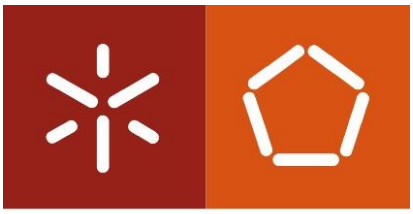

Universidade do Minho

Departamento de Informática

Relatório CG Fase 3
Grupo 31

Gustavo Lourenço a89561 João Machado a89510
Martim Almeida a89501

Ano Letivo 2020/21

Conteúdo

Capítulo 1

Introdução

Este relatório tem o objectivo de apresentar a terceira fase do Projecto da UC Computação Gráfica.

Neste relatório iremos apresentar os requisitos necessários presentes nesta fase:

- O *Generator* de ficheiros com a informação de um determinado Modelo que foi modificado de acordo com os requisitos pedidos desta fase para ler *Patches* de *Bezier*;
- O *Engine* que inicialmente lê um ficheiro escrito em XML que contém os modelos e posteriormente faz *display* dos mesmos, e que para esta fase também foi modificada para que todos os modelos no *engine* fossem trocados para *VBO*s. As translações foram modificadas para serem feitas em curvas de *Catmull-Rom* e as rotações foram modificadas para que sejam feitas em torno de eixos num determinado tempo.
- Uma *Scene* que gera uma representação aproximada do nosso Sistema Solar que levou às modificações acima mencionadas e a adição de nova primitiva, um *patch* de *Bezier* de um *Teapot* a servir de cometa.

Capítulo 2

Generator

As Primitivas que são possíveis de gerar da fase anterior são:

- Plano
- Caixa
- Esfera
- Cone
- *Torus*

Nesta fase implementamos a capacidade do *generator* interpretar *Patches* de *Bezier*.

2.1 Patches de Bezier

2.1.1 Parsing

Primeiro damos *parse* ao nível de *tesselation*(*tesselationLevel*). Para ler os dados presentes no ficheiro primeiro lemos o número de *patches* presentes (*numPatches*). Em seguida lemos os n *patches* seguintes e guardamos num vetor de vetores (*indices*). Depois está presente o número de pontos de controlo (*numControlPoints*). Por fim lemos os n pontos seguintes para um vetor de pontos (*controlPoints*).

2.1.2 Calcular Pontos

Para calcular os pontos criamos uma função, *getBezierPoint*, que calcule um ponto ao longo de um conjunto de 4 pontos (p0, p1, p2, p3) com base na matriz de constantes de *bezier* e o tempo decorrido (t). A parte de calcular as coordenadas do ponto é exatamente igual à parte de calcular um ponto numa curva de *Catmull-Rom* tirando a diferença na matriz de constantes.

```
1 // Bezier matrix
2     float m[4][4] = {{-1.0f, +3.0f, -3.0f, +1.0f},
3                      {+3.0f, -6.0f, +3.0f, +0.0f},
4                      {-3.0f, +3.0f, +0.0f, +0.0f},
5                      {+1.0f, +0.0f, +0.0f, +0.0f}};
```

Em seguida, utilizamos esta função para calcular as coordenadas de um ponto dentro de um *patch*. A função criada para isso recebe o índice do *patch* (*indice*) e o espaço relativo percorrido para cada lado do *patch* (u e v). Primeiro criamos um vetor com todos os pontos que o *patch* contem. Ou seja, converter índices de pontos para os pontos em si. Depois aplicamos a função calculada para obter o ponto com as coordenadas u dentro de cada *patch* de *bezier* de 4 em 4 pontos. Depois aplicamos novamente essa conta aos 4 pontos calculados e utilizamos a mesma função mas desta vez com o v.

```
std::vector<Point> patch = std::vector<Point>();
    std::vector<int> patchPoints = indices[patchNum];

    for(int i=0; i<patchPoints.size();i++)
        patch.push_back(controlPoints[patchPoints[i]]);

    std::vector<Point> bezierPoints;
    for (int i = 0;i < patch.size();i += 4) {
        float p[3] = {};
        getBezierPoint(u, patch[i], patch[i+1], patch[i+2], patch[i+3],p);
        bezierPoints.push_back(Point(p[0],p[1],p[2]));
    }

    float pos[3] = {};

    getBezierPoint(v, bezierPoints[0], bezierPoints[1], bezierPoints[2],
        bezierPoints[3], pos);

    return Point(pos[0],pos[1],pos[2]);
```

Dividindo 1 pelo *tesselationLevel* obtemos o delta de cada ponto dentro de um *patch*. Para obter triângulos válidos é necessário saber os 3 pontos que vêm imediatamente a seguir ao ponto que está a ser calculado. Para isso basta calcular as combinações de pontos para $j+1$ e para $k+1$.

```

for (int i = 0; i < numPatches; i++) {

    for (int j = 0; j < tesselationLevel; j++) {

        float u = j * step;
        float nextU = (j+ 1) * step;

        for (int k = 0; k < tesselationLevel; k++) {

            float v = k * step;
            float nextV = (k + 1) * step;

            Point p0 = getGlobalBezierPoint(i, u, v);
            Point p1 = getGlobalBezierPoint(i, u, nextV);
            Point p2 = getGlobalBezierPoint(i, nextU, v);
            Point p3 = getGlobalBezierPoint(i, nextU, nextV);

            points.push_back(p3);
            points.push_back(p2);
            points.push_back(p1);

            points.push_back(p2);
            points.push_back(p0);
            points.push_back(p1);
            }
        }
    }
```

Capítulo 3

Engine

O *Engine* foi a parte do projeto que mais mudanças sofreu. As principais mudanças feitas no *Engine* nesta fase foram:

- Os modelos foram convertidos para *VBOs*;
- As translações foram modificadas para que usem curvas de *Catmull-Rom*;
- As rotações foram modificadas para que sejam feitas em torno de eixos num determinado tempo.

3.1 Conversão para VBOs

Nas fases anteriores os modelos eram desenhados com o modo imediato no entanto para esta fase foi necessário que estes sejam desenhados com *VBOs*.

3.1.1 Estruturas de Dados

Nas fases anteriores utilizamos *vectors* para cada *Model* guardar as coordenadas dos seus pontos (através de uma estrutura de dados *std::vector<Point>*), nesta fase mudamos estas estruturas para utilizar *VBOs*. Para que tal fosse possível passamos a utilizar um array *GLuint vertices[1]* para que seja utilizado como um *VBO* e criamos uma variável *int vertices* para guardar o número de valores contidos no *VBO*.

```
class Model {
    private:
        GLuint vertices[1];
        int verticesCount;

    public:
        Model(const char *);
        void drawModel();
};
```

Para a criação deste *VBO*, utilizamos um *vector* para guardar os valores necessários enquanto iteramos pelo ficheiro do modelo, e, após acabar o parsing do modelo, utilizamos o tamanho do *vector* para obter o número de vértices e também utilizamos os comandos *glGenBuffers*, *glBindBuffer* e *glBufferData* para criar o *VBO* e armazenar os seus valores.

```
Model::Model(const char* fileName){
    std::vector<float> points = std::vector<float>();
    float x,y,z;
    std::ifstream file(fileName);
        while(file >> x >> y >> z){
            points.push_back(x);
```

```
            points.push_back(y);
            points.push_back(z);
        }
    verticesCount = points.size();
    glGenBuffers(1,vertices);
    glBindBuffer(GL_ARRAY_BUFFER, vertices[0]);
    glBufferData(GL_ARRAY_BUFFER,sizeof(float)*points.size(), points.data(),
        GL_STATIC_DRAW);
}
```

Para desenhar o respectivo modelo temos de associar novamente o nosso array e depois utilizamos as funções *glVertexPointer* e *glDrawArrays*.

```
void Model::drawModel(){
    glBindBuffer(GL_ARRAY_BUFFER, vertices[0]);
    glVertexPointer(3,GL_FLOAT,0,0);
    glDrawArrays(GL_TRIANGLES,0,verticesCount);
}
```

3.2 Transformações com tempo

3.2.1 Rotação

Nesta fase fizemos alterações à transformação de rotação estabelecida na última fase, adicionamos uma variável *time* à rotação que indica o tempo para fazer uma rotação completa em torno do eixo desejado, para tal temos de comunicar o tempo decorrido desde o início da representação da cena e multiplicamos por *360/time*.

```
void Rotate::transform(float timestamp){
    float realAngle;
    if(time == 0)
        realAngle = angle;
    else
        realAngle = angle + timestamp * (360/time);
    glRotatef(realAngle,x,y,z);
}
```

3.2.2 Translação - Curvas de Catmull-Rom

Para fazer uma curva de Catmull-Rom fazemos o parsing do *Translate* e necessitamos do tempo para fazer a translação pela curva e dos pontos de controlo para a criação da curva (o número mínimo de pontos é 4).

```
if(!strcmp(type->Value(), "translate")){
            if(type->ToElement()->Attribute("time")){
                std::vector<Point> nPoints = std::vector<Point>();
                float time = std::stof(type->ToElement()->Attribute("time"));
                std::vector<Point> auxPoints = translationParser(type);
                for(Point p : auxPoints)
                    nPoints.push_back(p);
                nCat = CatmullRom(time,nPoints);
            }
}
```

```
std::vector<Point> Models::translationParser(tinyxml2::XMLNode* points){
    std::vector<Point> nPoints = std::vector<Point>();
    tinyxml2::XMLNode* type = points->FirstChild();
    while(type){
        if(!strcmp(type->Value(), "point"))
            nPoints.push_back(Point(std::stof(type->ToElement()->Attribute("X")),
                                    std::stof(type->ToElement()->Attribute("Y")),
                                    std::stof(type->ToElement()->Attribute("Z"))));
        type = type->NextSibling();
    }
    return nPoints;
}
```

Para calcular a posição e o vetor de direção de um ponto numa curva de Catmull-Rom necessitamos de 4 pontos da curva, de um valor t que representa o instante atual entre os pontos *p1* e *p2* e de uma matriz m com os valores da matriz de Catmull-Rom. Começamos por calcular 2 vetores: $tv = [t^3, t^2, t, 1]$ (necessário para o cálculo da posição na curva) e $dtv = [3t^2, 2t, 1, 0]$ (necessário para o cálculo da derivada na curva), após isso nós iteramos por cada coordenada para obter as repectivas coordenadas da posição e da derivada. Para tal, calculamos o vetor a através da multiplicação de m pelo vetor com os valores da coordenada que desejamos calcular de cada um dos pontos. Por fim podemos obter o valor da coordenada da posição multiplicando os valores de a pelos valores de tv e conseguimos obter o valor da derivada multiplicando os valores de a pelos valores de dtv.

```
void CatmullRom::getCatmullRomPoint(float t, Point p0, Point p1, Point p2, Point p3
    , float *pos, float *deriv) {

    // catmull-rom matrix
```

```
    float m[4][4] = {   {-0.5f,  1.5f, -1.5f,  0.5f},
                        { 1.0f, -2.5f,  2.0f, -0.5f},
                        {-0.5f,  0.0f,  0.5f,  0.0f},
                        { 0.0f,  1.0f,  0.0f,  0.0f}};

    float tv[4] = { t * t * t,t * t,t,1 };
    float dtv[4] = { 3*t * t ,2*t ,1,0 };

    float p0a[3] = {p0.get_x(),p0.get_y(),p0.get_z()};
    float p1a[3] = {p1.get_x(),p1.get_y(),p1.get_z()};
    float p2a[3] = {p2.get_x(),p2.get_y(),p2.get_z()};
    float p3a[3] = {p3.get_x(),p3.get_y(),p3.get_z()};

    for (int i = 0; i < 3; i++)
    {
        float p[4] = { p0a[i],p1a[i],p2a[i],p3a[i] };
        float a[4];
        // Compute A = M * P
        multMatrixVector(m, p, a);
        // Compute pos = T * A
        pos[i] = 0;
        for (int j = 0; j < 4; j++) {
            pos[i] += tv[j] * a[j];
        }
        // compute deriv = T' * A
        deriv[i] = 0;
        for (int j = 0; j < 4; j++) {
                deriv[i] += dtv[j] * a[j];
        }
    }
}
```

A função *getCatmullRomPoint* é chamada através de uma função *getGlobalCatmullRomPoint* que recebe um valor *gt* entre 0 e 1 que indica onde o corpo se encontra na curva e, através deste valor e do número de pontos conseguimos obter o t e os 4 pontos necessários para chamar a função *getCatmullRomPoint*.

```
void CatmullRom::getGlobalCatmullRomPoint(float gt, float *pos, float *deriv) {
    int POINT_COUNT = points.size();
    float t = gt * POINT_COUNT; // this is the real global t
    int index = floor(t);  // which segment
    t = t - index; // where within  the segment

    // indices store the points
    int indices[4];
    indices[0] = (index + POINT_COUNT-1)%POINT_COUNT;
    indices[1] = (indices[0]+1)%POINT_COUNT;
    indices[2] = (indices[1]+1)%POINT_COUNT;
    indices[3] = (indices[2]+1)%POINT_COUNT;


    getCatmullRomPoint(t, points[indices[0]], points[indices[1]], points[indices
        [2]], points[indices[3]], pos, deriv);
}
```

Por fim, para fazer a translação através da curva necessitamos de um valor *timestamp* o tempo decorrido desde o início da representação da cena, multiplicamos o número de pontos de controlo por 30 para obter o nível de tesselação, após isto utilizamos a função *getGlobalCatmullRomPoint* com o *timestamp* dividido pelo tempo para fazer uma translação completa pela curva para obter a posição do corpo (podendo fazer assim a translação) e a derivada, após isto calculamos o ponto *X[3] = deriv*, o ponto *Y0[3]* = {*0,1,0*}, o ponto *Z[3] = X*Y0* e o ponto *Y[3] = Z*X*, aposá normalizar os pontos X, Y e Z, podemos calcular a matriz de rotação do corpo.

```
void CatmullRom::transform(float timestamp) {

    if(points.size()>=4){
```

```
        float pos[3];
        float deriv[3];
        const float tessNum = points.size()*30;
        const float mod = time/tessNum;
        getGlobalCatmullRomPoint(timestamp/time, pos, deriv);
        glTranslatef(pos[0], pos[1], pos[2]);

        float X[3] = { deriv[0], deriv[1], deriv[2] };
        normalize(X);

        float Y0[3] =  {0,1,0};

        float Z[3];
        cross(X,Y0,Z);
        normalize(Z);

        float Y[3];
        cross(Z,X,Y);
        normalize(Z);

        float m[16];

        buildRotMatrix(X,Y,Z,m);

       glMultMatrixf(m);
    }
}
```

Caso seja nesserário observar a curva de translação obtida podemos utilizar os seguintes comandos:

```
glBegin(GL_LINE_LOOP);
    glColor3f(1.0f,1.0f,1.0f);
    for (int i = 0; i < tessNum; i++)
    {
        getGlobalCatmullRomPoint(mod*i, pos, deriv);
        glVertex3f(pos[0],pos[1],pos[2]);
    }
glEnd();
```

3.3 Keybinds

Keybind	**Descrição**
1, **2**, **3**	Modifica aspeto das primitivas
+, **-**	*Zoom In* e *Zoom Out*
wasd	Modificação do Ângulo da Câmara
r	*Reset*
p	Pausar *Scene*
i,o	Alteração da Velocidade da *Scene*
←↓↑→	Movimentação da Câmara

Capítulo 4

Sistema Solar

Para criar o Ficheiro do XML do sistema solar era necessário simular as translações dos planetas em torno do Sol e representar um grande número de luas, devido as estes factores sentimos a necessidade de usar um *script* para facilitar a criação do Ficheiro de XML, para tal foi feito um *script* em *Python*.

Para a realização deste modelo nós utilizamos as transformações geométricas implementadas nesta fase, da seguinte maneira:

- *Rotate*: foi utilizado para rodar o anel de Saturno, os próprios planetas e suas luas sobre os seus eixos;
- *Translate*: foi utilizado para simular a translação dos planetas em torno do Sol, e das luas em torno dos seus respectivos planetas assim como a translação de um cometa em torno do sistema solar com uma trajetória elíptica;
- *Scale*: foi utilizado para gerar corpos celestes de tamanhos diferentes, e gerar uma *Torus* mais consistem com um disco para simular o anel de Saturno;
- *Color*: foi utilizado para gerar corpos celestes de cores diferentes;
- *Group*: foi utilizado para agrupar os planetas com as suas luas e no caso de Saturno o seu anel.

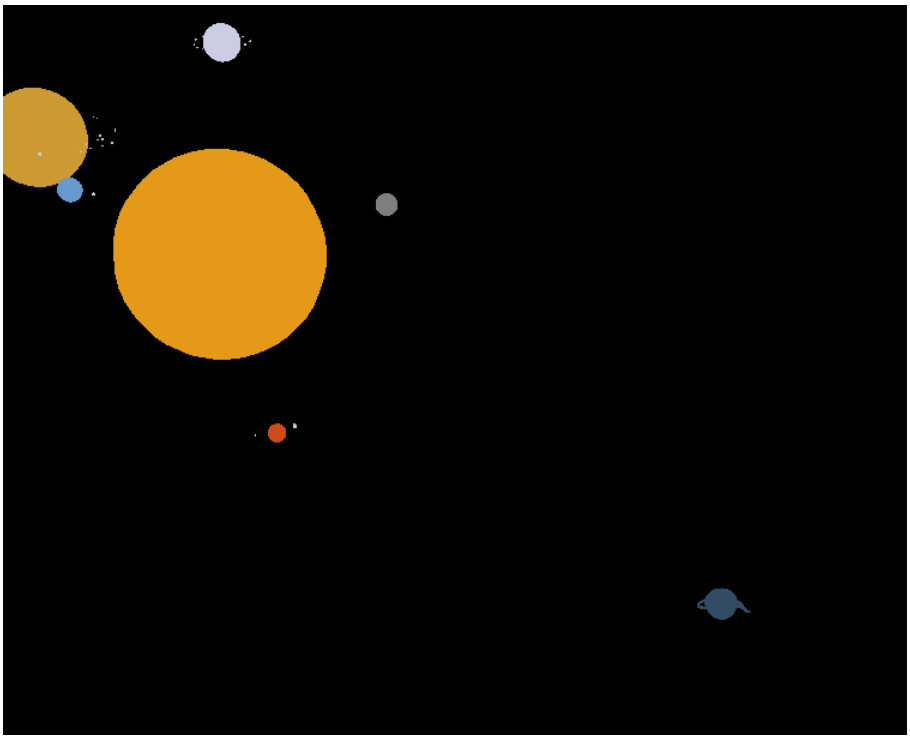

Figura 4.1: Sistema Solar

Capítulo 5

Conclusão

Com esta Fase aprendemos como aplicar diversos conteúdos leccionados nas aulas como a uso e implementação de transformações geométricas. Também aprofundamos ainda mais os conhecimentos da linguagem C++ assim como *CMakeLists*.

Futuramente, gostaríamos de implementar uma optimização para que o projecto funcione em todos os Sistemas Operativos assim como o uso de diversas directorias de forma a melhorar organizar o nosso projecto, também gostaríamos de mudar a forma como damos cor aos objectos, criando objectos com cores e padrões variados.