Universidade do Minho
Departamento de Informática

Computação Gráfica

Fase 3 - Grupo 59

26 de abril, 2024

Carlos Ferreira
a89509

Gerson Junior
a88000

Pedro Sousa
a100823

Pedro Viana
a100701

Sumário

1 Introdução 3
2 Engine 4
2.1 Desenhar com VBOs 4
2.2 Transformações Animadas 5
2.2.1 Timed Translate 5
2.2.2 Timed Rotate 9
3 Generator 10
3.1 Superfícies de Bézier 10
4 Demo 12
5 Conclusão 14

1 Introdução

Serve o seguinte relatório para documentar o processo de desenvolvimento da terceira fase do trabalho prático da unidade curricular de Computação Gráfica, do ano letivo 2023/2024.

Nesta fase a aplicação *Generator* deve ser capaz de criar modelos baseados em *patches* de Bézier e a aplicação *Engine* deve extender os elementos *translate* e *rotate* com o objetivo de realizar animações baseadas curvas cúbicas de Catmull-Rom.

Também foi necessário que os modelos fossem desenhados com VBOs.

2 Engine

2.1 Desenhar com VBOs

Nesta fase os modelos passaram a ser desenhados com *vertex buffer objects* (VBOs).

Foi optado usar *non-indexed rendering* por este ser mais simples e não requerer modificar o *Generator*.

Para implementar o uso de VBOs, cada modelo, para além de uma lista de vértices também contém um *unsigned int* que será usado como referência para o seu VBO.

Após a leitura de todos os modelos no ficheiro de configuração XML, é feito o *setup* do OpenGL e os VBOs para cada grupo são inicializados com o método `Group::initBuffers()` que por sua vez utiliza o método `Model::initBuffer()` em cada um dos modelos do grupo implementado da seguinte forma:

```
void Model::initBuffer()
{
    glGenBuffers(1, &buffer);
    glBindBuffer(GL_ARRAY_BUFFER, buffer);
    glBufferData(GL_ARRAY_BUFFER, (GLsizeiptr) (3 * vertices.size() * sizeof(float)),
vertices.data(), GL_STATIC_DRAW);
}
```

Para desenhar os modelos agora representados como VBOs foi necessário alterar o método de desenho destes para o seguinte:

```
void Model::draw() const
{
    glBindBuffer(GL_ARRAY_BUFFER, buffer);
    glVertexPointer(3, GL_FLOAT, 0, nullptr);
    glDrawArrays(GL_TRIANGLES, 0, (GLsizei) vertices.size());
}
```

Futuramente a gestão dos dados deverá ser melhorada para que a lista de vértices não se mantenha em memória depois desta ser usada para gerar o VBO assim como utilizar o mesmo VBO para modelos idênticos.

2.2 Transformações Animadas

Nesta fase os elementos *translate* e *rotate* do ficheiro de configuração XML foram extendidos de forma a permitir animações.

Para aplicar este tipo de transformações, o método `Transform::apply()` foi alterado para receber um `float` com o tempo decorrido em segundos desde o início do programa, `Transform::apply(float time)`. Este tempo é obtido através da expressão `glutGet(GLUT_ELAPSED_TIME) / 1000.0f`.

2.2.1 Timed Translate

O elemento *translate* pode agora receber um conjunto de pontos que definem uma curva cúbica Catmull-Rom sobre a qual o elemento será movido, o número de segundos para fazer estra translação, e especifica se o modelo deve estar alinhado com a curva.

Esta transformação foi ainda extendida pelo grupo para incluir a especificação se a curva deve ou não ser desenhada através do atributo booleano *draw*.

Segue um exemplo deste tipo de transformação:

```
<translate time = "10" align="true" draw="true">
  <point x = "0" y = "0" z = "4" />
  <point x = "4" y = "0" z = "0" />
  <point x = "0" y = "0" z = "-4" />
  <point x = "-4" y = "10" z = "0" />
</translate>
```

Esta nova transformação é representada no *Engine* pelo tipo `TimedTranslate`. Segue-se uma definição simplificada do estado deste:

```
class TimedTranslate : public Transform
{
    float time;
    bool align;
    bool draw;
    std::vector<Point3> points;
    std::vector<Point3> curve{};
    Vector3 yVector{0, 1, 0};
}
```

O atributo `yVector` representa o vetor sobre o qual o objeto é alinhado na curva e é inicializado com o vetor $(0, 1, 0)$. Este é

A lista de pontos `curve` representa a curva a desenhar caso o booleano `draw` seja verdadeiro. Esta curva contém cem vezes mais pontos que aqueles fornecidos no ficheiro de configuração XML de forma a desenhar uma curva mais suave e é definida no construtor do tipo que se segue:

```
TimedTranslate::TimedTranslate(float time, bool align, bool draw, std::vector<Point3>
points)
        : time(time), align(align), draw(draw), points(std::move(points))
{
    float gt = 0.0f;
    const unsigned long nSteps = this->points.size() * 100;
    const float tStep = time / (float) nSteps;

    for (int i = 0; i < nSteps; i++) {
        curve.push_back(getGlobalCatmullRomPoint(gt).first);
        gt += tStep;
    }
}
```

Este construtor dá uso à função `getGlobalCatmullRomPoint` que calcula o ponto e o vetor da tangente à curva nesse ponto para um dado "tempo global", o tempo decorrido desde o início do programa, que determina a posição do ponto ao longo da curva de Catmull-Rom.

A definição desta função foi fornecida pelos docentes da UC nas aulas TP pelo que o grupo apenas a adaptou para usar os tipos de dados do projeto. Esta, porém, utiliza a função `getCatmullRomPoint()` que foi definida pelo grupo.

Esta recebe os quatros pontos de controlo necessários para definir uma curva Catmull-Rom, e o um valor `t`, entre 0 e 1, que termina que ponto a calcular entre os pontos `p1` e `p2`. Segue-se o seu cabeçalho:

```
std::pair<Point3, Vector3> getCatmullRomPoint(float t, Point3 p0, Point3 p1, Point3 p2,
Point3 p3)
```

A função começa por definir a matriz de Catmull-Rom e a matriz das coordenadas dos quatro pontos fornecidos.

```
    static const 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},
    };

    const float p[4][3]{
            {p0.x, p0.y, p0.z},
            {p1.x, p1.y, p1.z},
            {p2.x, p2.y, p2.z},
            {p3.x, p3.y, p3.z},
    };
```

De seguida, são realizadas uma série de multiplicações matriciais que, na implementação do grupo, estão obfuscadas pela forma como os diferentes ciclos responsáveis por estas multiplicações foram combinados nas suas componentes comuns, levando a uma pequena melhoria na performance do *engine*.

Assim, passa-se a explicar a implementação das multiplicações recorrendo a código equivalente, que não combina os diferentes ciclos:

1. Multiplicar a matriz Catmull-Rom pela matriz dos pontos.

```
float a[4][3] = {0};
for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 3; j++) {
        for (int k = 0; k < 4; k++) {
            a[i][j] += m[i][k] * p[k][j];
        }
    }
}
```

2. Calcular o ponto na curva multiplicando a matriz resultante de (1) pelo vetor $(t^3, t^2, t, 1)$.

```
const float tv[4] = {powf(t, 3), powf(t, 2), t, 1};
float pv[3] = {0};
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 4; j++) {
        pv[i] += tv[j] * a[j][i];
    }
}
```

3. Calcular a derivada do ponto na curva multiplicando a matriz resultante de (1) pelo vetor $(3t^2, 2t, 1, 0)$.

```
const float tvd[4] = {3 * powf(t, 2), 2 * t, 1, 0};
float dv[3] = {0};
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 4; j++) {
        dv[i] += tvd[j] * a[j][i];
    }
}
```

Por fim, é devolvido um par com o ponto e o vetor da tangente do ponto na curva construídos a partir dos vetores calculados em (2) e (3), respetivamente:

```
return {{pv[0], pv[1], pv[2]}, {dv[0], dv[1], dv[2]}};
```

O método `apply` do tipo `TimedTranslate` é definido da seguinte forma:

```
void TimedTranslate::apply(float gt) noexcept
{
    if (draw) {
        drawCurve();
    }

    auto [pos, dir] = getGlobalCatmullRomPoint(gt);
    glTranslatef(pos.x, pos.y, pos.z);

    if (align) {
        Vector3 x = Vector3::normalize(dir);
        Vector3 z = Vector3::normalize(Vector3::cross(x, yVector));
        yVector = Vector3::normalize(Vector3::cross(z, x));

        glMultMatrixf(buildRotationMatrix(x, yVector, z).data());
    }
}
```

Este começa por determinar se deve ou não desenhar a curva.

Após isto obtém a posição e direção do modelo para o tempo atual (que deverá ser o tempo decorrido desde o início do programa) e aplica uma translação sobre as coordenadas de tal ponto.

Por fim, verifica se o modelo deve estar alinhado com o sentido da curva e, se tal for o caso, constrói e aplica a matriz de rotação calculada a partir do vetor direção calculado acima e do vetor y calculado até então.

2.2.2 Timed Rotate

O elemento rotate pode agora receber um atributo *time* ao invés de *angle* que define quantos segundos deve o modelo demorar a concluir uma rotação completa.

Segue um exemplo deste tipo de transformação:

```
<rotate time="3" x="0" y="1" z="0" />
```

Este nova transformação é representada no *Engine* pelo tipo `TimedRotate`. Segue-se uma definição simplificada do estado deste:

```
class TimedRotate : public Transform
{
    float anglePerSecond;
    Vector3 vector;
}
```

O seu construtor recebe o vetor sobre o qual a rotação deve ser executada e o número de segundos para a executar e, com este, calcula e guarda o ângulo que deve rodar por segundo.

```
TimedRotate::TimedRotate(float time, Vector3 vector) : vector(vector)
{
    anglePerSecond = (bool) time ? 360.0f / time : 0;
}
```

Este também verifica se o tempo de uma rotação é diferente de zero para evitar divisões por zero. Caso tal se verifique a rotação é nula.

O método `apply` do tipo `TimedRotate` é muito semelhante ao método `apply` do tipo `Rotate` divergindo apenas na necessidade de calcular o ângulo a rodar no momento da rotação, multiplicando o número de segundos decorridos desde o início do programa, pelo valor de "ângulo por segundo" que foi guardado no construtor. Segue-se a implementação do método `apply`:

```
void TimedRotate::apply(float gt) noexcept
{
    glRotatef(gt * anglePerSecond, vector.x, vector.y, vector.z);
}
```

3 Generator

3.1 Superfícies de Bézier

Com o objetivo de implementar um novo tipo de modelo definido com superfícies de Bézier foi desenvolvida a classe `PatchModel` que, tal como as primitivas já implementadas na primeira fase, extende a classe abstrata `Model`.

A classe `PatchModel` oferece dois construtores. O primeiro recebe o nome do ficheiro ".patch" e o segundo recebe a lista de listas de índices dos pontos de controlo que definem cada superfície e os próprios pontos de controlo. Ambos recebem ainda o nível de *tessellation* pretendido. O primeiro construtor, depois de carregar os dados do ficheiro ".patch" utiliza o segundo contrutor para gerar os vértices do módulo.

Segue-se o cabeçalho do primeiro construtor:

```
PatchModel::PatchModel(int tessellationLevel, const vector<array<int, 16>>
&indicesList, vector<Point3> controlPoints)
```

A estratégia de geração dos pontos escolhida aponta a ser o mais eficiente possível com foco no tempo de execução. Isto foi alcançado evitando cálculos redundantes, pré-calculando e reutilizando o máximo de dados possíveis.

Sendo TL o nível de *tessellation*, para cada superfície (lista de dezasseis índices) são feitos os seguintes passos:

1. Colecionar os dezasseis pontos de controlo da superfície:

   ```
   vector<Point3> patchControlPoints;
   for (const auto &index: indices) {
       patchControlPoints.push_back(controlPoints[index]);
   }
   ```

2. Calcular os $TL + 1$ vetores t a partir da matriz de Bézier:

   ```
   static const float bm[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}
   };

   vector<array<float, 4>> vectors;
   for (int i = 0; i <= tessellationLevel; i++) {
       float t = (float) i / (float) tessellationLevel;
       vectors.push_back(multMatrixVector(bm, {powf(t, 3), powf(t, 2), t, 1}));
   }
   ```

3. Calcular os $(TL + 1)^2$ pontos da superfície:

   ```
   vector<Point3> patchVertices;
   for (int i = 0; i <= tessellationLevel; i++) {
       for (int j = 0; j <= tessellationLevel; j++) {
           patchVertices.push_back(generatePoint(patchControlPoints, vectors[i],
   vectors[j]));
       }
   }
   ```

4. Popular a lista de vértices do modelo com os pontos calculados, por quad (a cada dois triângulos):

```
for (int i = 0; i < tessellationLevel; i++) {
    for (int j = 0; j < tessellationLevel; j++) {
        int a = i * (tessellationLevel + 1) + j;
        int b = a + tessellationLevel + 1;

        vertices.push_back(patchVertices[a]);
        vertices.push_back(patchVertices[a + 1]);
        vertices.push_back(patchVertices[b]);

        vertices.push_back(patchVertices[b]);
        vertices.push_back(patchVertices[a + 1]);
        vertices.push_back(patchVertices[b + 1]);
    }
}
```

A função `generatePoint()` usada no ponto 3 calcula as coordenadas de um ponto na superfície, dados os pontos de controlo e os vetores u e v nesse ponto, e é definida da seguinte forma:

```
Point3 generatePoint(vector<Point3> patchControlPoints, array<float, 4> um,
array<float, 4> vm)
{
    Point3 pointUV;
    for (int i = 0; i < 4; i++) {
        Vector3 ump;
        for (int j = 0; j < 4; j++) {
            ump += Vector3(patchControlPoints[j * 4 + i]) * um[j];
        }
        pointUV = pointUV + ump * vm[i];
    }
    return pointUV;
}
```

O passo 2 permite que, para cada superfície, sejam calculados apenas os únicos $TL + 1$ diferentes vetores t. Se este trabalho fosse feito no cálculo de cada ponto da superfície seria necessário calcular $2 * (TL + 1)^2$ vetores. É ainda evitado calcular duas listas de vetores, u e v, pois o n-ésimo vetor u é igual ao n-ésimo vetor v, pelo que, ao gerar o ponto (u,v) é apenas necessário usar o i-ésimo e o j-ésimo vetores t que corresponderão aos vetores u e v desse ponto, respetivamente.

Os passos 3 e 4 permitem calcular cada ponto da superfície apenas uma vez (passo 3) e posteriormente determinar os pontos de cada triângulo (passo 4).

A desvantagem desta estratégia é um elevado uso máximo de memória devido às listas intermediárias de dados que evitam cálculos redundantes.

4 Demo

Para demonstrar a capacidade de animar objetos, foi desenvolvido um demo de um sistema solar dinâmico (Figura 1).

Neste, o Sol gira sobre si mesmo e todos os corpos já representados no demo da fase dois giram também em torno de si mesmos e orbitam o Sol. Foi ainda adicionado um cometa que orbita o Sol ao longo de uma orbita elíptica sempre orientado no sentido da sua trajetória.

De forma a que os corpos consigam orbitar o Sol, após a primeira rotação estática que coloca cada corpo em períodos diferentes da órbita e antes da translação que coloca o corpo na órbita, é feita uma rotação dinâmica sobre o eixo do y.

Para que os corpos girem sobre si mesmos, a sua última transformação é também uma rotação dinâmica. Para que esta rotação não afete os corpos que orbitem o corpo a ser girado (planetas em relação ao Sol e luas em relação aos planetas), esta é feita num novo grupo.

Para que estas rotações sejam apreciáveis em tempo real, o tempo das órbitas foram calculados como o décimo do número de dias terrestres que o corpo demora a concluir uma órbita, em segundos. O tempo das rotações de cada corpo sobre si mesmo corresponde apenas ao número de dias terrestres em segundos.

Segue-se, como exemplo, a configuração da Terra e da Lua:

```
<!--Earth-->
<group>
    <transform>
        <rotate angle="18" x="0" y="1" z="0"/>
        <rotate time="36.5" x="0" y="1" z="0"/>
        <translate x="8" y="0" z="0"/>
        <scale x="0.3" y="0.3" z="0.3"/>
    </transform>
    <group>
        <transform>
            <rotate time="1" x="0" y="1" z="0"/>
        </transform>
        <models>
            <model file="sphere_1_32_32.3d"/>
        </models>
    </group>
    <!--Moon-->
    <group>
        <transform>
            <rotate angle="130" x="0" y="1" z="0"/>
            <rotate time="2.73" x="0" y="1" z="0"/>
            <translate x="3" y="0" z="0"/>
            <scale x="0.25" y="0.25" z="0.25"/>
            <rotate time="2.73" x="0" y="1" z="0"/>
        </transform>
        <models>
            <model file="sphere_1_32_32.3d"/>
        </models>
    </group>
</group>
```

O cometa usa como modelo o *utah teapot* definido com *patches* de Bézier. Este orbita a Terra através de uma translação dinâmica definida por dez pontos, ao longo de vinte segundos. A órbita em si também gira à volta do Sol ao longo de cinco minutos.

Segue-se a configuração deste modelo:

```
<!--Utah Comet-->
<group>
    <transform>
        <rotate time="300" x="0" y="1" z="0"/>
        <translate time="20" align="true" draw="true">
            <point x="2" y="-1" z="1"/>
            <point x="2" y="-1" z="-1"/>
            <point x="-1" y="0.5" z="-2.8"/>
            <point x="-6" y="3" z="-3.7"/>
            <point x="-11" y="5.5" z="-2.8"/>
            <point x="-14" y="7" z="-1"/>
            <point x="-14" y="7" z="1"/>
            <point x="-11" y="5.5" z="2.8"/>
            <point x="-6" y="3" z="3.7"/>
            <point x="-1" y="0.5" z="2.8"/>
        </translate>
        <scale x="0.1" y="0.1" z="0.1"/>
    </transform>
    <models>
        <model file="bezier_10.3d"/>
    </models>
</group>
```

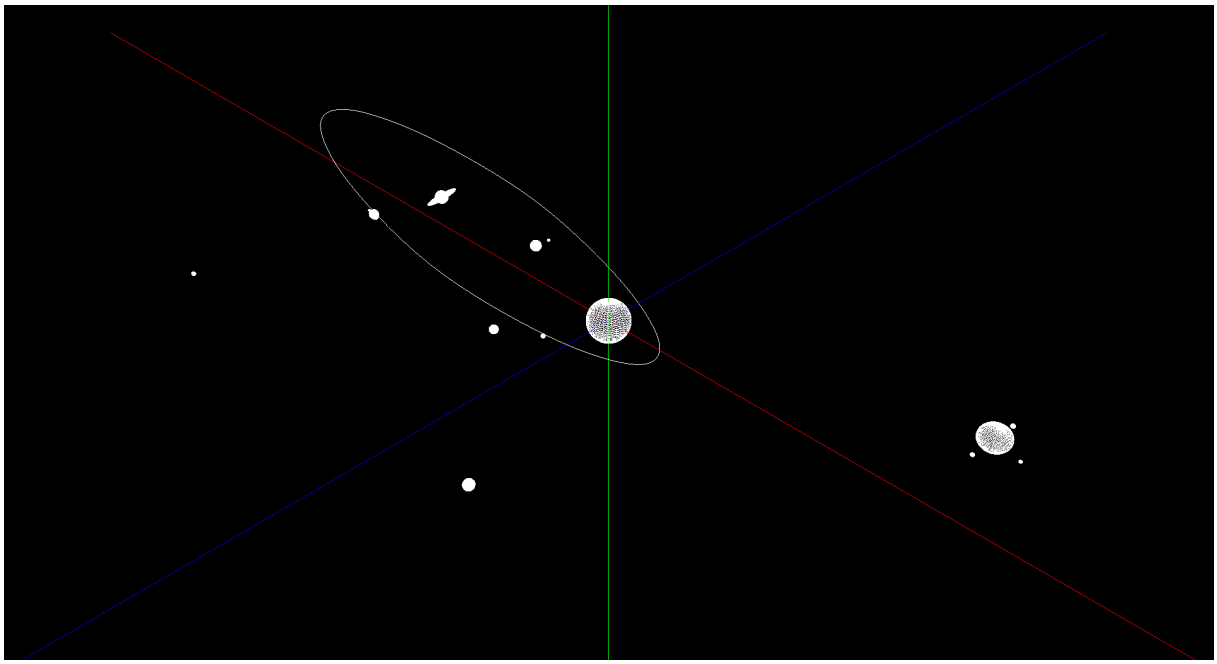

Figura 1: Sistema solar e cometa

5 Conclusão

Nesta fase do projeto foi possível identificar e desenvolver as estratégia necessárias para implementar os novos tipos de transformações dinâmicas, assim como gerar modelos a partir de ficheiros ".patch".

Considera-se que nesta fase foi possível consolidar o cálculo e a implementação do cálculo de curvas e superfícies de Bézier para definir modelos 3D e de curvas Catmull-Rom para definir animações.

Foi ainda possível consolidar o uso de VBOs, ainda que só tenha sido implementado uma versão simples, *non-indexed rendering*, e sem o esforço de evitar ter múltiplos buffers para o mesmo modelo, algo que esperamos melhorar na próxima fase.

Como o *demo* do sistema solar demonstra, é agora possível desenhar cenas animadas e construir modelos definidos com superfícies de Bézier.