Universidade do Minho
Escola de Engenharia

Universidade do Minho
Departamento de Informática
Computação Gráfica

Trabalho Prático Nº 3

Diogo Araujo (A89517)
António Silva (A89558)
Pedro Novais (A78211)
Gonçalo Soares (A84441)

2 de maio de 2021

Diogo Araújo

António Silva

Gonçalo Soares

Pedro Novais

Conteúdo

Fase 3 – Curvas, Superfícies Cúbicas e VBO's

1.1 Introdução

Nesta fase do projeto o nosso *generator* estará responsável por criar um modelo baseado em *Bezier patches*, onde recebemos um ficheiro *patch* e geramos uma superfície de *Bezier*. É de referir o recurso a VBO's onde está armazenado as coordenadas dos pontos num buffer e noutro os respetivos índices, sendo que posteriormente os triângulos serão renderizados com recurso a estes mesmos.

Em relação ao *engine* passou a ser capaz de fazer translações e rotaçoes amicamente. Começou com uma alteração do parsing dos ficheiros ".3d"no sentido de ser possível ler para VBO's e para as estrutras anteriormente implementadas e para ser possível incluir as curvas de *Catmull-Rom* e rotações sobre o próprio eixo.

1.2 Generator

O propósito do generator nesta fase é criar um ficheiro do tipo ".3d"pronto para ser lido pela *engine*. Recebemos um comando do tipo: "./generator bezier file tesselation", onde o 'file' é o ficheiro .patch a ser lido e 'tesselation' é a tesselação pretendida. O nível de tesselação representa o número de intervalos entre 0 e 1, estes intervalos são respresentados pelo u e v. De seguida utilizamos a seguinte fórmula do formulário para calcular os todos os pontos:

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

Figura 1.1: Fórmula para calcular pontos de Bezier

Esta é aplicada da seguinte maneira: Para cada patch chamamos a função *vector<vertice> createPatch(bezierPatch bp, vector<int> indices, int tesselation)*, que gera todos os pontos desse patch:

```
for(int i = 0; i <= tesselation; i++)
    {
        for(int j = 0; j <= tesselation; j++)
        {

            float u = float (i) / float(tesselation);
            float v = float (j) / float(tesselation);
            vector<float> uMatrix = {u*u*u, u*u,u,1.0};
            vector<float> vMatrix = {v*v*v, v*v,v,1.0};
            //U plus M Matrix
            vector<float> um_Matrix = multMatrix(uMatrix,mMatrix);
            //U plus M plus CpX/CpY/CpZ Matrix
            vector<float> umcpX_Matrix = multMatrix(um_Matrix,cpX);
            vector<float> umcpY_Matrix = multMatrix(um_Matrix,cpY);
            vector<float> umcpZ_Matrix = multMatrix(um_Matrix,cpZ);
            //U plus M plus CpX plus Mt Matrix
            vector<float> umcpXm_Matrix = multMatrix(umcpX_Matrix,mMatrix);
            vector<float> umcpYm_Matrix = multMatrix(umcpY_Matrix,mMatrix);
            vector<float> umcpZm_Matrix = multMatrix(umcpZ_Matrix,mMatrix);

            vertice ver;
            ver.x = umcpXm_Matrix[0]*vMatrix[0] + umcpXm_Matrix[1]*vMatrix[1] +
                    umcpXm_Matrix[2]*vMatrix[2] + umcpXm_Matrix[3]*vMatrix[3];
            ver.y = umcpYm_Matrix[0]*vMatrix[0] + umcpYm_Matrix[1]*vMatrix[1] +
                      umcpYm_Matrix[2]*vMatrix[2] + umcpYm_Matrix[3]*vMatrix[3];
            ver.z = umcpZm_Matrix[0]*vMatrix[0] + umcpZm_Matrix[1]*vMatrix[1] +
                      umcpZm_Matrix[2]*vMatrix[2] + umcpZm_Matrix[3]*vMatrix[3];

            res.push_back(ver);
        }
    }
```

Depois de todos os pontos do patch estarem gerados chamamos a função *vector¡int¿ ordenaPatch(vector¡vertice¿ patch, int tesselation, vector¡int¿ indices)* para guardar num vetor os indices dos pontos a serem desenhados. Diminuindo assim a carga sobre o desenho pois não "desenhamos"os pontos mais que uma vez.

```
    for(int j = 0; j < tesselation;j++)
    {
        for(int i = 0; i < tesselation;i++)
        {
            indices.push_back(i+(j*t));
            indices.push_back(i+t+(j*t));
            indices.push_back(i+1+(j*t));
            indices.push_back(i+t+(j*t));
            indices.push_back(i+t+1+(j*t));
            indices.push_back(i+1+(j*t));
        }
    }
```

1.3 Engine

1.3.1 VBOs

Na engine, o grupo decidiu começar pela implementação das VBOs. Para isto, é necessário antes de tudo, criar o array/vector em C e copiar para a placa gráfica. Este processo só é realizado uma vez durante a vida da aplicação. Primeiro começamos por transformar o array do tipo "vertices"em um array do tipo float com o x, y e z de um ponto guardado pela respectiva ordem.

```
for(; j < globalFigs->figuras[i].vertices.size(); j++)
{
    float xf = globalFigs->figuras[i].vertices[j].x;
    float yf = globalFigs->figuras[i].vertices[j].y;
    float zf = globalFigs->figuras[i].vertices[j].z
    p.push_back(xf);
    p.push_back(yf);
    p.push_back(zf);
}
```

Logo após a transformação do vector dos pontos é criado o VBO e é-lhe atribuído os respectivos vertices e indices a serem desenhados:

```
glGenBuffers(1,&(globalFigs->vbos[contador-1].vertice));
glBindBuffer(GL_ARRAY_BUFFER, globalFigs->vbos[contador-1].vertice);
glBufferData(GL_ARRAY_BUFFER, sizeof(float)*p.size(),p.data(),GL_STATIC_DRAW);

glGenBuffers(1,&(globalFigs->vbos[contador-1].indices));
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, globalFigs->vbos[contador-1].indices);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(unsigned int)*
globalFigs->figuras[i].indices.size()  ,globalFigs->figuras[i].indices.data()
,GL_STATIC_DRAW);
```

Depois da função *prepareData()* ser executada, falta desenhar os triângulos. Isto é feito pela função *drawVBOs(int i,float* rgb)*:

```
void drawVBOs(int i,float* rgb)
{
    int iC =  globalFigs->vbos[i].indexCount;

    glBindBuffer(GL_ARRAY_BUFFER, globalFigs->vbos[i].vertice);
    glVertexPointer(3,GL_FLOAT,0,0);
    glColor3f(rgb[0]/255,rgb[1]/255,rgb[2]/255);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, globalFigs->vbos[i].indices);
    glDrawElements(GL_TRIANGLES, globalFigs->vbos[i].indexCount,GL_UNSIGNED_INT,0);
}
```

1.3.2 Translação

Precisamos de fazer certas alterações à estrutura *translate* para podermos desenhar uma translação seguindo uma curva de Catmull-Rom. Foi adicionado uma flag *isStatic* que nos diz se a translação é estática ou dinâmica. Para caso seja dinâmica, temos na estrutura mais 3 vectores, um para o "xs"dos pontos e outros para os "ys"e os "zs". Para aplicar translações dinâmicas a uma translação usamos esta função *applyDynamicTranslate(translate t)*. Para calcular o tempo parcial, usamos a seguinte fórmula:

$$partialTime = glutGet(GLUT\_ELAPSED\_TIME)/t.time$$

Tal como nas aulas práticas foi dado, usamos esta fórmula da posição :

$$pos = T * M * P$$

Onde T é o vetor $[t^3, t^2, t, 1]$, M a matriz de *Catmull-Rom* e P as coordenadas dos pontos de controlo. Por fim, a derivada é obtida através da equação seguinte :

$$deriv = dT * M * P$$

onde dT é o vetor $[3 * t^2, 2 * t, 1, 0]$, M a matriz de *Catmull-Rom* e P as coordenadas dos pontos de controlo.

1.3.3 Rotação

A rotação em relação à fase anterior evoluiu de uma rotação estática para uma dinâmica, ou seja, varia consoante o tempo. Em termos de código a diferença é que recebe o GLUT_ELAPSED_TIME como argumento o que permite através de divisões sucessivas que a rotação varie ao longo do tempo tal como está demonstrado neste excerto de código :

```
glRotatef(glutGet(GLUT_ELAPSED_TIME)/ globalFigs->rotacoes[t].time,
globalFigs->rotacoes[t].aX, globalFigs->rotacoes[t].aY,
                                globalFigs->rotacoes[t].aZ);
```

1.4 Resultado obtido

O trabalho sofreu bastantes alterações desde a última fase pois passou a ser uma figura estática para algo animado e dinâmico. Todos os planetas têm a sua órbita e conseguem rodar sobre si mesmos, ou seja, têm rotação sendo que o mesmo se aplica para as luas. É de referir que todos os planetas e luas deixaram de ser desenhados a partir de estruturas implementadas por nós mas sim a partir de VBO's. A imagem seguinte mostra o resultado:

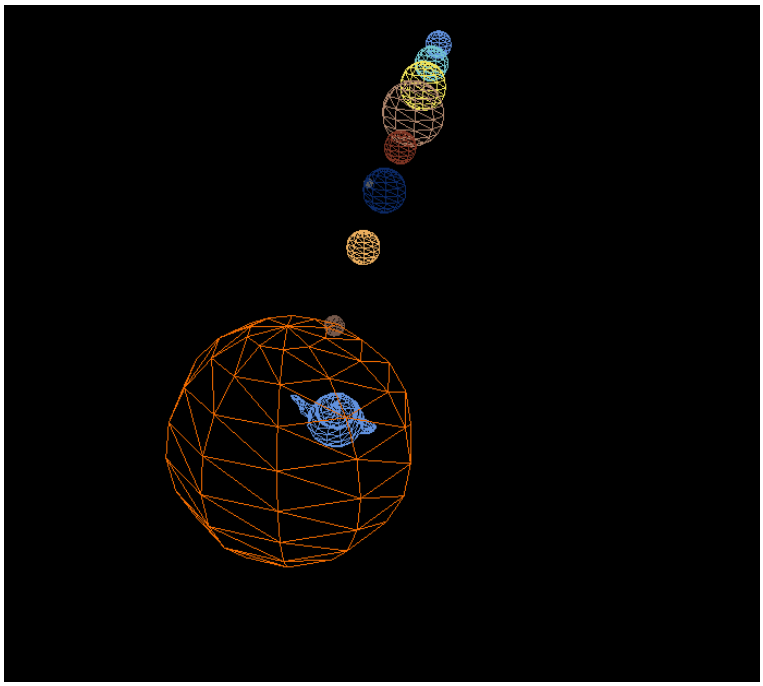

Figura 1.2: Exemplo de Sistema Solar gerado

1.5 Conclusão

Este trabalho introduziu dois termos teóricos lecionados nesta unidade curricular sendo estes as curvas de *Bezier* e *CatmullRom*. A maior dificuldade a superar foram as estruturas usadas nas fases anteriores onde se mostraram muito incompatíveis com as VBO's e a sua implementação, sendo então a *engine* a fase mais demorada deste projeto. No que toca às rotações a implementação foi bastante simples pois apenas alteramos o ângulo consoante o tempo o que levou a poucas alterações no código.

Em relação às translações, o processo foi mais demorado e complexo no entanto, o código está baseado no que foi feito nas aulas práticas onde se aplicou as curvas de *CatmullRom* sendo possível assim fazer translações amicas.