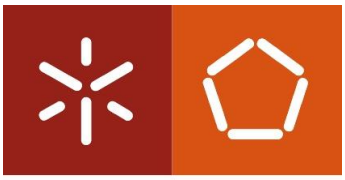

Universidade do Minho

Departamento de Informática

Trabalho Prático
Computação Gráfica

Catarina Machado (a81047) Gonçalo Faria (a86264)
João Vilaça (a82339)

18 de Abril de 2019

Conteúdo

Parte I
Primitivas Gráficas

Parte II
Transformações Geométricas

Parte III
Curvas, Superfícies Cúbicas e VBO

Com o objectivo de complementar o motor gráfico desenvolvido e, simultaneamente, criar as primitivas gráficas necessárias para a construção do sistema solar, nesta fase, serão introduzidas animações e superfícies cúbicas. Adicionalmente, como já na fase anterior foi iniciada a utilização de VBO's, nesta fase, este uso foi estendido para de melhor forma incorporar os novos requisitos.

1 Superfícies de Bazier

Ao módulo responsável pela geração de modelos foi introduzida uma nova primitiva. Esta primitiva, tal como especificada pelos docentes, permite a geração de superfícies de Bazier. Partindo de um ficheiro com pontos de controlo é possível construir todos os fragmentos que compõem a dada superfície bicúbica e traduzir esta em conjuntos de triângulos interpretáveis pela outro módulo deste projeto, o gerador.

1.1 Implementação

Partindo dos pontos de controlo $\mathbf{k}_{i,j}$, através da expressão em (1), determinamos os pontos $\mathbf{p}(u,v)$. O domínio desta função paramétrica é um subconjunto do conjunto real em que $v \in [0,1]$ e $u \in [0,1]$.

$$\mathbf{p}(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} B_i^3(u)\ B_j^3(v)\ \mathbf{k}_{i,j} \tag{1}$$

$$B_i^3(u) = \binom{3}{i}\ u^i(1-u)^{3-i} \tag{2}$$

$$\binom{3}{i} = \frac{3!}{i!(3-i)!} \tag{3}$$

Os valores de u e v são seleccionados consoante o grau de detalhe que se pretende. Este grau de detalhe é especificado no momento de modelação e é independente do ficheiro com os pontos de controlo. Tendo os pontos da superfície estes são organizados em fragmentos de 4 pontos. A cada fragmento é associado um rectângulo construído através de dois triângulos.

Após criado o modelo este é convertido para ficheiro no formado já especificado nas partes anteriores.

1.2 Demonstração

De forma a apresentar a primitiva gráfica desenvolvida, partido dos pontos de controlo disponibilizados pelos docentes, criamos o correspondente modelo. Na figura **1** é possível ver a representação visual do modelo criado.

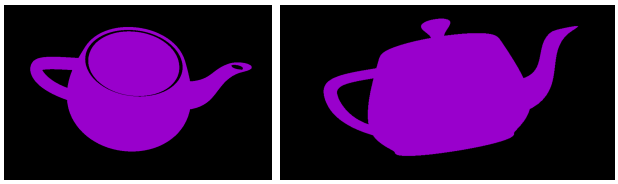

Figura 1: Modelo da cafeteira especificado pelos docentes.

Decidimos apenas modelar, para significativamente diminuir o número de vértices do modelo final, os triângulos correspondentes a um dos lados da superfície de Bazier.

2 Gerador

O módulo de geração dos gráficos, realizado nas fases anteriores deste projecto, foi construído tendo como base a mesma estrutura de dados usada na criação dos modelos. Esta decisão simplificou o código fonte e diminui significativamente o tempo de desenvolvimento. No entanto, esta estrutura não é capaz de suportar a interpretação de cenas que contêm animações. Após várias tentativas falhadas de a estender para suportar esta funcionalidade o grupo decidiu criar uma nova, denominada **Assembler**, particularmente concebida para responder aos requisitos do módulo gerador.

2.1 Assembler

A estrutura de dados desenvolvida, em suma, compreende a raiz de uma árvore, um apontador para um ramo da mesma, um vector de pontos e o identificador de um buffer da placa gráfica. A árvore em questão é composta por ramos de diferentes tipos, cada um corresponde às primitivas de modelação de cena em uso. Nas folhas da árvore encontram-se apenas os modelos.

```
typedef struct assembler{
    struct branch root;
    Branch current;
    vector< Point > * points;
    GLuint buffer;
} *Assembler;

typedef struct branch{
    BranchType type;
    BranchElement node;
} *Branch;
```

Os tipos de ramos definidos são **Animação**, **Transformação** e **Modelo**.

O tipo Animação generaliza as operações de translação periódica em curva e rotação periódica. O tipo Transformação generaliza diferentes tipos de transformações lineares, mais precisamente, escala, translação, rotação, assim como composições destas. O tipo **Modelo** que, como os vértices estão guardados directamente na estrutura **Assembler**, compreende apenas o intervalo de índices onde se encontram os vértices que constituem o modelo que o dado ramo representa.

2.1.1 Animação

```
typedef struct animation{
    vector< Branch > * descendents;
    int period;
    vector< Point > * controlPoints;
```

```
    AnimationType type;
} * Animation;
```

O tipo animação, de momento, apenas compreende as operações de rotação periódica e de translação em curvas catmull-rom.

A estrutura de dados que representa este tipo de ramo compreende o período em milissegundos da dada animação, um vetor de ramos descendentes, o tipo de animação, se é um translação em curva ou rotação, e um vetor com pontos de controlo.

No caso de translação periódica em curva, os pontos de controlo são os pontos da curva e o vetor normal do objecto em causa. Na rotação periódica, o vector apenas contém um elemento que é o eixo de rotação.

2.1.2 Transformação

O tipo Transformação é responsável pela representação de escala, translação e rotação, assim como composições destas.

A estrutura de dados que representa este tipo de ramo compreende uma matriz, que corresponde à representação matemática da transformação linear associada a esta, e um vetor de ramos descendentes.

```
typedef struct transformation{
    vector< Branch > * descendent;
    float **mat;
} * Transformation;
```

2.1.3 Modelo

O tipo Modelo que, é responsável pela representação das figuras geométricas geradas pelo módulo de modelação, é unicamente um intervalo de índices. Os índices correspondem a vértices na estrutura **Assembler**.

```
typedef struct model{
    long starti;/* including */
    long endi;/* not including */
} *Model;
```

2.1.4 Construção

A construção desta estrutura é feita partindo do ficheiro xml com a especificação de cena. Em essência o algoritmo é o mesmo que o apresentado na segunda parte deste projeto, apenas foram incluídas clausulas que lidam com os novos atributos temporais associados as animações e em alternativa a usar CoordinateFrame é usada a nova estrutura de dados Assembler.

2.1.5 Optimização de cena

Dado que as cenas que o motor gráfico se destinava a apresentar eram estáticas, todas as transformações que levavam à posição final dos vértices eram feitas na inicialização do programa. Desta vez, como isso não será possível, introduzimos funções que optimizam uma dada instância da estrutura de dados Assembler por forma a diminuir o número de cálculos necessários entre *frames*.

As principais optimizações resumem-se à combinação das transformações em série, à concatenação de modelos em paralelo e à aplicação direta, no vetor de pontos, das transformações que directamente antecedem modelos. Desta forma, cenas estáticas serão, tal como na fase anterior, processadas no momento de inicialização do programa, assim como, tanto quanto possível, as cenas dinâmicas, reduzindo o número de cálculos em tempo de execução ao mínimo.

2.1.6 Interpretação

Dada a natureza recursiva da árvore contida na estrutura Assembler, a interpretação desta foi feita através de um algoritmo recursivo como é possível verificar no algoritmo 2.

Algorithm 1 Interpretador de Assembler

```
1: procedure ASSEMBLERINTERPRET(Assembler referente, Point* outbuffer, int time)
2:     unmkModel( recInterpret(&(reference → root), reference → points, outbuffer, time) )
```

A principal ideia por detrás do algoritmo é de, primeiramente, efectuar a chamada recursiva quando esta é possível, e de seguida, propagar sucessivamente um ramo do tipo modelo, que representa os pontos da cena final já interpretados.

Algorithm 2 Interpretador Recursivo

```
1: procedure RECINTERPRET(Branch b, vector <Point >inpoints, Point* outpoints, int time)
2:     if b is Animation or Transformation then
3:         minv ← (inpoints → size())
4:         maxv ← −1
5:         for Branch descendent : (b → descendents) do
6:             Model tmp ← recInterpret(b, inpoints, outpoints, time)
7:             minv ← min(minv, mod → starti)
8:             maxv ← max(maxv, mod → endi)
9:             unmkModel(tmp)
10:        if b is Animation then
11:            applyAnimation(b → node, outpoints, minv, maxv, time)
12:        else
13:            applyTransformation(b → node, outpoints, minv, maxv)
14:        if minv <maxv then
15:            return mkModel(minv, maxv)
16:        else
17:            return mkModel(0, 0)
18:    else
19:        for i ← (mo → starti) ; i <(mo → endi); i++ do
20:            outpoints[i] ← (inpoints → at(i))
21:        return mkModel(mo → starti, mo → endi);
```

A função *applyTransformation*, multiplica os pontos, no intervalo de indices indicado, pela matriz representativa da transformação linear e a função *applyAnimation* aplica, a todos os pontos no intervalo de índices, a respetiva animação que um dado ramo representa.

O resultado da execução da interpretação é um *array* de vértices transformados correspondentes à cena final.

2.1.7 Rotação periódica

A animação correspondente à rotação periódica é trivialmente calculada com base nas funções já descritas nas fases anteriores. A única diferença é que o ângulo da rotação varia ao longo do tempo, sendo que a equação que define o seu valor encontra-se descrita na equação 4.

$$\theta(t) = 360 \cdot \frac{t \mod periodo}{periodo} \tag{4}$$

Onde *periodo* é uma constante especificada no momento de construção da animação e representa, efetivamente, o periodo a animação em questão.

2.1.8 Translação periódica em curva Catmull-Rom

Os pontos da curva Catmull-Rom necessários para a implementação da animação de translação foram calculados através da equação 5.

$$P(t) = \begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 0 & 0 \\ -\frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 1 & -\frac{5}{2} & 2 & -\frac{1}{2} \\ -\frac{1}{2} & \frac{3}{2} & -\frac{3}{2} & \frac{1}{2} \end{bmatrix} \cdot \begin{bmatrix} p_{n-1} \\ p_n \\ p_{n+1} \\ p_{n+2} \end{bmatrix} \tag{5}$$

Adicionalmente, para que a animação de translação tomasse um comportamento mais natural e altera-se a inclinação do modelo à medida que este progride na curva foi, posteriormente à aplicação da translação no objeto, aplicada a matriz descrita em 10 a todos os seus pontos. O valor de Y^0 é especificado à priori pois corresponde à normal do objeto, que por padrão será $(0, 1, 0)$.

$$\frac{dP(t)}{dt} = \begin{bmatrix} 0 & 1 & 2t & 3t^2 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 0 & 0 \\ -\frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 1 & -\frac{5}{2} & 2 & -\frac{1}{2} \\ -\frac{1}{2} & \frac{3}{2} & -\frac{3}{2} & \frac{1}{2} \end{bmatrix} \cdot \begin{bmatrix} p_{n-1} \\ p_n \\ p_{n+1} \\ p_{n+2} \end{bmatrix} \tag{6}$$

$$X^i = \frac{dP(t)}{dt} \tag{7}$$

$$Z^i = X \times Y^{i-1} \tag{8}$$

$$Y^i = Z^i \times X^i \tag{9}$$

$$M^i = \begin{bmatrix} X_1^i & Y_1^i & Z_1^i & 0 \\ X_2^i & Y_2^i & Z_2^i & 0 \\ X_3^i & Y_3^i & Z_3^i & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{10}$$

Por fim, para que a matriz em 10 apresente transformações de acordo com o esperado os vetores usados têm de se encontrar normalizados.

2.2 Objeto de Buffer de Vértices

Sendo que até esta parte do projeto as cenas geradas eram estáticas, após criado e carregado com dados, o objeto de buffer de vértices nunca era alterado. No entanto, dada a inclusão de animações é necessário um constante carregamento de vértices no VBO. Esta funcionalidade foi obtida através do uso da função **glMapBuffer**, que permite directamente manipular o buffer de vértices na placa gráfica.