Universidade do Minho
Escola de Engenharia

Computação Gráfica

Fase 3
Grupo 7

Braga, Maio de 2023

Bernardo Amado Pereira da Costa, A95052
Eduardo Miguel Pacheco Silva, A95345
José Carlos Gonçalves Braz, A96168

Índice

1. Introdução

Nesta terceira fase foi-nos proposta a ilustração de rotações ou translações, ou seja, uma rotação completa (360˚) passa a ser efetuada num certo período de tempo, ao passo que uma translação é definida à custa duma curva de *Catmull-Rom* e esta por sua vez também dispõe dum período de tempo para ser realizada. Houve, também, a adição duma nova primitiva (*Teapot*) baseada em *Bezier Patches.*

Recorre-se a *VBOs* com índices para redesenhar as primitivas no que diz respeito ao cálculo dos seus vértices.

2. Novos Ficheiros

2.1. Splines.cpp & splines.h

Estes são dois novos ficheiros e, como tal, iremos começar por apresentar o propósito da criação dos mesmos. Este módulo apresenta métodos para obtenção e cálculos de pontos e derivadas duma curva de *Catmull-Rom*.

Tal como se poderá imaginar, a função **`getCatmullRomPoint`** é usada para calcular um ponto e a sua respetiva derivada ao longo duma secção da curva que está definida por 4 pontos de controlo (*p0, p1, p2, p3*). Nessa função é utilizada uma matriz de *Catmull-Rom* para calcular as posições e derivadas em cada eixo.

Por sua vez, a função **`getGlobalCatmullRomPoint`** usa a função **`getCatmullRomPoint`** para calcular na mesma o ponto e a derivada mas numa posição geral na curva que é especificada pelo parâmetro gt (entre 0 e 1). A curva é definida por um vetor de pontos (catmrPts).

Já a função **`getBezierPoint`** usa as equações de Bezier para calcular a posição e o vetor normal da curva num ponto específico (u,v). A matriz de Bezier é pré-calculada e multiplicada pelos vetores de controlo, produzindo assim pontos de curva. São, também, calculadas as derivadas parciais relativamente a (u,v) e isso permite-nos determinar a orientação da curva.

O objetivo da criação deste ficheiro é a implementação de funções que nos permitam trabalhar com splines cúbicas utilizando para tal a técnica de *Catmull-Rom.*

3. VBOs

Temos 3 modelos presentes no nosso Sistema Solar que são a caixa, a esfera e o anel. A estes aplicaremos transformações com VBOs. Os *Virtual Buffer Objects* são uma funcionalidade oferecida pelo OpenGL que nos permite inserir informação sobre os nossos modelos diretamente na placa gráfica do nosso dispositivo. A performance será evidentemente muito melhor, pois a renderização será realizada de imediato. Criamos um VBO para os modelos, sendo que os dados dos nossos modelos (por exemplo, pontos que lhe estão associados) são guardados num *map* juntamente com a VBO que lhes está associada. Isto permite-nos também poupar espaço em memória e ser eficientes. Apresentamos, de seguida, as funções que nos permitem executar esta tarefa:

```
vector<std::tuple<GLuint, int, int>> GeometricShape::convertToVBO(vector<GSPoints> gsps) {
    vector<std::tuple<GLuint, int, int>> res;
    for (GSPoints gsp : gsps) {

        vector<float> f_pts;
        for (_3f p : gsp.getPoints()) {
            f_pts.push_back(p.x);
            f_pts.push_back(p.y);
            f_pts.push_back(p.z);
        }

        GLuint vbo;
        glGenBuffers(1, &vbo);
        glBindBuffer(GL_ARRAY_BUFFER, vbo);
        glBufferData(GL_ARRAY_BUFFER, f_pts.size() * sizeof(float), f_pts.data(), GL_STATIC_DRAW);

        res.push_back(make_tuple(vbo, gsp.getPrimitive(), f_pts.size()));
    }

    return res;

}
```

```
void GeometricShape::drawObjectVBOMode(vector<std::tuple<GLuint, int, int>> v) {
    for (auto t : v) {
        glBindBuffer(GL_ARRAY_BUFFER, std::get<0>(t));
        glEnableClientState(GL_VERTEX_ARRAY);
        glVertexPointer(3, GL_FLOAT, 0 ,0);
        glDrawArrays(std::get<1>(t), 0, std::get<2>(t));
    }
}
```

A função **`drawObjectVBOMode`** permite-nos desenhar com os pontos guardados no buffer e a função **`convertToVBO`** devolverá um vetor de tuplos onde cada tuplo contem um Vertex Buffer Object. É criada uma nova VBO através do método **`glGenBuffers`** e utiliza o método **`glBufferData`** para povoar a VBO com os dados necessários dos pontos. Assim conseguimos uma renderização mais eficiente dos nossos modelos.

Estes métodos serão depois utilizados numa classe chamada GeometricShape como se pode ver pela figura abaixo:

```
class GeometricShape {
protected:
    vector<GSPoints> points;
    string fileName;

public:
    virtual vector<GSPoints> getPoints() {return points;};

    static void drawObject(vector<GSPoints> points);
    static void drawObjectVBOMode(vector<std::tuple<GLuint, int, int>>);

    static void writeTo3DFile(vector<GSPoints> points, string fName);
    static vector<std::tuple<GLuint, int, int>>convertToVBO(vector<GSPoints> gsps);
    static vector<GSPoints> readFrom3DFile(string fName);
    static vector<std::tuple<GLuint, int, int>> readFrom3DFileVBOMode(string fName);
    static vector<GSPoints> readFromBezierPatchFile(string pathFName,  int tesselation);

    string getFileName() { return fileName; }


    friend ostream &operator<<(ostream &out, const GeometricShape &go) {
        go.Print(out);
        return out;
    }

private:
    virtual void Print(ostream &) const = 0;
};
```

As alterações especificadas aqui foram realizadas nos ficheiros geometricShapes.cpp e geometricShapes.h, sendo depois estes métodos novamente chamados no ficheiro materials.cpp e, podendo também ser chamados em engine.cpp se assim se entender para contexto de teste.

4. Transformações e Curvas de *Catmull-Rom*

4.1. Translação

Como já abordamos anteriormente quando escrevemos sobre a criação dos ficheiros de splines, as funções relativas às curvas de *Catmull* estão incluídas nestes ficheiros. Isto permite-nos trabalhar com splines cúbicas, descobrindo os pontos e as suas derivadas ao longo duma curva. Transformações como translações e rotações serão definidas ao longo de um certo período de tempo. Incluímos, também, um atributo booleano (*align*) que define no XML se o objeto a ser transformado (no caso duma translação) deverá estar alinhado com a curva ou não no momento da transformação.

```
Translate::Translate(XMLElement * translate) : t(0), isAlign(false), prev_y(_3f(0, -1, 0)) {
    this->p = _3f(
        translate->FloatAttribute("x"),
        translate->FloatAttribute("y"),
        translate->FloatAttribute("z")
    );

    XMLElement * translate_child = _getChildElement(translate, "point");
    if (translate_child) {

        this->time = translate->FloatAttribute("time");
        this->isAlign = translate->BoolAttribute("align");

        while (translate_child) {
            this->catmullPoints.push_back(_3f(
            translate_child->FloatAttribute("x"),
            translate_child->FloatAttribute("y"),
            translate_child->FloatAttribute("z")
        ));
            translate_child = translate_child->NextSiblingElement("point");
        }
    }
}
```

```
void Translate::run() {
    glTranslatef(this->p.x, this->p.y, this->p.z);
    if (this->catmullPoints.size() > 0) {
        if (this->catmullPoints.size() < 4) {
            throw "excpected atleast 4 points but less were given!";
        }
        else {

            renderCatmullRomCurve(this->catmullPoints);

            _3f pos = _3f();
            _3f deriv = _3f();

            getGlobalCatmullRomPoint(this->t, &pos, &deriv, this->catmullPoints);

            glTranslatef(pos.x, pos.y, pos.z);

            if (this->isAlign) {

            _3f x = deriv;
            x.normalize();
            _3f z = _3f::cross(x, this->prev_y);
            z.normalize();
            _3f y = _3f::cross(z, x);
            y.normalize();
            this->prev_y = y;

            float m[16];
            buildRotMatrix(x, y, z, m);
            glMultMatrixf(m);

            }

            this->t += 1 / (this->time * 100);
        }
    }
}
```

Esta última figura exemplifica a implementação do método run de uma translação. Ele é responsável por executar uma translação e definição da curva de *Catmull-Rom* (no caso de existirem, pelo menos, 4 pontos para que tal seja possível). Caso seja possível, faz-se a renderização da curva e calcula a posição e a derivada da curva no ponto correspondente ao tempo atual *t*. Este cálculo é realizado com base na função já referida e especificada anteriormente **`getGlobalCatmullRomPoint`**. Se a opção align estiver habilitada, o método também realiza uma rotação para manter a orientação do objeto ao longo da curva. Para isso, ele calcula a nova base ortonormal (vetores x, y e z) usando a derivada da curva, o vetor *prev_y* (que armazena a orientação anterior) e as funções **`normalize`** e **`cross`** da classe _3f. Em seguida, usamos a função **`buildRotMatrix`** para criar uma matriz de rotação e multiplica-se a matriz atual com a função **`glMultMatrixf`**.

4.2. Rotação

De modo a ser possível implementar uma nova forma de rotação (definida ao longo de um certo período de tempo) foi necessário implementar uma variável que acompanha o tempo decorrido. Caso contrário, a rotação será executada instantaneamente com o ângulo definido. No caso de ser suposto a rotação ser realizada ao longo dum período de tempo, o método **`glRotatef()`** recebe *t* como um parâmetro que faz o objeto rodar durante uma porção de tempo. *t* é definido como **`t += 360 / (this->time * 1000)`**.

4.3. Desenho das curvas de *Catmull-Rom*

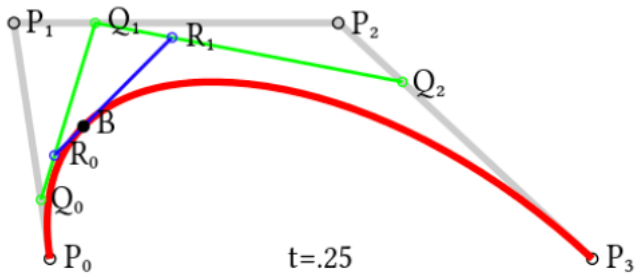
O método **`renderCatmullRomCurve`** é responsável por renderizar a curva na cor respetiva tendo por base um conjunto de pontos inseridos num vetor de pontos (catmrPts) sobre o qual já tínhamos escrito na secção 2. A curva é desenhada utilizando um loop while e a cada iteração do loop o método **`getGlobalCatmullRomPoint`** é chamado. Como já é sabido, este método permite-nos calcular uma posição e uma derivada dum ponto ao longo da curva. Este ciclo contém uma variável que vai incrementando em porções de 1/100.

5. Bézier

Antes de realizar o processamento do ficheiro de input (.patch) foi necessário entender o mesmo para gerar a figura. O que se concluiu foi:

- Na primeira linha surge o número de patches;
- As seguintes linhas (que são o total de patches) são compostas por 16 números inteiros que correspondem ao total de índices de pontos de controlo que são parte do patch;
- De seguida existe um inteiro que representa o número de pontos de controlo;
- No final surgem esses mesmos pontos de controlo.

Para a criação duma curva de Bézier são necessários 4 pontos definidos tridimensionalmente. Porém, esses pontos por si só não geram a curva, ou seja, terão que ser combinados com alguns coeficientes. De seguida apresentamos um exemplo duma curva de Bézier:



Esta curva é definida por uma equação onde existe uma variável t∈[0,1] de tal modo que:

$$B(t) = (1-t)^3 * P0 + 3 * t * (1-t)^2 * P1 + 3 * t^2 * (1-t) * P2 + t^3 * P3$$

O resultado desta equação para qualquer t contido no intervalo de 0 a 1 corresponde a uma posição na curva. P0, P1, P2 e P3 são os pontos de controlo. Resolvendo a equação diversas vezes substituindo t por vários valores no seu intervalo, achamos toda a curva. O problema que nos é apresentado é semelhante a este princípio, com a nuance de que em vez de termos 4 pontos teremos 16.

Iremos focar-nos em duas funções para este capítulo: **`readFromBezierPatchFile`** e **`getBezierPoint`**.

A função **`readFromBezierPatchFile`** recebe como parâmetros o caminho do arquivo e a quantidade de tecelagem para a forma geométrica. Lemos o número de patches e criamos um array bidimensional de inteiros que armazena os índices dos pontos para cada patch. Em seguida, é lido o número de pontos e criam-se três arrays de pontos que armazenam as coordenadas x, y e z para cada ponto. Depois, cada patch é lido, extraem-se os pontos correspondentes aos índices do array bidimensional e usa-se a função **`getBezierPoint`** para calcular os pontos de Bezier para cada patch. Os pontos são

adicionados ao vetor points, que é usado para criar o objeto GSPoints. Importante será referir que quanto maior for a tecelagem, melhor irá ser a figura.

6. Funcionalidades adicionais

6.1. *Specular Mapping* na esfera

Specular mapping é uma técnica utilizada para criar superfícies com reflexos brilhantes, simulando o comportamento de materiais como plástico, metal e vidro. Essa técnica é aplicada em conjunto com outras técnicas, como o mapeamento de texturas, para criar uma aparência mais realista.

Neste caso, aplicamos na classe *Sphere* para adicionar mais realismo aos planetas presentes no nosso modelo. A realização de *Specular mapping* processa-se através dum ficheiro que contém informações sobre a refletividade na superfície esférica. O valor de multiplier é utilizado para ajustar o valor da refletividade.

A partir dessas informações, o código utiliza a função **`height`** para obter a altura da superfície em cada ponto da esfera, com base na refletividade da textura. Esse valor é utilizado para calcular a posição dos vértices da esfera, de modo que a superfície apresente reflexos brilhantes onde a refletividade é maior.

Os vértices são armazenados num vetor e adicionados à lista de pontos GSPoints, que serão utilizados para renderizar a esfera com a técnica de GL_TRIANGLE_STRIP. O resultado final é uma esfera com uma aparência mais realista, com brilhos e reflexos na superfície, simulando a presença de materiais brilhantes ou metálicos.

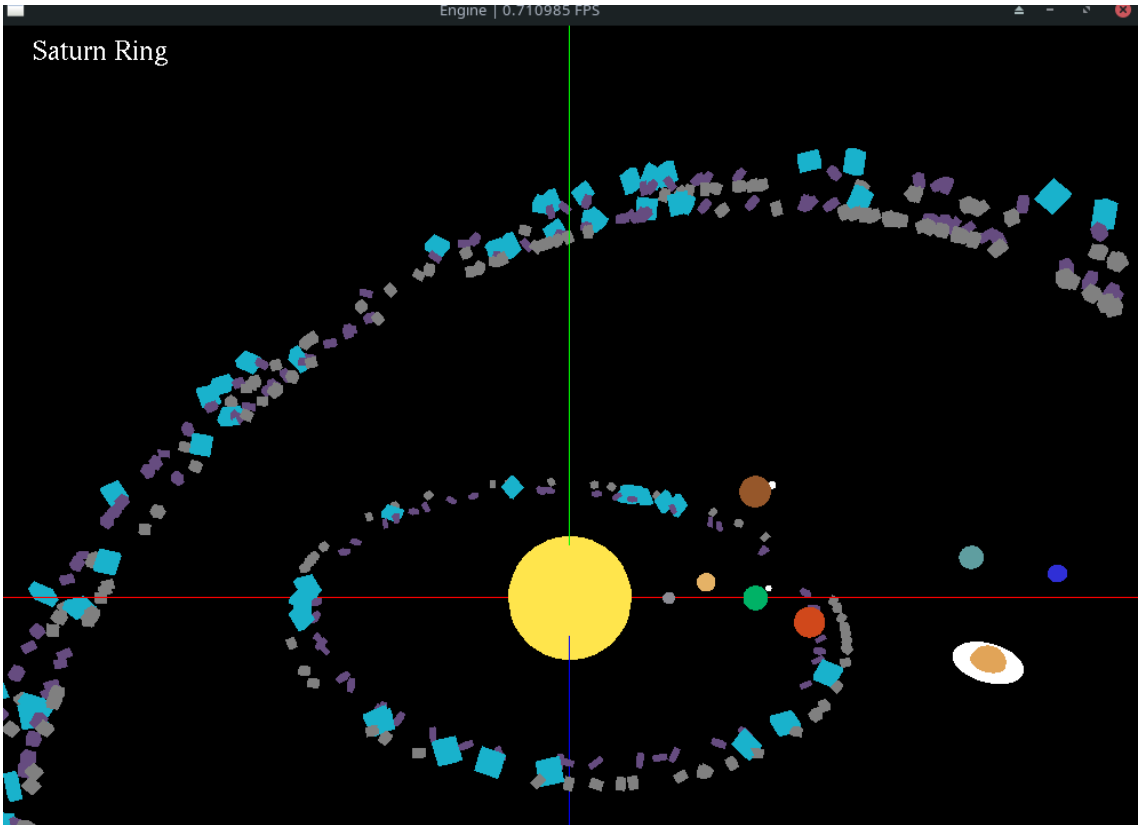
Para observar os ficheiros que dão origem a esta funcionalidade é necessário ir à pasta *test_files* e aí estarão os ficheiros em formato .jpg.

6.2. *DisableCull*

Aplicamos esta propriedade por causa dos anéis que é um dos 3 modelos que temos. Esta propriedade é aplicada na class Model e é utilizada para garantir que estes sejam renderizados corretamente, não sendo ocultados em certas *views,* o que resultaria numa aparência irregular. Para tal, a funcionalidade tem que tomar um valor de verdadeiro, havendo a necessidade de verificar *disablecull.*

6.3. Identificação dos planetas através dum *click*

Consideramos que seria uma boa decisão implementar uma simples *feature* de quando se movimenta e prime o rato em cima da representação do mesmo aparecer o nome do planeta e as coordenadas onde o mesmo se encontra.



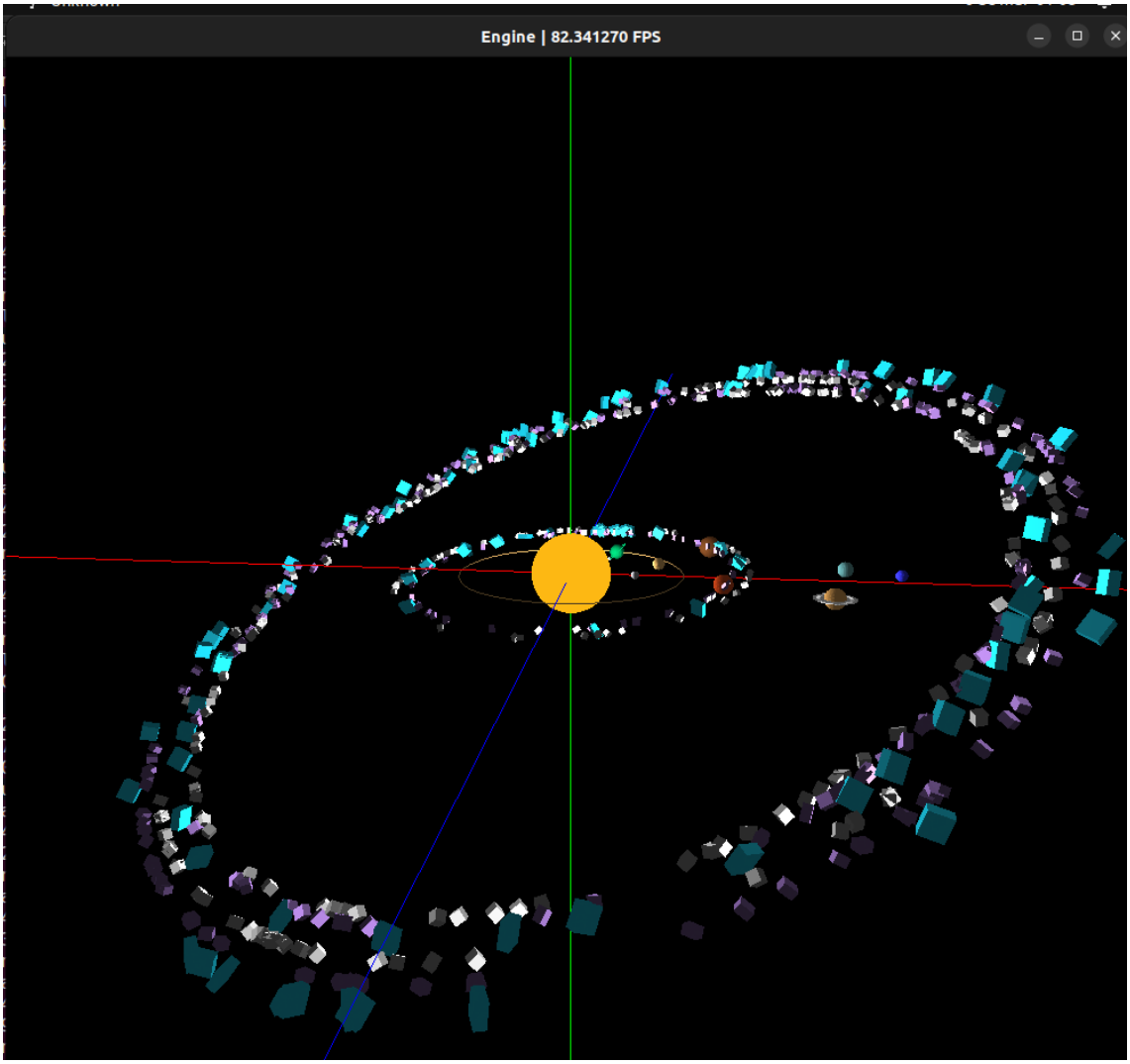
6.4. *Imports* no XML

Como é possível observar na pasta *test_files* existem 3 ficheiros .xml. Um é destinado à cintura de Kuiper, outro ao asteróide e outro ao Sistema Solar em geral. Para compor a cena num todo, apenas é necessário realizar *import* dos dois primeiros ficheiros .xml para o solar_system.xml. Julgamos que isto é importante para manter os ficheiros de construção e configuração de cena organizados e mais facilmente personalizáveis, dado que a escala do projeto vai sendo cada vez maior.

Anexamos o exemplo de como importar o asteroid_group.xml para o ficheiro xml do sistema solar:

```
<group file="../../test_files/solar_system/asteroid_group.xml"/> <!--
ASTEROID BELT GROUP -->
```

7. Resultados obtidos



8. Conclusão

Nesta fase conseguimos explorar os conceitos de animações em transações e rotações, bem como as Curvas de *Catmull-Rom* que foram um auxílio para o cálculo de pontos.

A abordagem de superfícies curvas de Bézier permitiram a construção dum *Teapot* através duma sequência de instruções que simplificam bastante a tarefa.

Uma grande utilidade a nível de eficiência consistiu no desenho com recurso a VBOs. A renderização dos modelos é realizada de imediato e a informação insere-se diretamente na placa gráfica, o que também representa otimização de memória.

As funções extra que adicionamos também permitem melhorar o projeto e torná-lo mais completo e organizado.

Em suma, julgamos ter cumprido com os objetivos propostos e ter consolidado bastante a nossa aprendizagem na UC.