Computação Gráfica: Fase 3 – Curvas, Superfícies cúbicas e VBOs

Afonso Faria,
André Ferreira,
Bruno Campos e
Ricardo Gomes

Universidade do Minho, Braga, Portugal
{a83920,a93211,a93307,a93785}@alunos.uminho.pt

Resumo Nesta fase foi-nos proposta a implementação de *Bézier Patches*, curvas cúbicas de *Catmull-Rom*, rotações com tempo e *VBOs*.

1 Introdução e Contextualização

O presente relatório diz respeito à terceira fase do projeto da Unidade Curricular de Computação Gráfica.

Nesta fase foi-nos proposta a implementação de modelos baseados em *Bézier Patches* no *generator* e a implementação de curvas cúbicas de *Catmull-Rom*, rotações com tempo e *VBOs* no *Engine*. Adicionalmente, foi-nos proposto uma *demo scene* de um sistema solar dinâmico.

Nas seguintes secções do relatório iremos descrever a abordagem tomada na resolução dos problemas e os obstáculos que surgiram durante a realização do projeto.

2 *Generator*

2.1 *Bézier Patches*

Na criação de modelos baseados em *Bézier Patches*, tiramos partido do conhecimento transmitido nas aulas teóricas e práticas.

```
void multMatrixVector(float *m, 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 * 4 + k];
        }
    }
}

glm::vec3 calcBezierCurve(
    float t,
    glm::vec3 p0,
    glm::vec3 p1,
    glm::vec3 p2,
    glm::vec3 p3
) {
    glm::vec3 res;
    float A[3][4];
    float m[4][4] = { {-1,  3, -3, 1},
                      { 3, -6,  3, 0},
                      {-3,  3,  0,  0},
                      { 1,  0,  0,  0} };

    for (int i = 0; i < 3; i++) {
        float pp[4] = { p0[i],p1[i],p2[i],p3[i] };
        multMatrixVector(*m, pp, A[i]);

        // Compute pos = T * A
        res[i] = t * t * t * A[i][0] + t * t * A[i][1] + t * A[i][2] + A[i][3];
    }
    return res;
}

auto generate_bezier_patch(
    std::ifstream& patch_input_file,
    u32 const tesselation
) noexcept -> cpp::result<std::vector<glm::vec3>, GeneratorErr>
try {
    auto const& bezier_patch = TRY_RESULT(parse_patch_file(patch_input_file));
    glm::vec3 p0, p1, p2, p3;
```

```
glm::vec3 pp0, pp1, pp2, pp3;
float tess = tesselation;
float step = 1 / tess;
glm::vec3 pa, pb, pc, pd;

auto points = bezier_patch.ctrl_points;
auto vertices = std::vector<glm::vec3>{};


for (auto const& patch : bezier_patch.indices)
{

for (float i = 0; i < 1; i += step)
{
    p0 = calcBezierCurve(i, points[patch[0]], points[patch[1]],
                            points[patch[2]], points[patch[3]]);
    p1 = calcBezierCurve(i, points[patch[4]], points[patch[5]],
                            points[patch[6]], points[patch[7]]);
    p2 = calcBezierCurve(i, points[patch[8]], points[patch[9]],
                            points[patch[10]], points[patch[11]]);
    p3 = calcBezierCurve(i, points[patch[12]], points[patch[13]],
                            points[patch[14]], points[patch[15]]);

    pp0 = calcBezierCurve(i + step, points[patch[0]], points[patch[1]],
                                    points[patch[2]], points[patch[3]]);
    pp1 = calcBezierCurve(i + step, points[patch[4]], points[patch[5]],
                                    points[patch[6]], points[patch[7]]);
    pp2 = calcBezierCurve(i + step, points[patch[8]], points[patch[9]],
                                    points[patch[10]], points[patch[11]]);
    pp3 = calcBezierCurve(i + step, points[patch[12]], points[patch[13]],
                                    points[patch[14]], points[patch[15]]);

    for (float j = 0; j < 1; j += step)
    {
        pa = calcBezierCurve(j, p0, p1, p2, p3);
        pb = calcBezierCurve(j, pp0, pp1, pp2, pp3);
        pc = calcBezierCurve(j + step, p0, p1, p2, p3);
        pd = calcBezierCurve(j + step, pp0, pp1, pp2, pp3);

        vertices.emplace_back(pa[0],pa[1],pa[2]);
        vertices.emplace_back(pb[0], pb[1], pb[2]);
        vertices.emplace_back(pc[0], pc[1], pc[2]);

        vertices.emplace_back(pb[0], pb[1], pb[2]);
        vertices.emplace_back(pd[0], pd[1], pd[2]);
        vertices.emplace_back(pc[0], pc[1], pc[2]);
```

```
        }
    }
    }
    return vertices;
} catch (std::bad_alloc const&) {
    return cpp::fail(GeneratorErr::NO_MEM);
} catch (std::length_error const&) {
    return cpp::fail(GeneratorErr::NO_MEM);
}
```

Utilizando a função `calcBezierCurve` calculamos o ponto que se encontra na curva de Bézier, tendo em conta os pontos de controlo e o tempo num dado momento. Tirando partido desta função, conseguimos definir uma "rede"de pontos na superfície, com n colunas e linhas, onde n é a tesselação pedida.

Finalmente, geramos um ficheiro `bezier.3d` que contém 6 vértices por cada quadrilátero da rede (2 triângulos). Para um ficheiro com múltiplos *patches* basta fazer o mesmo para cada um.

3 *Engine*

3.1 Curva cúbica de Catmull-Rom

Na realização de translações temporizadas em volta de uma curva de *Catmull-Rom*, reutilizamos grande parte do código feito nas aulas práticas:

```
void buildRotMatrix(float *x, float *y, float *z, float *m) {

    m[0] = x[0]; m[1] = x[1]; m[2] = x[2]; m[3] = 0;
    m[4] = y[0]; m[5] = y[1]; m[6] = y[2]; m[7] = 0;
    m[8] = z[0]; m[9] = z[1]; m[10] = z[2]; m[11] = 0;
    m[12] = 0; m[13] = 0; m[14] = 0; m[15] = 1;
}

void cross(float *a, float *b, float *res) {

    res[0] = a[1]*b[2] - a[2]*b[1];
    res[1] = a[2]*b[0] - a[0]*b[2];
    res[2] = a[0]*b[1] - a[1]*b[0];
}

void normalize(float *a) {

    float l = sqrt(a[0]*a[0] + a[1] * a[1] + a[2] * a[2]);
    a[0] = a[0]/l;
    a[1] = a[1]/l;
    a[2] = a[2]/l;
}

void multMatrixVector(float *m, 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 * 4 + k];
        }
    }
}

void getCatmullRomPoint(float t, glm::vec3 p0, glm::vec3 p1, glm::vec3 p2,
                                 glm::vec3 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}};
```

```
    // Compute A = M * P
    float A[3][4];
    for (int i = 0; i < 3; i++) {
        float pp[4] = { p0[i],p1[i],p2[i],p3[i] };
        multMatrixVector(*m, pp, A[i]);

    // Compute pos = T * A
        pos[i] = t * t * t * A[i][0] + t * t * A[i][1] + t * A[i][2] + A[i][3];

    // compute deriv = T' * A
        deriv[i] = 3 * t * t * A[i][0] + 2 * t * A[i][1] + A[i][2];
    }
}

// given  global t, returns the point in the curve
void getGlobalCatmullRomPoint(float gt,std::vector<glm::vec3> p,
                                        float *pos, float *deriv) {
    float t = gt * p.size(); // 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 + p.size()-1)%p.size();
    indices[1] = (indices[0]+1)%p.size();
    indices[2] = (indices[1]+1)%p.size();
    indices[3] = (indices[2]+1)%p.size();

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

void renderCatmullRomCurve(std::vector<glm::vec3> points) {
    float pos[3];
    float deriv[3];
// draw curve using line segments with GL_LINE_LOOP
    glBegin(GL_LINE_LOOP);
    for (int i = 0; i <= 100; i += 1) {
        getGlobalCatmullRomPoint(i*0.01,points,pos, deriv);
        glVertex3f(pos[0], pos[1], pos[2]);
    }
    glEnd();
}


//...
```

```
[](DynamicTranslate const& dynamic_translate) {
    float pos[3];
    float deriv[3];
    static float yo[3] = { 0, 1, 0 };
    float z[3];
    float m[16];

    float gt = glutGet(GLUT_ELAPSED_TIME);
    float timer = dynamic_translate.time;
    float realt = gt / (timer * 1000);

    renderCatmullRomCurve(dynamic_translate.points);
    getGlobalCatmullRomPoint(realt,dynamic_translate.points, pos, deriv);

    glTranslatef(pos[0], pos[1], pos[2]);
    if(dynamic_translate.align){ // if assign = True
        normalize(deriv);
        cross(deriv, yo, z);
        normalize(z);
        cross(z, deriv, yo);
        normalize(yo);
        buildRotMatrix(deriv, yo, z, m);
        glMultMatrixf(m);
    }
}

// ...
```

Após obter a posição atual do modelo na curva de *Catmull-Rom*, falta apenas garantir que este percorre uma volta no tempo requerido. Para isso, obtemos o tempo de execução através da chamada à função `glutGet(GLUT_ELAPSED_TIME)`, e dividimos pelo tempo requirido multiplicado por 1000.

Por fim, se for pedido que o modelo seja alinhado com a curva (`align="True"`), multiplicamos a matriz de transformações atual por uma construida por 3 vetores normalizados (correspondentes a x, y e z) calculados a partir da derivada da curva de Catmull-Rot.

3.2 Rotações com tempo

Na implementação de rotações de modelos com temporizador, apenas precisamos de calcular o passo atual da rotação, tendo em conta o temporizador. Depois, tirando partido da chamada à função `glutGet(GLUT_ELAPSED_TIME)`, resta apenas multiplicar o valor por 360^{o} e aplicar a rotação resultante.

```
// ...

case ROTATE:
    switch (rotate.kind) {
        using enum Rotate::Kind;

        case Angle:
            glRotatef(
                rotate.rotate[0],
                rotate.rotate[1],
                rotate.rotate[2],
                rotate.rotate[3]
            );
            break;

        case Time: {
            float gt = glutGet(GLUT_ELAPSED_TIME);
            float timer = rotate.rotate[0];
            float realt = gt / (timer * 1000);
            glRotatef(
                360 * realt,
                rotate.rotate[1],
                rotate.rotate[2],
                rotate.rotate[3]
            );
        } break;
    }

// ...
```

3.3 VBOs

De forma a suportar *VBOs* foram definidas três novas variáveis:

- `std::vector<std::vector<float>> buffers;`
- `GLuint bind[500];`
- `int iii = 0;`

`buffers` guarda os pontos de um modelo sequencialmente, na ordem `x, y, z`.

`bind` indica que *arrays* estão ativos. Como ainda se trata de uma fase de prototipagem, definimos um limite de 500 modelos ativos em *VBOs*.

Por fim, `iii` guarda o índice do modelo cujos pontos estão guardados na variável `buffers`. Este índice é utlizado na chamada à função `glDrawArrays`.

O primeiro passo é gerar os vectores contidos na variável `buffers`.Para isso, definiu-se a função `bufferVBOs`, que percorre de forma recursiva todos os `Groups`, guardando apenas os pontos de cada modelo num vetor.
Era necessário fazer este passo pois os `Models` estavam guardados em pontos. Ao utilizar o bufferVBOs os models passaram a ser representados apenas por vértices (as componetes x,y,z dos pontos).

```
auto static bufferVBOs(Group const& root) noexcept -> void {
    for (auto const& model : root.models) {
        std::vector<float> BufferModel;
        for (auto const& vertex : model.vertices) {
            BufferModel.push_back(vertex[0]);
            BufferModel.push_back(vertex[1]);
            BufferModel.push_back(vertex[2]);
        }
        state::buffers.push_back(BufferModel);
    }
    for (auto const& child_node : root.children) {
        bufferVBOs(child_node);
    }
}
```

Utlizando esta função no set_world.É nesta parte que faz-se o bind com os dados contidos no buffers com o arry bind.

```
auto Renderer::set_world(World& world) -> Renderer& {
    if (auto* const world_ptr = &world;
        world_ptr != state::world_ptr
    ) {
        state::world_ptr = world_ptr;
        state::model_refs = state::build_model_refs(world);
        bufferVBOs(state::world_ptr->root);

        int i = 0;

        glEnableClientState(GL_VERTEX_ARRAY);
```

```
        glGenBuffers(500,state::bind);
        int tamanho = static_cast<int> (state::buffers.size());

        for(int i = 0; i < tamanho;i++){
            glBindBuffer(GL_ARRAY_BUFFER, state::bind[i]);
            glBufferData(GL_ARRAY_BUFFER,
                         sizeof(float) * state::buffers[i].size(),
                         state::buffers[i].data(),GL_STATIC_DRAW);
        }

    }
    return *this;
}
```

Por fim, faltou alterar a função render_group, função resopnsável para desenhar os grupos de forma recursiva. Nesta função era essencial haver uma correspondência entre o buffers com o desenho que o render_group estava a desenhar no momento, isto é, fazer uma correspondência enter um array e de uma árvore. Para isso criou-se a variável iii que diz qual é o model atual a desenhar no vector buffers quando se esta a percorrer a árvore. Era necessário percorrer a árvore uma vez que as transformações só podiam afetar um certo número de models.
Como na criação do buffers fez-se da mesma forma que se esta o render_group esta variável será sempre incrementada sempre que denha um model (conjunto de pontos) e volta a 0 quando acaba de desenhar todos os models.
Ao ter esta correspondência, era só necessário dar bind com o conteúdo do vector do model a desenhar e desenhar tendo em conta que antes de desenhar continua-se a fazer primeiro as transformações geométricas pedidas.

```
auto static render_group(Group const& root) noexcept -> void {
    glPushMatrix();

    for (auto const& transform : root.transforms) {
        switch (transform.kind) {
            using enum Transform::Kind;

            case TRANSLATE:
                glTranslatef(
                    transform.translate.x,
                    transform.translate.y,
                    transform.translate.z
                );
                break;

            case ROTATE:
                glRotatef(
                    transform.rotate[0],
                    transform.rotate[1],
                    transform.rotate[2],
```

```
                    transform.rotate[3]
                );
                break;

            case SCALE:
                glScalef(
                    transform.scale.x,
                    transform.scale.y,
                    transform.scale.z
                );
                break;
        }
    }


    for (auto const& model : root.models) {

        glBindBuffer(GL_ARRAY_BUFFER, state::bind[iii]);
        glVertexPointer(3,GL_FLOAT,0,0);
        glDrawArrays(GL_TRIANGLES,0,state::buffers[iii].size()/3);
        iii++;
    }
    for (auto const& child_node : root.children) {
        render_group(child_node);
    }


    glPopMatrix();
}
```

4 Resultados

Utilizando os ficheiros de configuração XML fornecidos pelos docentes para fins de testes, bem como os modelos especificados nestes, obtemos os seguintes resultados:

4.1 Teste 1- Patch Bezier com curvas

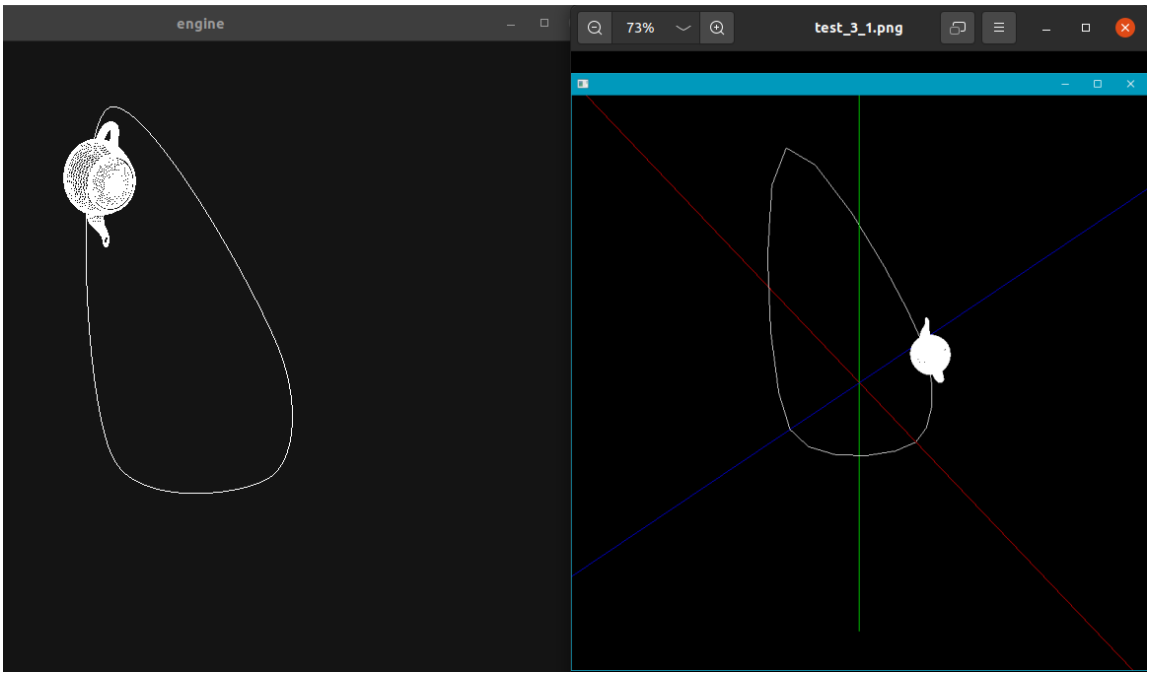

Figura 1. test_3_1.xml

4.2 Teste 2- Patch Bezier

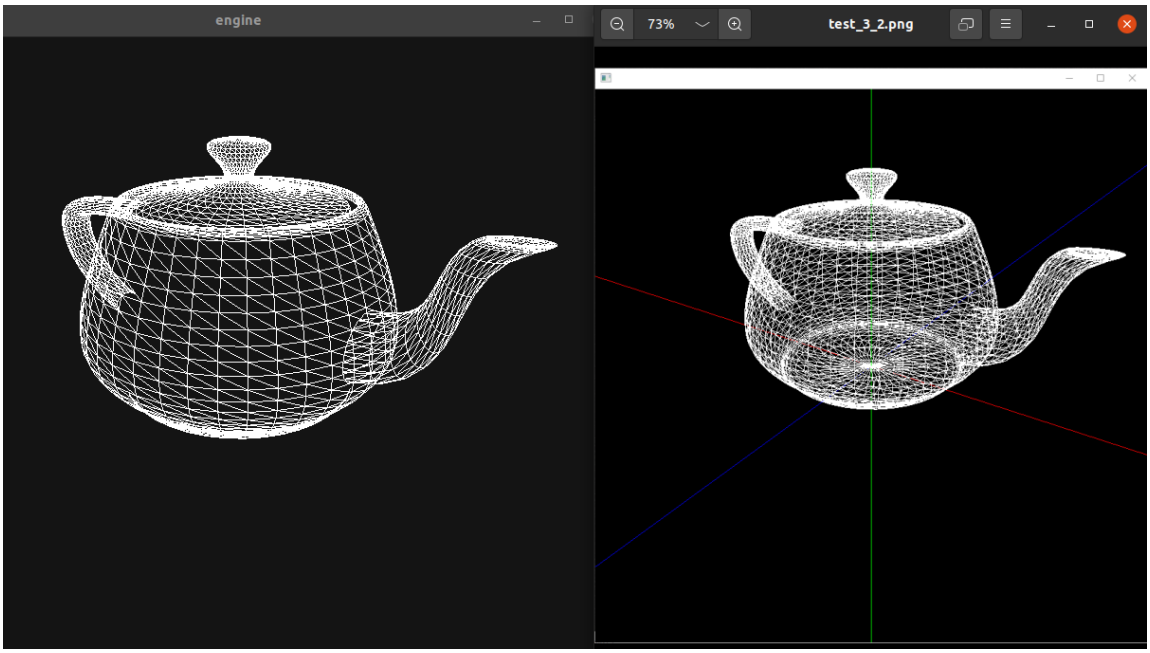

Figura 2. test_3_2.xml

Comparando as imagens produzidas pelo programa com as imagens fornecidas juntamente com os ficheiros XML, podemos concluir que foram atingidos os resultados pretendidos.

4.3 Sistema Solar Dinâmico

Como pedido no enunciado, segue-se a demo scene de um sistema solar dinâmico com a adição de um cometa que segue uma rota definida com curvas de Catmull-Rom:

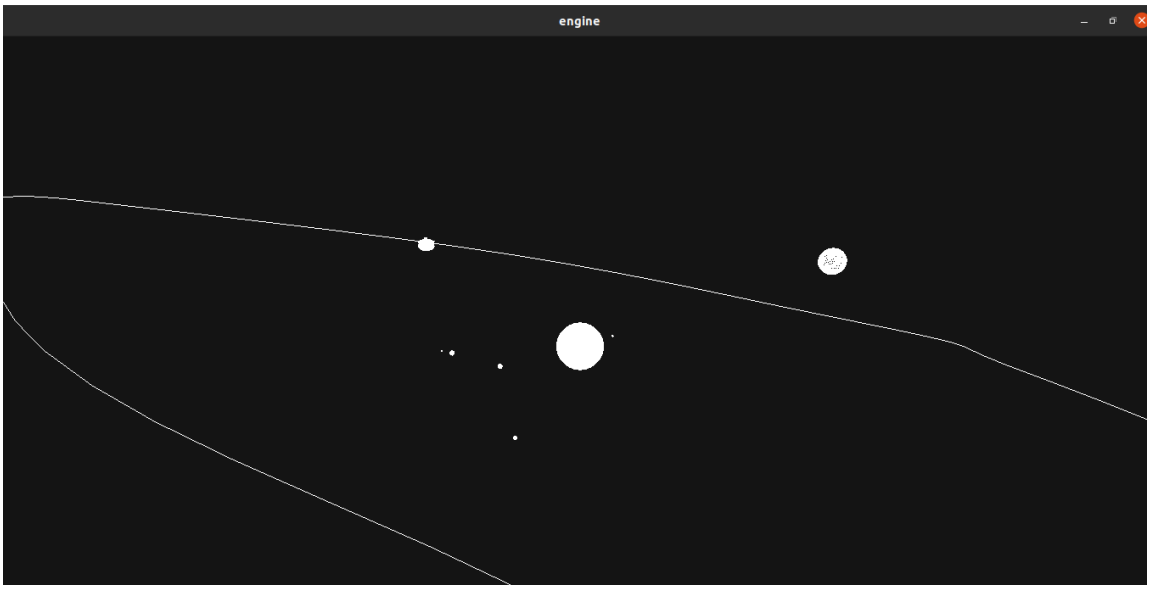

Figura 3. dinamic_solar_system.xml

5 Conclusão

O projeto e todos os testes foram desenvolvidos em Linux. Várias funcionalidades implementadas tiram partido de bibliotecas standard de C++20. Também recorremos às bibliotecas:

- **{fmt}**, que facilita operações de *input/output* com formatação;
- **glm**, que oferece tipos de dados e funções de matemática para OpenGL;
- **rapidxml**, para efetuar a leitura de ficheiros XML;
- **result**, para efetuar *value-based error handling*;
- **speedlog**, para imprimir informação de *logging*;
- **tinyobjloader**, para efetuar a leitura de ficheiros Wavefront .obj.

Finalizando este relatório, podemos afirmar que os problemas propostos para esta fase foram resolvidos com sucesso.

Além do proposto, corrigimos alguns problemas da fase anterior e implementamos funcionalidades extra.

Como trabalho futuro, segue-se a implementação de **Normals and Texture Coordinates**.