Universidade do Minho
Escola de Engenharia
Mestrado Integrado em Engenharia Informática

Unidade Curricular de Computação Gráfica

Ano Lectivo de 2018/2019

Sistema Solar

Henrique Faria (A82200), João Marques (A81826),
José Pereira (A82880), Ricardo Petronilho (A81744)

Abril, 2019

CG

Data de Recepção	
Responsável	
Avaliação	
Observações	

Sistema Solar

Henrique Faria (A82200), João Marques (A81826),

José Pereira (A82880), Ricardo Petronilho (A81744)

Abril, 2019

Índice

Introdução

1.1. Contextualização

O relatório refere-se à terceira fase do projeto da Unidade Curricular de Computação Gráfica da Universidade do Minho, onde o objetivo é a construção de um sistema solar.

Na terceira fase, começou-se por modificar a aplicação do generator, adicionando uma opção que através de um ficheiro.patch com os pontos de controlo e o nível de tesselation, gera uma lista de triângulos para depois serem desenhados na aplicação engine.

De seguida, adaptou-se o parser do *XML*, pois para as curvas de Catmull-Rom, tem-se novos campos como o time e pontos nas transformações *translate* e *rotate*.

Na aplicação *engine* colocou-se os modelos a serem desenhados com *VBOs*, onde os índices e pontos são preenchidos a partir de ficheiros.

Por fim, ainda no *engine*, fez-se as funções necessárias para o desenho das curvas *Catmull-Rom*, que serão necessárias para as trajetórias do sistema solar, como por exemplo o cometa.

Generator

1.2. Superfície de Bezier

Para gerar e imprimir uma superfície de Bezier é necessário, primeiramente, ler o ficheiro **.patch**.

1.2.1. Leitura do ficheiro .patch

A leitura do ficheiro foi bastante facilitada devido ao formato do mesmo, criou-se a função **readPatchFile()**, desta forma o procedimento foi o seguinte:

1. ler o número de patches - N
2. ler N patches
3. ler o número de pontos – M
4. ler M pontos

À medida que se faz a leitura tanto dos patches como dos pontos, os mesmos são migrados para as respetivas estruturas em memória.

1.2.2. Gerar a superfície de Bezier

De seguida criou-se a função **getPointsOfBezier()** que, após a leitura do ficheiro .patch, segue o seguinte procedimento:

1. Percorrer cada patch e a respetiva tesselation associada ao mesmo.
2. Para a o procedimento anterior calcular os 6 pontos de Bezier.
3. Armazenar os pontos calculados num vetor.

Após a geração dos pontos escreve-se os mesmos num ficheiro .3d evocando a função **create_bezier().**

1.2.3. Gerar um ponto de Bezier

Para calcular um ponto de bezier é utilizado o procedimento seguinte:

1. gerar o polinómio de Bernstein para o ponto U
2. gerar o polinómio de Bernstein para o ponto V
3. Implementar a seguinte fórmula: B(u,v) = Bi(u) * Pij * Bj(v).

Sendo Bi(u) o polinómio para o ponto U, Bj(v) o polinómio para o ponto V e Pij o ponto de controlo calculado e previamente lido do ficheiro .patch.

Engine

1.3. Implementação de VBOs

Inicialmente definiu-se uma função denominada por *filter*, que filtra os pontos repetidos gerados pelo generator e calcula os índices respetivos. Isto é feito da seguinte forma, primeiramente remove-se os repetivos, de seguida, calculam-se os índices, através da lista de pontos com repetidos e sem repetidos, verificando-se na lista com repetidos, os pontos que tem que ser desenhados, e na lista dos sem repetidos, calcula-se o respetivo índice de cada ponto.

Como a complexidade desta função é **N^2+N^2**, então a mesma é feita no generator para melhorar a eficiência do *engine*, pois os ficheiros são gerados sempre antes do executar engine.

De seguida, usando a função *file2list*, que lê os ficheiros gerados pelo generator com os índices e os respetivos pontos, guardam-se em 2 buffers na gráfica esses mesmos indices e pontos, ficando cada um com o seu buffer.

Para a criação dos buffers de pontos e de indices respoetivamente usaram-se as funções :

- glGenBuffers(nFiguras, buffers);
- glGenBuffers(nFiguras, indexes);

No qual a variável nFiguras corresponde ao número de figuras diferentes encontradas no file.xml.

Nota: Os buffers gerados variam com o número de figuras diferentes no projeto pois, a bem da eficiencia, existem 2 buffers para cada figura diferente evitando que existam vários buffers de índices ou pontos para a mesma figura.

Após a leitura, usaram-se as seguintes funções:

- glBindBuffer(GL_ARRAY_BUFFER, buffers[x]);
- glBufferData(GL_ARRAY_BUFFER,sizeof(float)*(it->second.vertexBTAM), it->second.vertexB, GL_STATIC_DRAW);
- glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,indexes[x]);
- glBufferData(GL_ELEMENT_ARRAY_BUFFER,sizeof(unsigned int) *(it->second.indicesTAM), (it->second.indices), GL_STATIC_DRAW);

Para saber que buffer preencher usou-se a função GlBindBuffer.

No primeiro campo desta temos GL_ARRAY_BUFFER o qual possibilita a escrita para o buffer de pontos e GL_ARRAY_BUFFER que é utilizado de forma análoga para os buffers de índices.

Estas funções foram usadas num ciclo for no qual a variável x variava de 0 a nFiguras para a cada posição do buffers e do indexes poder preencher um buffer na gráfica com o conteúdo dos pontos e com o conteúdo dos índices necessários para desenhar uma figura, respetivamente.

Posteriormente fez-se uso da função glBufferData, para preencher cada um dos buffers selecionados com recurso á função glBindBuffer.

Esta função recebe o tipo de array (índices ou pontos) a ser preenchido, o tamanho do array a ser copiado, o array a ser copiado e a forma como este vai ser usado na aplicação.

A variável it é um iterador sobre um map de <string,Figura> o qual é percorrido para garantir a unicidade de cada buffer gerado para cada figura diferente.

Após estes procedimentos, percorremos a class Group que contem o sol, os planetas e respetivas luas.

Para desenhar estes, as funções usadas foram:

- glBindBuffer(GL_ARRAY_BUFFER, buffers[count]);
- glVertexPointer(3, GL_FLOAT, 0, 0);
- glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexes[count]);
- glDrawElements(GL_TRIANGLES, tam, GL_UNSIGNED_INT, 0);

Estas funções usadas por esta ordem indicam:

1. Qual o buffer de pontos a ser utilizado.
2. Quantos elementos do buffer são usados para definir 1 ponto e qual o tipo do elemento.
3. Qual o buffer de índices a usar com o buffer de pontos selecionado anteriormente.
4. Quais as figuras base a ser desenhadas, o número de vertices a desenhar o tipo de elementos guardados no buffer e qual a posição em que se começa a percorrer o mesmo.

1.4. Classes modificadas

A classe *Operation* é responsável por representar uma transformação, das quais o *translate, rotate e scale.*

A classe já estava definida na fase anterior, no entanto, necessitou-se de redefinir a mesma, pois foi-nos proposta a inserção de um novo campo nestas transformações, que é o *tempo que os planetas/cometa/luas* vão demorar a dar uma volta completa na curva de *Catmull-Rom* em que se movem. Tendo sido também alterado o *translate* que em vez de ter um X, Y e Z passou a ter um conjunto de *point* que contêm um X, Y e Z cada um, por fim, apesar de já existir na fase anterior a flag identifica o tipo de transformação.

```
Operation::Operation(char flag, vector<TAD_POINT> points, int time){
    this->flag = flag;
    this->points = points;
    this->time = time;
}
```

Figura 1 – Class Operation.

A classe Figura representa o plane, box, cone e/ou sphere, da mesma forma, que a classe anterior, a mesma estava predefinida na fase precedente, no entanto devido à necessidade de inserção de VBOs, decidiu-se definir os arrays *vertexB* e *índices*, e os respetivos tamanhos, nesta classe para se guardarem os mesmos após a leitura do ficheiro correspondente, e ser mais fácil o acesso dos mesmos.

```
Figura :: Figura(unsigned int* indices, int indicesTAM, float* vertexB, int vertexBTAM){
    this->indices = indices;
    this->indicesTAM = indicesTAM;
    this->vertexB = vertexB;
    this->vertexBTAM = vertexBTAM;
}
```

Figura 2 – Class Figura.

1.5. Desenho das curvas *Catmull-Rom*

Para desenhar as curvas de *Catmull-Rom* foram precisos implementar algumas funções como *renderCatmullRomCurve, getGlobalCatmullRomPoint, getCatmullRomPoint e multMatrixVector.*

A função *renderCatmullRomCurve* é a função que desenha os pontos, mas para isso precisa de calculá-los com a função *getGlobalCatmullRomPoint.* Para obter curvas mais perfeitas a função *getGobalCatmullRomPoint* é invocada 100 vezes para criar mais pontos através dos pontos de controlo lidos do ficheiro **file.xml.** Através do número de vezes que a função *getGlobalCatmullRomPoint* é invocada é possível determinar quais os 4 pontos de controlo a utilizar para invocar a função *getCatmullRomPoint* para obter o próximo ponto da curva de *Catmull-Rom*, o número de invocações da função *getGlobalCatmullRomPoint* varia entre 0 e 1 com incrementos de 0.01.

A função *getCatmullRomPoint* invoca três vezes a função auxiliar *multMatrixVector* para esta fazer a multiplicação entre a *Catmull-Rom Matrix* com os valores de x, y e z de todos os pontos, gerando uma matriz de 3x4. Após estas multiplicações calculam-se as coordenadas finais do ponto multiplicando um vetor T = {t^3, t^2, t, 1} pela matriz calculada anteriormente, o valor de t é o número da invocação da função *getGlobalCatmullRomPoint vezes* o número de pontos de controlo existentes.

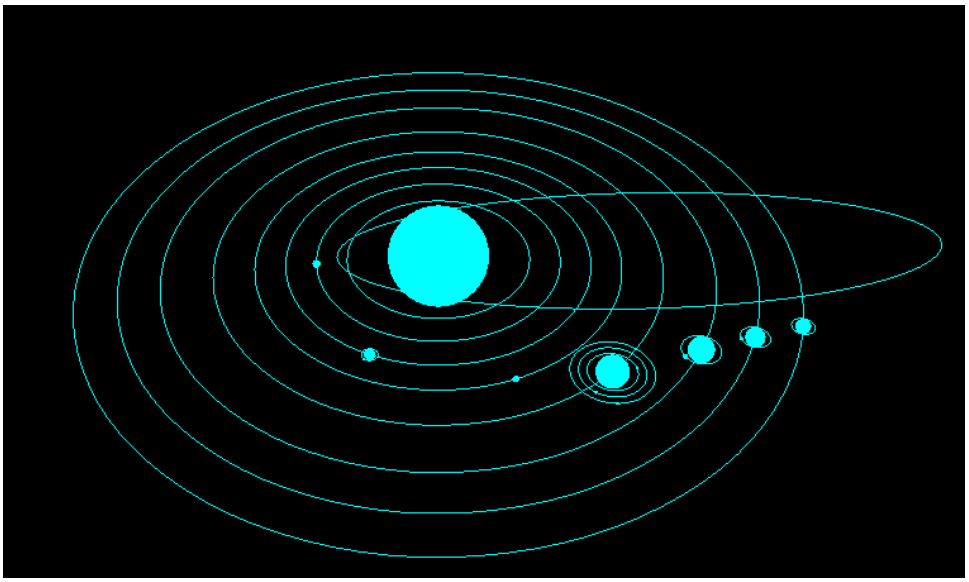

Figura 3 – Demo Scene do Sistema Solar.

Conclusões

Em suma, a terceira fase, foi mais complexa comparado com as fases precedentes, pois necessitou-se de um aprofundamento do conhecimento sobre *Bezier Patchs* e curvas de *Catmull-Rom.*

Existiram algumas dificuldades em aplicar as fórmulas fornecidas pela equipa docente, presentes no formulário da Blackboard, mas que após uma análise das mesmas se dissiparam.

Por fim, os objetivos propostos pelo enunciado foram atingidos, dos quais o *generator* capaz de gerar ficheiros com triângulos, a partir de ficheiros *(.patch)*. O *engine* desenha os pontos com *VBOs* e ainda se tem o mesmo a desenhar as curvas de *Catmull-Rom.*

Lista de Siglas e Acrónimos

XML Extensible Markup Language

VBOs Vertex Buffer Objects