Universidade do Minho
Escola de Ciências

Computação Gráfica (3ª ano de LCC)

Trabalho Prático

3ª Fase

Relatório

Grupo 15

Pedro Manuel Pereira dos Santos	(A100110)
João Manuel Franqueira da Silva	(A91638)
David Alberto Agra	(A95726)
João Pedro da Silva Faria	(A100062)

26 de abril de 2024

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

No âmbito da unidade curricular de Computação Gráfica da Licenciatura em Ciências da Computação, foi proposto o desenvolvimento de duas aplicações utilizando a linguagem C++, recorrendo à ferramenta *OpenGl*.
Nesta terceira fase do projeto foram-nos pedido mudanças em ambas as aplicações. No generator, temos que ser capazes de, dado um ficheiro em que se encontram os patches que definem uma superfície de Bezier, obter os pontos dos triângulos que constroem essa mesma superfície, escrevendo-os também em ficheiro, como é efetuado nas outras primitivas.
No caso da engine, teremos que estender o conceito das translações e rotações, podendo agora estas incluírem atributos temporais. As translações poderão conter pontos de controlo que definem uma curva de Catmull-Rom, e as rotações poderão ser efetuadas durante um certo tempo, num dado eixo. Nesta aplicação, é pedido também que os objetos sejam renderizados com o uso de VBOs.
Para demonstrar a implementação dessas novas funcionalidades, e conforme também nos foi solicitado, criamos um arquivo XML que representa um modelo dinâmico representativo do nosso Sistema Solar. Nele, os planetas terão órbitas em torno do Sol, assim como em torno de si próprios. Além disso, incluiremos um cometa construído a partir das superfícies de Bézier.
Será fornecida uma análise detalhada das decisões e abordagens adotadas para viabilizar a implementação dessas aplicações propostas neste relatório.

Capítulo 2

Generator

Como mencionado, esta aplicação terá a funcionalidade extra de, dado um ficheiro contendo os pontos de controlo de uma superfície de Bezier, bem como um nível de tesselação desejado, escrever noutro ficheiro os vértices dos triângulos que formam a superfície desejada.

2.1 Superfícies de Bezier

Recebendo então um ficheiro .patch, primeiramente extrai-se e organiza-se toda a informação necessária com a função *ReadPatchesFile*. Guardam-se os índices dos 16 pontos que formam cada patch, e em seguida, as coordenadas de cada ponto. Por último, cria-se um vetor, em que para cada patch, terá toda a informação necessária dos seus pontos de controlo.
Depois de extraída a informação, para cada patch e para cada coordenada x, y, z, construímos a matriz P através dos pontos de controlo, e pré-calculamos a sua multiplicação com as matrizes de Bezier para obter os pontos dos triângulos que formam a superfície, multiplicando no final pelas matrizes v e u.
Tal é feito nas funções *bezier_patches* e *surface*, com a ajuda de algumas funções auxiliares, para simplificar as multiplicações de matrizes.

Temos então as seguintes funções:

ReadPatchesFile: Função responsável por extrair informações do arquivo que contém os patches.

```
std::vector<std::vector<std::vector<float>>> readPatchesFile(std::string patch) {

    std::vector<std::vector<int>> indexPatch;
    std::vector<std::vector<float>> controlPoints;

    const char* bpatch = patch.c_str();
    std::fstream file(bpatch);
    std::vector<std::vector<std::vector<float>>> ret;

    if (!file.is_open()) {
        return ret;
    }

    std::string line;

    // Leitura dos patches e do ponto de controlo
    for (int j = 0; j < 2; j++) {
        // Numero de patches/pontos de controlo (1a linha)
        if (!std::getline(file, line)) return ret;
        int n = std::stoi(line);

        // se j == 0, entao esta a ler os patches, se nao esta a ler os pontos de controlo
        for (int i = 0; i < n; i++) {
            if (!std::getline(file, line)) return ret;
            std::vector<int> index;
            std::vector<float> point;
            std::istringstream iss(line);
            std::string token;
            while (std::getline(iss, token, ',')) {
                if (j == 0) {
                    index.push_back(std::stoi(token));
                }
                else {
                    point.push_back(std::stof(token));
                }
            }
            if (j == 0) {
                indexPatch.push_back(index);
            }
            else {
                controlPoints.push_back(point);
            }

        }
    }

    for (std::vector<int> indexes : indexPatch) {
        std::vector<std::vector<float>> patch;
        for (int indexes : indexes) {
            patch.push_back(controlPoints[indexes]);
        }
        ret.push_back(patch);
    }
    file.close();

    return ret;
}
```

Figura 2.1: Codigo da Função ReadPatches

MultMatrixVector: Realiza a multiplicação de uma matriz por um vetor.

```
void multMatrixVector(float m[4][4], float *v, float *res) {

    for (int j = 0; j < 4; ++j) {
        res[j] = 0;
        for (int k = 0; k < 4; ++k) {
            res[j] += v[k] * m[j][k];
        }
    }

}
```

Figura 2.2: Codigo da Função MultMatrixVector

MultMatrixMatrix: Realiza a multiplicação de uma matriz por outra.

```
void multMatrixMatrix(float a[4][4], float b[4][4], float res[4][4]){
    for(int i = 0; i < 4; i++)
        for(int j = 0; j < 4; j++){
            res[i][j] = 0;
            for(int k = 0; k < 4; k++)
                res[i][j] += a[i][k] * b[k][j];
        }
}
```

Figura 2.3: Código da Função MultMatrixMatrix

B: Esta função é referida ao longo da superfície e serve para calcular o polinómio de Bernstein. Sendo que depois é aplicada a cada componente dos pontos (x, y, z).

```
float B(float U, float V, float m[4][4]){

    float aux[4];
    float v[4];
    float r;

    v[0] = powf(V,3);
    v[1] = powf(V,2);
    v[2] = V;
    v[3] = 1;

    multMatrixVector(m,v,aux);

    r = powf(U,3)*aux[0] + powf(U,2)*aux[1] + U*aux[2] + aux[3];

    return r;
```

Figura 2.4: Código da Função B

Surface: Recorre à função B para gerar os pontos que compõem a superfície de Bezier.

```
std::string surface(float mx[4][4], float my[4][4], float mz[4][4], int tesselation){


    std::stringstream buffer;

    float x1, x2, x3, x4, y1, y2, y3, y4, z1, z2, z3, z4;
    float tesselation_level = 1.0/tesselation;

    for(float i=0; i<1; i+=tesselation_level){
        for(float j=0; j<1; j+=tesselation_level){

            x1 = B(i,j,mx);
            x2 = B(i+tesselation_level,j,mx);
            x3 = B(i+tesselation_level,j+tesselation_level,mx);
            x4 = B(i,j+tesselation_level,mx);

            y1 = B(i,j,my);
            y2 = B(i+tesselation_level,j,my);
            y3 = B(i+tesselation_level,j+tesselation_level,my);
            y4 = B(i,j+tesselation_level,my);

            z1 = B(i,j,mz);
            z2 = B(i+tesselation_level,j,mz);
            z3 = B(i+tesselation_level,j+tesselation_level,mz);
            z4 = B(i,j+tesselation_level,mz);
```

Figura 2.5: Código da função surface

BezierPatches: Invoca as funções referidas acima. Obtém os vértices dos triângulos que formam a superfície de Bezier e escreve-os em um arquivo.

```
float bezier[4][4]={
    {-1.0f, 3.0f, -3.0f, 1.0f},
    {3.0f, -6.0f, 3.0f, 0.0f},
    {-3.0f, 3.0f, 0.0f, 0.0f},
    {1.0f, 0.0f, 0.0f, 0.0f}
};
```

Figura 2.6: Matriz Bezier

```
    multMatrixMatrix(bezier, mx, aux);
    multMatrixMatrix(aux, bezier, mx);

    multMatrixMatrix(bezier, my, aux);
    multMatrixMatrix(aux, bezier, my);

    multMatrixMatrix(bezier, mz, aux);
    multMatrixMatrix(aux, bezier, mz);


    myfile << surface(mx,my,mz,tesselation);
}
```

Figura 2.7: Parcela do Código da Função BezierPatches

Capítulo 3

Engine

Para a aplicação Engine, foi-nos pedido então as extensões para as transformações geométricas, translação e rotação.
No caso da translação, são-nos dados os pontos que definem uma curva de Catmull-Rom, bem como o número de segundos que a mesma deve ser percorrida por um objeto, podendo este estar ou não alinhado com a curva. No caso da rotação, podemos ter a opção de um parâmetro temporal, que define o tempo em que a rotação deve ser efetuada sobre um dado eixo. Nesta fase, todos os modelos devem ser renderizados também através de VBOs.

3.1 Rotação

Neste novo tipo de rotação, apenas precisamos de saber o valor do atributo temporal, multiplicando-o por 1000 e guardando este resultado numa variável **tempo**, para obter o seu valor em milissegundos.
Com a função **gluGet(GLUT_ELAPSED_TIME)**, obtemos o tempo decorrido aquando da inicialização do programa. Multiplicando este valor pela divisão de 360 com variável previamente referida, obtemos o ângulo que um dado objeto deveria ter rodado. Por último, basta aplicar a rotação com o ângulo calculado em torno do eixo representado pelo vetor indicado no arquivo XML.

3.2 Translação

Agora, as translações podem ser constituídas também por um conjunto de pontos de controlo, que definem uma curva de Catmull-Rom, por onde os futuros modelos que herdarem esta transformação, terão que percorrer, com a duração que o parâmetro de tempo indicar, podendo ou não estes serem alinhados pela curva
Começamos em primeiro lugar, por renderizar a curva resultante dos pontos de controlo. De seguida, para o tempo atual, calcula-se a posição e a derivada na curva, efetuando uma translação com o valor da posição. Caso o objeto necessite de ser alinhado, atribuímos a um vetor **x** o valor da derivada, e a partir do produto externo, obtemos dois vetores **y** e **z**, em que aplicando uma matriz de rotação a estes 3, conseguimos orientar o objeto pela curva.
Por último, atualizamos as variáveis de tempo, consoante o tempo decorrido, através de
gluGet(GLUT_ELAPSED_TIME).

3.3 VBOs

Por último, de forma a melhorar a performance do nosso programa e conforme nos foi pedido, avançamos para a implementação de VBOs para o desenho da nossa cena.
Em primeiro lugar, continuamos ainda a guardar os nossos pontos num vetor global **numberArrays**. No entanto, desta vez, quando encontramos um arquivo cujos pontos foram extraídos, guardamos no vetor global a posição no mesmo vetor que contém os pontos desse arquivo.
Em seguida, geramos um ID de buffer para cada figura única a ser desenhada. Percorremos o vetor **numberArrays**, transferindo a informação de cada elemento que contenha pontos para uma posição do vetor de **GLuint**, **vbo**, que será incrementada. Quando nos deparamos com um elemento que aponta para a posição onde se encontram os pontos da figura, então, não transferimos dados repetidos, otimizando assim o nosso programa. Em ambos os casos, guardamos num vetor de pares global **mapa**, um par com a primeira componente sendo o índice do vetor **numberArrays** e a segunda o respetivo índice que nos diz em que posição os pontos se encontram no vetor **vbo**.
Para desenhar, percorremos o vetor **numberArrays** e, para cada elemento, desenhamos os pontos que se encontram na posição do vetor **vbo** indicada pelo vetor de pares.

Capítulo 4

Funcionalidades Extra

Como incentivado no enunciado do trabalho, tomamos a iniciativa de estender os nossos arquivos XML, acrescentando um grupo **eixos** com um atributo **value**, e, caso tenha o valor **false**, então os eixos do plano não serão renderizados. Para o caso das translações estendidas, podemos também encontrar um atributo **draw**, que caso seja **false**, indica que a curva não deve também ser renderizada.
Por questões de compatibilidade com os arquivos XML que nos são fornecidos, caso estes atributos não se encontrem declarados, então a cena renderizada não é afetada.

Eixos: Grupo com atributo value, que indica se os eixos do sistema devem ser ilustrados.

```
<world>
    <eixos value="false" />
    <window width="1000" height="1000" />
    <camera>
```

Figura 4.1: Grupo eixos

Draw: Atributo que indica se a curva gerada pelos pontos de controlo deve ser ou não renderizada.

```
<group> <!-- Lua -->
    <transform>
        <translate time = "8" align="true" draw="false">
    <point x="12.0207" y="0" z="12.0207"/>
    <point x="0" y="0" z="17"/>
```

Figura 4.2: Atributo draw

Capítulo 5

Sistema Solar

Com o propósito de demonstrar as funcionalidades implementadas nesta fase, foi-nos solicitado então um arquivo XML que modele dinamicamente o nosso Sistema Solar, fazendo uso das novas transformações estendidas e das superfícies de Bezier.

5.1 Modelos Utilizados

Para o desenho do Sistema Solar, usaremos as esferas para representar os planetas, as luas e o Sol, um toro para o anel de Saturno, e um bule de chá gerado a partir de superfícies de Bezier, com a intenção de representar um cometa.

5.2 XML Utilizado

O modelo XML utilizado já apresenta um maior nível de complexidade, comparado ao da fase anterior. Primeiramente, começamos por desenhar o Sol, aplicando a este uma rotação temporal e uma escala.
Agora, para todos os planetas, através da nova extensão para as translações, construímos uma curva de Catmull-Rom com 6 pontos, que respeite a distância relativa de cada um ao Sol, colocando então cada planeta alinhado com a curva, também com uma rotação temporal aplicada, de forma a este rodar sobre o próprio, e por último, uma escala. No caso de alguns planetas que possuem luas, para a lua construímos uma curva de Catmull-Rom à volta do seu planeta, com o novo atributo **draw**, de forma a não renderizar a tal curva, e por último, uma rotação temporal, seguida por uma escala.
No caso do anel de Saturno, apenas efetuamos uma rotação, para inclinar minimamente o próprio. De resto, basta apenas garantir que após a escala realizada para desenhar Saturno, o raio da menor circunferência pertencente ao anel seja superior ao raio do planeta.

```
<group> <!-- Neptuno -->
    <transform>
        <translate time = "280" align="true" >
    <point x="212.13" y="0" z="212.13"/>
    <point x="0" y="0" z="300"/>
    <point x="-212.13" y="0" z="212.13"/>
    <point x="-300" y="0" z="0"/>
    <point x="-212.13" y="0" z="-212.13"/>
    <point x="0" y="0" z="-300"/>
    <point x="212.13" y="0" z="-212.13"/>
    <point x="300" y="0" z="0"/>
        </translate>
        <rotate time="16" x="0" y="1" z="0"/>
        <scale x="0.79" y="0.79" z="0.79" />
    </transform>
    <models>
        <model file="sphere_10_20_20.3d" /> <!-- ./generator sphere 10 20 20 sphere_10_20_20.3d -->
    </models>
    <group> <!-- Lua Neptuno-->
        <transform>
            <translate time = "20" align="True" draw="false">
        <point x="12.0207" y="0" z="12.0207"/>
        <point x="0" y="0" z="17"/>
        <point x="-12.0207" y="0" z="12.0207"/>
        <point x="-17" y="0" z="0"/>
        <point x="-12.0207" y="0" z="-12.0207"/>
        <point x="0" y="0" z="-17"/>
        <point x="12.0207" y="0" z="-12.0207"/>
        <point x="17" y="0" z="0"/>
    </translate>
            <rotate time= "7" x = "0" y = "1" z = "0" />
            <scale x="0.12" y="0.12" z="0.12" />
        </transform>
        <models>
            <model file="sphere_10_20_20.3d" /> <!-- ./generator sphere 10 20 20 sphere_10_20_20.3d -->
        </models>
    </group>
</group>

<group> <!-- Cometa -->
    <transform>

        <translate time = "10" align="true">
            <point x = "0" y = "-80" z = "80" />
            <point x = "80" y = "-80" z = "0" />
            <point x = "0" y = "300" z = "-40" />
            <point x = "-40" y = "300" z= "0" />
        </translate>
        <rotate angle="-90" x="1" y="0" z="0"/>
        <scale x="3" y="3" z="3" />
    </transform>
    <models>
        <model file="bezier_10.3d" /> <!-- generator patch teapot.patch 10 bezier_10.3d -->
```

Figura 5.1: Parte do ficheiro XML para o Sistema Solar

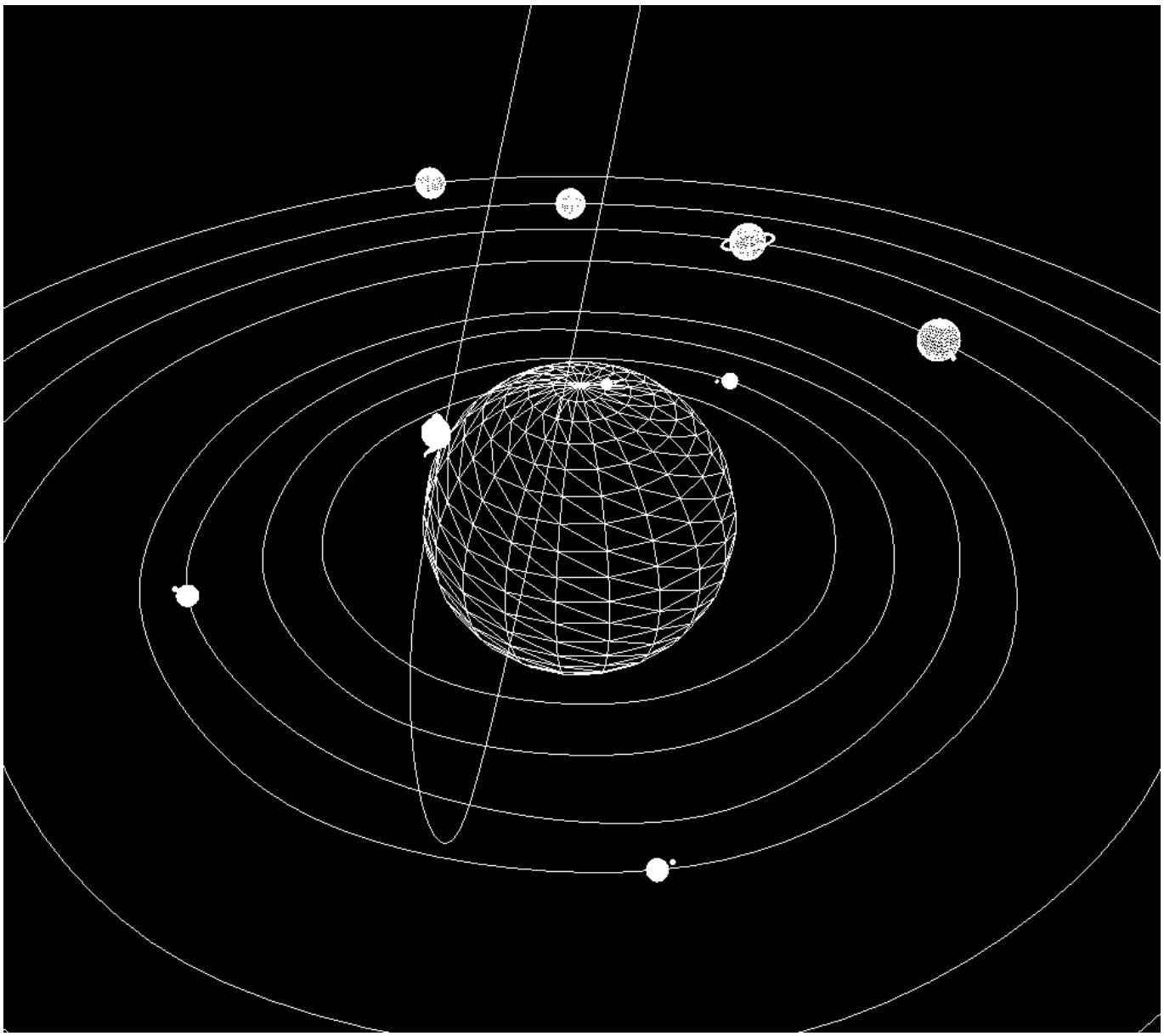

Figura 5.2: Desenho do Sistema Solar

Capítulo 6

Conclusão

Concluindo então esta terceira fase do projeto, destacamos a implementação bem-sucedida das funcionalidades propostas, como as extensões pretendidas para as transformações geométricas.
Conseguimos também, com sucesso, construir superfícies de Bezier, através de um arquivo com os pontos de controle, sendo agora capazes de modelar objetos mais complexos.
Por último, destacamos também a melhoria de desempenho das nossas aplicações, devido à transição para VBOs para a renderização dos nossos modelos, bem como algumas melhorias na correção de tipos das nossas variáveis globais, evitando cálculos desnecessários, otimizando assim o nosso programa.
A cena dinâmica do sistema solar, com os planetas a rodarem à volta do sol e entre si próprios, bem como as luas rodarem à volta dos seus planetas, e com um cometa construído a partir de um arquivo .patch, é a perfeita demonstração de como os objetivos desta fase foram alcançados.
Resumindo, o grupo atribui um balanço positivo ao trabalho realizado nesta fase, tendo cumprido todos os requisitos pedidos, e sentimos-nos fortemente preparados para a última etapa deste projeto.