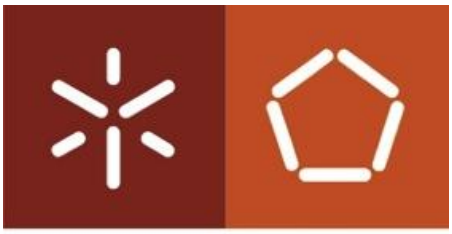


Universidade do Minho

Computação Gráfica

Relatório Projeto Prático – Parte III

Mestrado Integrado em Engenharia Informática

Ano Letivo 2016/2017

Docente: António José Borba Ramires Fernandes

Elementos do Grupo:

Daniel Teixeira Militão – A74557

Hugo Alves Carvalho – A74219

João Ismael Barros Dos Reis – A75372

Luís Miguel Da Cunha Lima – A74260

Índice

1. Introdução

O presente relatório descreve a terceira etapa do projeto prático da unidade curricular de Computação Gráfica.

Nesta etapa foi pedido que o generator criasse um novo tipo de modelo baseado em patches de Bezier, que fossem extendidos os elementos translate e rotate da engine, que todos os modelos fossem desenhados com VBOs e, por fim, que a demo scene seja um sistema solar dinâmico, incluindo um cometa, construido utilizando patches de Bezier e com a trajetória definida utilizando a curva Catmull-Rom.

Neste relatório vamos explicar, através de figuras e algoritmos em pseudocódigo, o nosso raciocínio para a realização daquilo que foi pedido.

2. Generator

Nesta fase do projeto é necessário que o *generator* seja capaz de criar um novo tipo de modelo baseado em *Bezier patches*.

O *generator* recebe como parâmetros o nome do ficheiro onde os pontos de controlo de *Bezier* estão definidos bem como o nível de *tessellation*.

O resultado será um ficheiro que contém a lista de triângulos para desenhar a superfície, tal como os ficheiros produzidos pelo *generator* na primeira fase.

Nesta secção vai ser explicado o processo de leitura do ficheiro *patch* e do cálculo dos pontos dos triângulos.

2.1. Leitura do ficheiro *patch*

Nesta secção iremos descrever o algoritmo de leitura dos ficheiros *patch*.

1. Abre o ficheiro que o utilizador pedi u;
    a. Se não obtiver sucesso, termina execução.
2. Lê a primeira linha, que indica o número de *patches* e guarda esse valor;
3. Alocar espaço necessário para armazenar os 16 índices de cada *patch*;
4. Ler os 16 índices de cada patch e armazená-los;
    a. Iterar linha a linha, cada linha correspondendo a um *patch*, armazenando os valores separados por *','*;
5. Ler o número de pontos de controlo e armazená-lo;
6. Alocar espaço necessário para armazenar as coordenadas de cada ponto de controlo;
7. Ler as coordenadas de cada ponto de controlo e armazená-las;
    a. Iterar linha a linha, cada linha correspondendo a um ponto de controlo, armazenando os valores separados por *','*;
8. Todos estes valores são armazenados em variáveis globais, para facilitar o seu acesso no futuro.

2.2. Criação da superfície de Bezier

Nesta secção iremos descrever o algoritmo usado para a criação de uma superfície de Bezier.

1. Leitura e armazenamento dos dados dos *patches* de Bezier (descrito na seção anterior);
2. Iterar sobre o número de *patches*:
    a. Iterar sobre o nível de *tessellation*:

b. Calcular o ponto v, dividindo o nível de *tessellation* usado atualmente neste ciclo por o nível de *tessellation* máximo;
    i. Iterar novamente sobre o nível de *tessellation*:
        1. Calcular o ponto u, dividindo o nível de *tessellation* usado atualmente neste ciclo por o nível de *tessellation* máximo;
        2. Calcular os pontos de Bezier o triângulo, utilizando o número do *patch* atual, *u* e *v* (descrito na próxima secção);
3. Depois de calculados todos os pontos, eles são armazenádos num ficheiro.
4. Libertação da memória alocada para o armazenamento dos *patches* de Bezier.

2.3. Cálculo dos pontos

Nesta secção iremos descrever o algoritmo usado para o cálculo de um ponto de Bezier.

1. Criação dos polinomiais de Bernstein, para u e para v, da seguinte maneira:

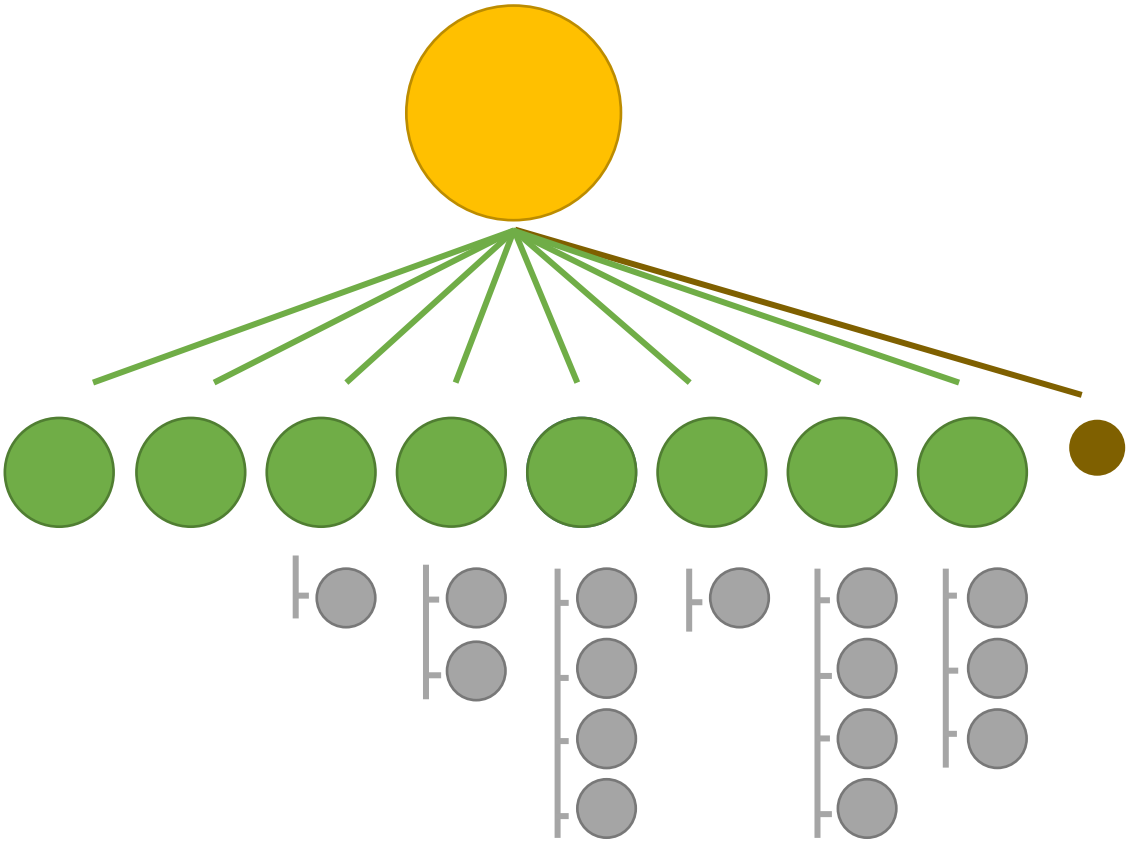
```
bpu[4] = { powf(1-u, 3), 3*u*powf(1-u, 2), 3*powf(u, 2)*(1-u), powf(u, 3) };
bpv[4] = { powf(1-v, 3), 3*v*powf(1-v, 2), 3*powf(v, 2)*(1-v), powf(v, 3) };
```

2. Iterar sobre o polinomial de Bernstein para u:
    a. Iterar sobre o polinomial de Bernstein para v:
        i. Calcular o índice do *patch* que iremos usar;
        ii. Calcular o índice do ponto de controlo que iremos usar, utilizando o índice do *patch* calculado;
        iii. Para cada uma das coordenadas do ponto (x, y, z) adicionar a multiplicação da respetiva coordenada do ponto de controlo pelos polinomiais de Bernstein de u e de v.

3. Engine

3.1. Estruturas de Dados

Derivado aos requisitos pedidos para a terceira fase deste projeto, foi necessário mudar a estrutura da árvore anteriormente criada - a rotação, translação e escalação de uma figura passam a ter classes distintas. De um modo geral, cada nodo da árvore contêm figuras com rotações, translações e escalações a si associadas, sendo as subárvores figuras dependentes. Tomando em conta o exemplo do sistema solar, está representado abaixo em árvore que lhe corresponderia:



Temos como primeiro nodo da árvore o sol, e como respetivos filhos estão os planetas que por sua vez contêm as luas como filhos. Existem também cometas que fazem translações relativas ao sol.

Tendo então a classe partilhada 'Coordinate' deixado de existir, foram criadas três novas:

```
class Scale {
public:
	Scale() :empty(true) {}
	bool empty;
	float x, y, z;
};
```

A classe `Scale` contêm três coordenadas `x`, `y` e `z` que têm como função representar o escalamento de uma determinada figura nessas componentes. O booleano `empty` será `true` se uma dada figura for escalada e usar-se-á na altura da representação das figuras.

```
class Rotation {
public:
	Rotation() :empty(true), angle(-1), time(-1) {}
	bool empty;
	float x, y, z, angle, time;
};
```

No que diz respeito à classe Rotação, esta contem também três componentes `x`, `y` e `z` bem como angulo e tempo. Só uma das duas últimas é preenchida - consoante com o que é especificado no ficheiro XML – e outra terá valor -1 de modo a auxiliar na representação. O booleano `empty` será `true` se uma dada figura tiver rotação.

```
class Translate {
public:
	Translate() :empty(true) {}
	bool empty;
	float time;
	float** points;
	int npointers;
	float** matrix;
};
```

A classe `Translate` é classe mais complexa pois é a que diz respeito à translação de uma determinada figura. Contêm o tempo, uma matriz `points` com os vários pontos de translação (cada linha da matriz contem um ponto `x`, `y` e `z`), e número de pontos presentes. Terá também mais uma matriz `matrix` que mais tarde terá as transformações necessárias para movimento.

```
class Figure {
public:
	Figure() : vboIndex(-1), triangles(0) {}
	string name;
	int vboIndex, triangles;
	Scale scale;
	Rotation rotate;
	Translate translate;

};
```

A classe `Figure` contêm a figura bem como a sua escala `scale`, rotação `rotate` e translação `translate`. Indica em que índice do VBO (`vboIndex`) se encontra e numero de triângulos existentes (`triangles`). O nome da figura será representado em `name`.

```
class Tree {
public:
	Figure figure;
	std::vector<Tree> subtrees;
};
```

Finalmente, a classe `Tree` que corresponde à árvore que terá como seu nodo uma `figure` e as respetivas subárvores.

3.2. Leitura e Representação

Tendo em vista o aumento de performance, foram implementados *VBOs*. A sua aplicação consiste essencialmente no processo de leitura e representação de imagens, como abaixo irá ser abordado.

3.2.1. Ficheiro XML

Antecipadamente à aprimoração da engine, foi criado um ficheiro XML com o objetivo de apresentar o sistema solar, com os respetivos planetas e satélites naturais, que, neste contexto serão todos tratados como luas, como apresentado na parte anterior. A sua organização passa por um conjunto de grupos e subgrupos – nesta situação o grupo principal trata-se do Sol, que contem vários grupos correspondentes aos seus planetas, que por si contêm outros grupos, as luas.

Ainda neste ficheiro é possível representar as órbitras, translações, rotações e escalas de cada um dos ficheiros a ser lidos. De modo a responder a um dos requisitos do enunciado era importante ainda representar a órbitra de um cometa ao qual foi representado por um teapot.

```
    <group a="Cometa">
        <translate time ="50" X="0" Y="0" Z="0">
            <point X="0.0" Y="-0.5" Z="2.0" />
            <point X="0.5" Y="0.5" Z="1.0" />
            <point X="1.0" Y="2.0" Z="-2.0" />
            <point X="1.0" Y="1.0" Z="-3.0" />
            <point X="1.0" Y="-5.0" Z="4.0" />
        </translate>
        <scale X="0.05" Y="0.05" Z="0.05" />
        <models>
            <model file="cometa.3d" />
        </models>
    </group>
</scene>
```

3.2.2. Leitura e Representação

De modo a explicar não só a implementação de *VBOs* mas também das alterações no sistema quanto à necessidade de representar translações ao longo do tempo, bem como rotações e escalações, passa-se a explicar o processo de leitura, usando a função `void getFigures()`. Resume-se o seu funcionamento a uma leitura recursiva de cada grupo, como é possível observar nesta explicação detalhada:

1. Abre ficheiro “scene.xml”;
    a. Se não obtiver sucesso, termina execução.
2. Procura primeiro elemento, “scene”;
    a. Se não existir, termina execução.
3. Usando esse elemento como parâmetro para `getGroup()`, este devolve a arvore que contêm a informação do ficheiro XML.

Ainda a função `getGroup()`:

1. Cria arvore `t`;
2. Cria elemento `child`, que corresponderá ao primeiro filho do elemento recebido;
3. Itera ao último filho do elemento recebido;
    a. Se se tratar de um `translate`, preenche `translate` do nodo `t`:
        i. Coloca no em `time` o tempo especificado;
        ii. Aloca espaço para as matrizes
        iii. Preenche matriz com pontos de translação com a lista dos pontos especificados;
        iv. Recorre à função `toMatrix()` que criará as transformações necessárias para movimento.

b. Se se tratar de um `rotate`, preenche `rotate` do nodo `t`:
    i. Preenche as componentes `x`, `y` e `z`;
    ii. Verifica se especifica tempo ou angulo, e guarda o valor pretendido.
c. Se se tratar de um `scale`, preenche `rotate` do nodo `t`:
    i. Preenche as componentes `x`, `y` e `z`;
d. Se se tratar de um `models`, itera-se sobre os existentes modelos:
    i. Verifica em lista de índices VBO se já existe a figura em VBO:
        - Se existir, preenche o indice vbo da figura do nodo t, assim como o número de triângulos indicados nessa lista;
        - Se não existir, recorre à função `loadFigure()` que carrega para VBO o array de pontos e devolve o indice e o numero de triângulos existente, que os adiciona à lista.
e. Se se tratar de um `group`, recursivamente usa-se a função `getGroup()` usando child como parâmetro, e coloca-se como nome no nodo arvore recebida o nome do tal `group`. Adiciona a árvore recebida como subárvore de `t`.

4. Retorna a árvore `t`.

3.2.3. Ciclos de Rendering

De modo a fazer *rendering* ao projeto final usa-se uma pesquisa em profundidade à árvore de modo a representar e efetuar as translações, rotações e escalas.

A função `renderScane()` chama a função `drawScene()`, que efectua a representação das figuras. De maneira mais detalhada, de modo a desenhar a cena usa-se a função `drawScene()`, que recebe como parâmetro uma arvore:

1. Efetua `glPushMatrix();`
2. Verifica se o a figura do nodo atual tem translação;
    a. Se tiver, efetua as translações necessária relativas ao tempo decorrido, recorrendo a `glTranslatef();`
3. Verifica se o a figura do nodo atual tem rotação;
    a. Se tiver, efetua recorrendo a `glRotatef()` relativamente ao angulo ou tempo;
4. Verifica se o a figura do nodo atual tem escala;
    a. Se tiver, efetua recorrendo a `glScalef();`

5. Verifica o índice VBO da figura e representa a figura indicando o índice e a quantidade máxima de pontos, que trata-se do numero de triângulos * 3, recorrendo a `glBindBuffer()` e `glDrawArrays()`
6. Itera pelas subárvores;
    a. Recursivamente usará `drawScene()` usando cada elemento como paramento;
7. Efetua `glPopMatrix()`

4. Conclusão

Com a realização desta terceira fase foi possível aperfeiçoar as técnicas de programação em C++, rever conceitos matemáticos e geométricos, bem como aprofundar a matéria lecionada nas aulas teóricas e práticas.

Foi importante para o grupo perceber a forma como iriam ser utilizados os *patches* de Bezier de forma a trabalhar e desenvolver o sistema solar dinâmico pedido no enunciado. Foi de igual forma revelante o compreender o funcionamento da curva Catmull-Rom para criar a órbitra do cometa desenvolvido.

A realização deste projeto permitiu-nos consolidar os conhecimentos adquiridos na unidade curricular, bem como identificar algumas lacunas que temos que corrigir futuramente.