tesselacao + 1) ^ 2

//calcular os indices
para <i> até <total_indices t> [i+=16]

    para <v = 0> até <nivel_tesselacao>
        para <u = 0> até <nivel_tesselacao>

            patch_atual = i / 16

            //Considerando um quadrado ABCD
            iA = (patch_atual*verticesInPatch) + u + (nivel_tesselacao+1) * (v)
            iB = iA + 1
            iC = (patch_atual*verticesInPatch) + u + (nivel_tesselacao+1) * (v+1)
            iD = iC + 1

            escrever A C B
            escrever B C D

        fim para
    fim para
```

Fim

Neste último excerto do algoritmo faz sentido apresentar o quadrado ABCD lá referido para poder associar as letras aos vértices do quadrado, pelo que consideramos o seguinte quadrado:

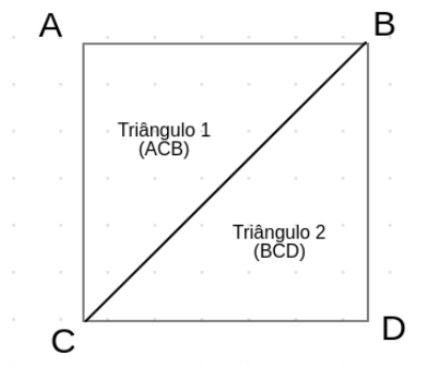


Figura 2.3: Triângulos considerados na escrita dos índices.

Por fim, resta-nos apresentar o cálculo do ponto propriamente dito, seguindo as fórmulas de *Bezier* disponibilizadas no formulário da UC.

O cálculo do ponto segundo *Bezier* segue a equação seguinte:

$$B(u, v) = \sum_{j=0}^3 \sum_{i=0}^3 B_i(u) * P_{ij} * B_j(v) \quad (2.1)$$

A análise desta fórmula permite-nos efetuar uma simplificação da mesma² através do produto das diferentes matrizes e vetores que a compõem, desde a matriz de *Bezier* aos vetores U e V que variam mediante o valor de **u** e **v** obtido através do nível de tesselação (intervalo entre 0 e 1 atual):

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

Figura 2.4: Cálculo do ponto por Bezier (retirado do formulário).

Resta-nos aplicar as sucessivas multiplicações no algoritmo que aplica esta fórmula, dada a variação de **u** e **v** e os pontos de controlo P_{ij} associados ao *patch* atual:

algoritmo "calcular_ponto_bezier"

Inicio

//aqui u e v representam o u ou v original / nivel de tesselacao

vetorU = [u*u*u, u*u, u, 1]

vetorV = [v*v*v, v*v, v, 1]

//Passo 1: M_BEZIER^T * vetorV

//como M_BEZIER é simétrica, transposta = matriz

M_v = mult_matriz_Vector(M_BEZIER, vetorV);

//Passo 2: P_{ij} * M_v

//no código, o cálculo está dividido em 3 passos

//para guardar os vetores X, Y e Z em separado

//assim, Pxyz_M_v[0,1,2] representam X, Y e Z

Pxyz_M_v[0,1,2] = mult_matriz_Vector(P_{ij}, M_v);

//Passo 3: M_BEZIER * Pxyz_M_v

M_Pxyz_M_v[0,1,2] = mult_matriz_Vector(M_BEZIER, Pxyz_M_v);

(...)

Fim

²O formulário da disciplina serviu de suporte aos passos intermédios omitidos.

Por fim, falta-nos calcular $\text{vetorU} * \text{M_Pxyz_M_v}$, ou seja, multiplicar o vetor U inicial pela matriz resultante de todas as multiplicações feitas até então, resultando nas coordenadas finais $\mathbf{B(u,v)}$ para o ponto de Bezier pretendido, sendo assim:

```
algoritmo "calcular_ponto_bezier"
Inicio

    //Passos 1, 2 e 3
    (...)

    //Passo 4 (e último)
    pontoResultante = [0, 0, 0]

    para <i = 0> até <4> [i = i + 1]

        //coordenada X
        pontoResultante[0] += vetorU[i] * M_Pxyz_M_v[0][i];

        //coordenada Y
        pontoResultante[1] += vetorU[i] * M_Pxyz_M_v[1][i];

        //coordenada Z
        pontoResultante[2] += vetorU[i] * M_Pxyz_M_v[2][i];

    fim para

Fim
```

Não foi referido até agora, mas a matriz de *Bezier* citada nos algoritmos está definida no próprio módulo C++ "bezier-patch.cpp/h" no programa *generator*.

2.3 Execução do novo modelo

O programa *generator* sofreu, deste modo, alterações a nível da geração de modelos, pelo que um novo fluxo de execução foi introduzido. Deste modo, para gerar o novo tipo de modelos devem ser respeitadas as seguintes normas:

1. Os ficheiros *.patch* devem seguir a estrutura anteriormente indicada e devem ser armazenados em **examples/Models.patch**;
2. A geração do modelo pode ser obtida com o seguinte comando:

```
//Comando geral
./generator bezier-patch <infile> <tessellation-level> <outfile>

//Exemplo:
./generator bezier-patch teapot.patch 20 teapot.3d
```

3. O ficheiro resultante será a concatenação de "outfile" com ".indexed" para que o *engine* possa diferenciar os ficheiros na inicialização de VBOs com ou sem índices.

Assim, os modelos resultantes serão armazenados em **Models.3d** e podem ter dois tipos de extensões '**outfile.3d**' ou '**outfile.3d.indexed**'.

Capítulo 3

Atualizações ao Engine

3.1 Novas formas de configuração em xml

O ficheiro de configuração em **xml** que temos vindo a utilizar dará suporte, nesta fase, a um novo tipo de translações, utilizando curvas (interpolação) de *catmull-rom*, e rotações de modelos para a simulação das órbitas dos planetas e a rotação dos mesmo sob o seu próprio eixo durante um certo período de tempo.

Analisando cada transformação geométrica até então referida, temos as seguintes atualizações:

- **Translações** com base em curvas cúbicas de *catmull-rom*, fornecendo:

1. O tempo total (**time**), em segundos, para percorrer a curva toda;
2. Um conjunto de pontos de controlo (**point**) P_0 a P_n com $n \geq 4$;

Partindo da estrutura seguinte:

```
<translate time="...">
  <point X="..." Y="..." Z="..." />
  <point X="..." Y="..." Z="..." />
  ...
  <point X="..." Y="..." Z="..." />
  <point X="..." Y="..." Z="..." />
</translate>
```

- **Rotações** com base no tempo de rotação (360 graus) segundo um (ou mais) eixo(s):

1. O tempo total (**time**), da mesma forma que nas translações;
2. O(s) eixo(s) para orientar a rotação do modelo.

```
<rotate time="..." X="..." Y="..." Z="..." />
```

3.2 Curvas de *Catmull-rom* na definição das trajetórias

Na definição das curvas de *Catmull-rom* a partir de um conjunto de pontos de controlo assume a estratégia adotada no guião da aula prática que aplicava este conceito, pelo que, na sua maioria, adotaram-se as funções fornecidas na altura.

No entanto, para o bem entendimento desta matéria e a sua correta aplicação, decidimos fazer uma explicação teórica dos cálculos feitos ao mesmo tempo que vamos apresentando o algoritmo.

De notar que todas as funções definidas para o cálculo das curvas se encontram no módulo "catmull-rom.cpp/h" definido no programa *engine*.

3.2.1 Condições de interpolação da curva

Uma curva de *Catmull-Rom* exige que sejam definidos, no mínimo, 4 pontos de controlo, P_0 , P_1 , P_2 e P_3 nos quais a curva seria desenhada entre P_1 e P_2 . Por outro lado, o ponto na curva (entre P_1 e P_2) é especificado através do valor de t , com $t \in [0.0, \dots, 1.0]$ e representando a porção da distância entre os dois pontos de controlo mais próximos (neste caso o ponto 1 e 2). A imagem seguinte¹ ilustra o que foi referido anteriormente:

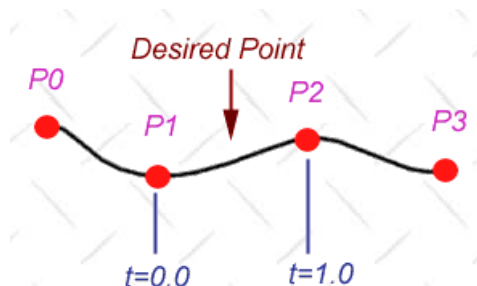


Figura 3.1: Representação de uma *spline* de *Catmull-Rom*

No nosso caso, o valor de t é calculado de modo a ter um número de intervalos suficiente para simular o tempo, em segundos, que levaria o modelo a percorrer a curva:

```
se Translacao_Com_Tempo então

    //tempoTranslacao em segundos
    t = (glutGet(GLUT_ELAPSED_TIME) / 1000) / tempoTranslacao

    //ponto resultante
    pos = [0.0, 0.0, 0.0]
    //vetor tangente à curva
    deriv = [0.0, 0.0, 0.0]

    pos, deriv = calcular_ponto_global_de_catmull(pontosControlo, t)

    glTranslatef(pos[0], pos[1], pos[2])

    //normalizar e aplicar matriz de rotação
    (...)

fim se
```

Onde, **GLUT_ELAPSED_TIME** representa o número de milissegundos desde que foi chamado o *glutInit(...)* ou a primeira chamada do *glutGet(GLUT_ELAPSED_TIME)*.

3.2.2 Cálculo do ponto de *Catmull-Rom*

Deste modo, passaremos à demonstração do algoritmo de cálculo dos pontos:

```
algoritmo "calcular_ponto_global_de_catmull"
Inicio

    POINT_COUNT = tamanho(pontosConstrolo)
```

¹Retirado de <https://www.mvps.org/directx/articles/catmull/>

```

//o valor real de t
tAux = t * POINT_COUNT
//que segmento
indice = floor(t)
//onde no segmento
tAux = tAux - indice

//que pontos usar da lista de pontos de controle
matriz indices[4]

indices[0] = (indice + POINT_COUNT - 1) % POINT_COUNT;
indices[1] = (indices[0] + 1) % POINT_COUNT;
indices[2] = (indices[1] + 1) % POINT_COUNT;
indices[3] = (indices[2] + 1) % POINT_COUNT;

//buscar os pontos dados os indices calculados anteriormente
matriz pontosSelecionados[4][3]

para <i=0> até 4 [i++]
    //cada ponto pertence à classe POINT_3D(x,y,z)
    p[i][0] = pontosControlo[indices[i]].x
    p[i][1] = pontosControlo[indices[i]].y
    p[i][2] = pontosControlo[indices[i]].z
fim para

pos, deriv = calcular_ponto_catmull(tAux, pontosSelecionados)

```

Fim

No pseudocódigo anterior estabelecemos, para um dado valor de **t** os pontos a usar relativamente ao segmento no qual se insere **t**. De seguida, utilizaremos esses pontos e o valor de **t** para obter o ponto resultante no segmento devido:

algoritmo "calcular_ponto_catmull"

Inicio

```

//definicao da matriz de catmull-rom
m = { {-0.5f, 1.5f, -1.5f, 0.5f}, ..., { 0.0f, 1.0f, 0.0f, 0.0f}}
//vetorT'
T = { t*t*t, t*t, t, 1}, TD = { 3*t*t, 2*t, 1, 0}
//contendo todos os 4 pontos dados como argumento
matriz P[3][4] = {...inicializar...}

matriz A[3][4]
//Compute vector A = M * P
A[0] = mult_matriz_Vector(m, P[0]) //x
A[1] = mult_matriz_Vector(m, P[1]) //y
A[2] = mult_matriz_Vector(m, P[2]) //z

//Compute pos[i] = T * A (Output)
para <i=0> até 4 [i++]
    pos[0] += T[i] * A[0][i];
    pos[1] += T[i] * A[1][i];
    pos[2] += T[i] * A[2][i];
fim para

```

```

//Compute deriv[i] = T' * A (Output)
//...
Fim

```

Assim, tendo o ponto de *Catmull-Rom* calculado para o segmento selecionado, resta-nos adaptar os vetores X_i , Y_i e Z_i dada a derivada calculada anteriormente. A figura seguinte, adaptada dos *slides* do guião prático 09 da UC, demonstra como os vetores se devem adaptar ao ponto atual da curva:

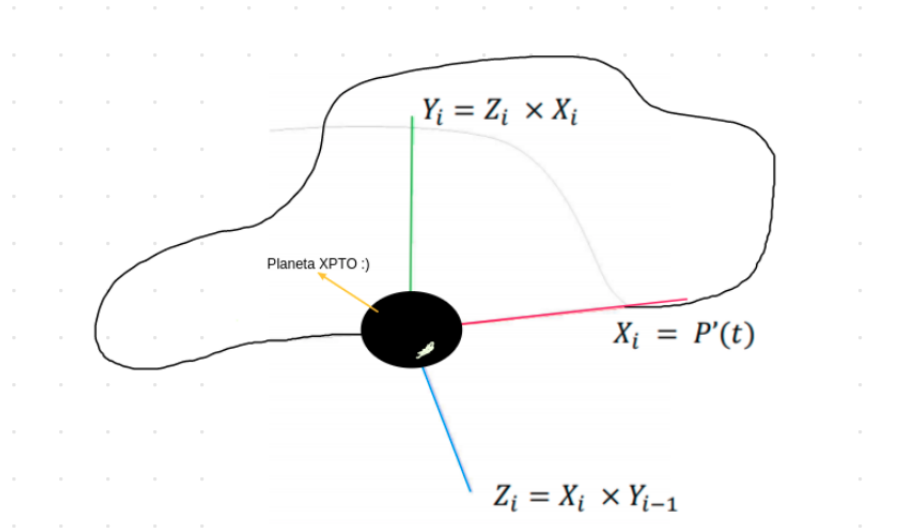


Figura 3.2: Vetores na trajetória dos planetas segundo curvas cúbicas de *Catmull-Rom*.

Sendo que, adaptar o vetor X_i implica utilizar o vetor **deriv** calculado. Já o vetor Z_i é calculado a partir do produto vetorial entre o X_i e Y_i^2 . Por fim, Y_i é obtido calculando o produto entre Z_i e **deriv**. A normalização dos vetores calculados trata-se do passo a seguir, permitindo-nos depois utilizar a função do *Glut* **glMultMatrix** e fornecendo-lhe a matriz de rotação determinada a partir dos vetores X_i , Y_i e Z_i .

```

//deriv == X
Z = produto(deriv, Y_anterior)
Y_anterior = produto(Z, deriv)

normalizar(deriv, Y_anterior, Z)

ROT_MATRIX = construirMatrizRotacao(deriv, Y_anterior, Z)

//aplicar a matriz de rotacao ao OpenGL
glMultMatrixf(ROT_MATRIX)

```

3.3 Rotações com especificação de tempo

A última alteração no ficheiro **xml** incide, como já foi referido, na definição da rotação dos modelos segundo um ou mais eixos, dado um tempo de rotação de 360 graus.

Esta adição é simples, na verdade, sendo apenas necessário utilizar, da mesma forma que as translações, o tempo a partir do *Glut*:

²Onde Y_i começa como $\{ 0.0f, 1.0f, 0.0f \}$ e vai sendo alterado sucessivamente na estrutura das transformações do modelo respetivo.

```
//tempoRotacao em segundos
angulo = ((glutGet(GLUT_ELAPSED_TIME) / 1000) * 360) / (tempoRotacao)

glRotatef(angulo, ROT.x, ROT.y, ROT.z);
```

Onde, multiplicar por 360 permite distribuir a variação do tempo em função dos segundos providenciados na rotação completa do modelo.

3.4 Desenho dos modelos com VBOs

A adaptação com VBOs trouxe dúvidas em saber se seria totalmente necessário a utilização dos mesmos com índices ou não nos modelos já gerados, pelo que achamos por bem manter as duas opções: sem índices para os modelos gerados na fase 1 e com índices no processamento do novo modelo *patch* introduzido nesta fase³. Assim, mais tarde, numa ótica de otimização, podemos adaptar os modelos da fase 1 para geração dos seus índices.

Deste modo, foi necessário armazenar, juntamente com cada modelo, para além do vetor de vértices, o vetor de índices e os *buffers* que serão ativados nos VBOs, ou seja:

```
class MODEL_INFO {
public:
    (...)
    vector<float>* vertices;
    vector<GLuint>* indexes;
    GLuint verticesBuffer[1];
    GLuint indexesBuffer[1];
    (...)
}
```

Portanto, para cada modelo, na função de desenhar os diferentes grupos da *scene*, inicializamos os VBOs antes de chamar o *renderScene*. Aqui exemplificaremos com VBOs com índices, utilizados nos modelos gerados a partir dos ficheiro de *patch*:

```
algoritmo "inicializar_vbo_indices_modelo"
Inicio
    vertices = modelo -> vertices
    indices  = modelo -> indices

    //Vertices
    glGenBuffers(1, vertices);
    glBindBuffer(GL_ARRAY_BUFFER, vertBuffer[0]);
    glBufferData(GL_ARRAY_BUFFER, tamanho(vertices) * tamanho(float),
                 vertices, GL_STATIC_DRAW);

    //Indexes
    glGenBuffers(1, model->indexesBuffer);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, model -> indexesBuffer[0]);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, tamanho(indices) * tamanho(GLuint),
                 indices, GL_STATIC_DRAW);

Fim
```

³Dada a facilidade da criação de índices no desenvolvimento do modelo com *patches* de Bezier

Já para desenhar os VBOs, na *renderScene*, para todos os modelos dos grupos de elementos carregados do *xml* fazemos:

```
glBindBuffer(GL_ARRAY_BUFFER, model.verticesBuffer[0]);
glVertexPointer(3, GL_FLOAT, 0, 0);

//se for VBO com indices
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, model.indexesBuffer[0]);
glDrawElements(GL_TRIANGLES, indexArraySize, GL_UNSIGNED_INT, NULL);

//se for VBO sem indices
glDrawArrays(GL_TRIANGLES, 0, count);
```

De notar que,

3.5 Adaptações às estruturas de dados

Capítulo 4

Sistema Solar e órbitas dos planetas

Para esta fase, a *scene* de demonstração pretendida passa pela configuração de um Sistema Solar dinâmico, incluindo um cometa¹ definido a partir de uma trajetória de *Catmull-Rom*.

¹Construído a partir do *patch* de Bezier fornecido.

Capítulo 5

Conclusão