Computação Gráfica (3º Ano de LCC)

Trabalho Prático

Relatório

Departamento de Informática

Tiago Pereira	Miguel Oliveira	Pedro Moura	Gonçalo Borges
A77504	A81942	A82258	A82457

8 de Maio de 2019

Resumo

Neste documento é descrito a realização do projeto proposto para U.C. de Computação Gráfica, do 3º ano da Licenciatura em Ciências da Computação.
Tem como objetivo o desenho de modelos utilizando a biblioteca gráfica openGL, a partir da leitura de um ficheiro de configuração escrito em XML, gerado previamente por um ficheiro gerador.

Conteúdo

Capítulo 1

Introdução

Este relatório descreve por etapas a realização do trabalho prático da disciplina de Computação Gráfica.
O trabalho encontra-se dividido em 2 aplicações: o **Gerador** e o **Motor**.
O gerador tem como função imprimir os vértices de uma figura, tal como o vetor normal de cada vértice e a sua coordenada de textura. Tem de ser capaz também de ler e interpretar outro ficheiro no caso excepcional das superfícies de Bezier.
Ao motor cabe interpretar o ficheiro gerado, e consequentemente desenhar a figura por ele representado.
Existem 6 tipos de figuras: plano, caixa, cone, esfera, anel e superfície de Bezier. Estas figuras são desenhadas com recurso à biblioteca gráfica OpenGL. É também necessária a capacidade de executar transformações geométricas, bem como iluminação e textura.

Capítulo 2

Solução

2.1 Primitivas

2.1.1 Plano

A primitiva *plano* ("Plane") é um quadrado no plano `XZ`, que está centrado na origem, e tem apenas 1 parâmetro: `size` (*tamanho*), que indica o comprimento dos lados do quadrado (em `float`).

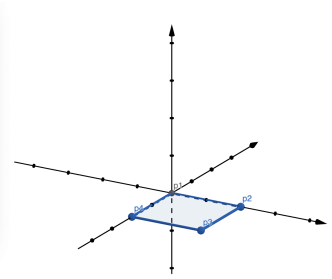

Figura 2.1: Plano

Com isto, podemos ver que o *plano* tem 4 vértices: `p1 = (0,0,0)`, `p2 = (size,0,0)`, `p3 = (size,0,size)` e `p4 = (0,0,size)`.
Como queremos que o *plano* se encontre centrado na origem, temos de alterar as coordenadas dos pontos de forma a que isto seja possível. Para isto, basta descobrir a metade do `size` dos lados do quadrado, e fazer uma espécie de "translação"deste valor tanto no eixo do `X` como no do `Z`.
Seja `s' = size / 2`, Podemos ver que o *plano* se encontra centrado, sendo os vértices: `p1 = (-s',0,-s')`, `p2 = (s',0,-s+)`, `p3 = (s',0,s')` e `p4 = (-s',0,s')`.

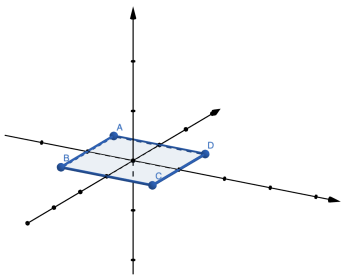

Figura 2.2: Plano Centrado

2.1.2 Caixa

A primitiva *caixa* ("Box") é um paralelípedo, centrado na origem, e tem 3 (opcionalemente 4) parâmetros: a dimensão `X`, a dimensão `Y` e a dimensão `Z` (opcionalmente, o número de divisões, que caso não seja passado, tem como valor *default* `1`).

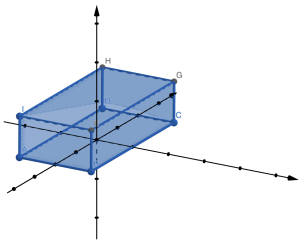

Figura 2.3: Caixa

É fácil visualizar que cada caixa tem 8 vértices.
O número de divisões de uma *caixa* é um valor `n`, tal que cada que cada face da caixa está dividida em `n * n` retângulos.
Para centralizar, a estratégia é semelhante à do plano. Descobrimos a metade das dimensões `X` e `Z`, e "movemos"a *caixa* de acordo com estes valores.

2.1.3 Cone

A primitiva *cone* ("Cone") está centrada na origem, e tem 4 parâmetros: `radius`, `height`, `slices` e `stacks`.

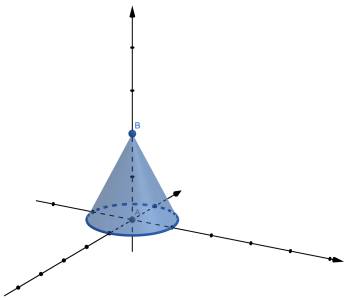

Figura 2.4: Cone

Sejam `sl` o número de slices e `st` o número de stacks.
A base do cone é um círculo com raio = `radius`. Como estamos a trabalhar com triângulos, este círculo é obtido construindo `sl` triângulos apontados para o centro.
É então necessário descobrirmos um ângulo α tal que cada triângulo tem este formato,

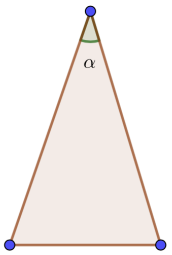

Figura 2.5: Triângulo

Este ângulo é calculado (em radianos) pela seguinte fórmula:

$$\alpha = 2\pi/slices.$$

Tendo o raio e o α, já temos tudo o que precisamos para construir a base.
A face lateral é construída stack a stack, onde por cada stack construímos `sl` trapézios. Cada stack tem uma altura h, que é calculada pela equação:

$$h = height/stacks$$

Cada ponto do trapézio é descoberto utilizando a seguinte fórmula:

$$P = (r' * sin\alpha', h', r' * cos\alpha')$$

onde r' é a distância do ponto ao eixo do `Y`, h' é a distância do ponto ao plano `XZ` e α é o ângulo do ponto em relação ao eixo do `Z`.
Se tivermos o trapézio seguinte,

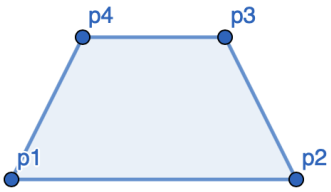

Figura 2.6: Trapézio

e seja $p1 = (r' * sin\alpha', h', r' * cos\alpha')$, então conseguimos encontrar os restantes pontos apartir deste.

$$p2 = (r' * sin(\alpha' + \alpha), h', r' * cos(\alpha' + \alpha))$$
$$p3 = (r' * sin(\alpha' + \alpha), h' + h, r' * cos(\alpha' + \alpha))$$
$$p4 = (r' * sin\alpha', h' + h, r * cos\alpha')$$

Sabendo descobrir as coordenadas de cada ponto, conseguimos construir os trapézios, ou mais especificamente, os triângulos que constroem os trapézios, e tendo isto, conseguimos fazer o cone.

2.1.4 Esfera

A primitiva *esfera* ("Sphere") está centrada na origem, e tem 3 parâmetros: `radius`, `slices` e `stacks`.

Começamos por construir os 2 polos da esfera. Estas 2 stacks são feitas separadamente porque , ao

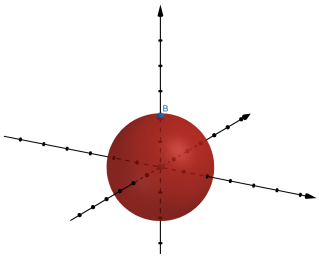

Figura 2.7: Esfera

contrário das restantes, são formadas por triângulos e não por trapézios. Nestes triângulos, o vértice da "ponta"situa-se no eixo do `Y`.
Seja t o triângulo da figura anterior, os seus vértices são:

1. Pólo Norte

$$p1 = (radius * cos(\beta + \beta') * sin(\alpha), radius * sin(\beta + \beta'), radius * cos(\beta + \beta') * cos(\alpha))$$
$$p2 = (0, -radius, 0)$$
$$p3 = (radius * cos(\beta + \beta') * sin(\alpha + \alpha'), radius * sin(\beta + \beta'), radius * cos(\beta + \beta') * cos(\alpha + \alpha'))$$

2. Pólo Sul

$$p1 = (radius * cos(\beta) * sin(\alpha), radius * sin(\beta), radius * cos(\beta) * cos(\alpha))$$
$$p2 = (0, radius, 0)$$
$$p3 = (radius * cos(\beta) * sin(\alpha + \alpha'), radius * sin(\beta), radius * cos(\beta) * cos(\alpha + \alpha'))$$

No que toca à face lateral, a estratégia é semelhante à do cone. Construímos stack a stack, e por cada stack construímos trapézios de quantidade igual ao número de slices. Os pontos seguem o mesmo formato do trapézio do cone, mas agora temos de ter em atenção o ângulo β.
Vejamos novamente o trapézio da figura 11. Se $p1 = (r * cos(\beta) * sin(\alpha), r * sin(\beta'), r * cos(\beta') * cos(\alpha)$, então

$$p2 = (r * cos(\beta') * sin(\alpha' + \alpha), r * sin(\beta'), r * cos(\beta') * cos(\alpha' + \alpha))$$
$$p3 = (r * cos(\beta' + \beta) * sin(\alpha' + \alpha), r * sin(\beta' + \beta), r * cos(\beta') * cos(\alpha' + \alpha))$$
$$p4 = (r * cos(\beta' + \beta) * sin(\alpha'), r * sin(\beta' + \beta), r * cos(\beta') * cos(\alpha'))$$

onde α' e β' são os ângulos dos pontos em relação ao referencial e r o raio.

2.1.5 Anel

A primitiva *anel* tem 3 parâmetros: `radius`, `width` e `slices`.
A sua construção é semelhante à base do cone, só que temos de ter em atenção a `width` do anel.
Mais uma vez, a estratégia passa por construir trapézios, da seguinte forma:

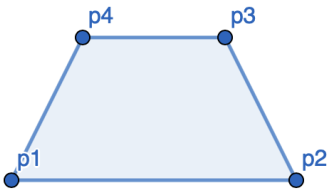

Figura 2.8: Trapézio

Seja $r' = radius - width$, Temos que

$$p1 = (radius * sin\alpha', 0, radius * cos\alpha')$$
$$p2 = (radius * sin(\alpha' + \alpha), 0, radius * cos(\alpha' + \alpha))$$
$$p3 = (r' * sin(\alpha' + \alpha), 0, r' * cos(\alpha' + \alpha))$$
$$p4 = (r' * sin\alpha', 0, r * cos\alpha')$$

Iterando esta fórmula sobre o número de `slices`, temos como o resultado o seguinte:

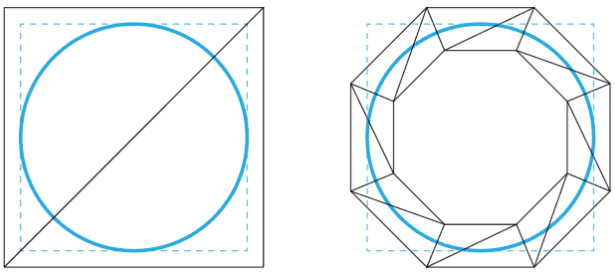

Figura 2.9: Anel

2.1.6 Superficie de Bezier

Ao contrário das primitivas anteriores, a superfície de Bezier recebe como parâmetros o nome de um ficheiro, onde se encontram definidos os pontos de controlo de Bezier, e o nível de tesselação esperado. Com isto, tal como as primitivas anteriores, gera um ficheiro que contêm os vértices dos triângulos para desenhar a superfície.
Uma superfície de Bezier é apenas uma extensão da ideia de curva de Bezier. Só que em vez de termos, por exemplo, 4 pontos, a superfície é definida por 16 pontos que podem ser vistos como uma grelha de 4x4 pontos de controlo. (De notar que nesta fase apenas serão utilizadas superfícies bicúbicas, ou seja, que têm 16 pontos de controlo.)
Ora, nas curvas tínhamos um parâmetro que se movia ao longo da curva (`t`). Agora, teremos 2 parâmetros, `u` e `v`, que se irão mover em direções diferentes (ver figura). Tal como o `t`, o valor destes 2 parâmetros está contido entre `0` e `1`.

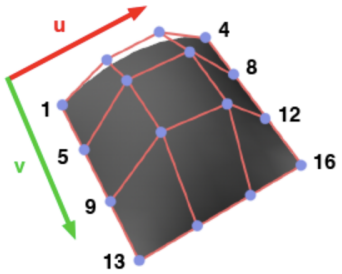

Figura 2.10: Superfície de Bezier e os seus pontos de controlo

Um ponto na superfície de Bezier é dado pela seguinte fórmula:

$$P(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} B_i^n(u)B_j^m(v)P_{ij}$$

onde,

$$B_i^n(u) = \binom{n}{i} u^i(1-u)^{n-i}$$

é o polinómio de Bernstein, e

$$\binom{n}{i} = \frac{n!}{i!(n-i)!}$$

é o coeficiente binomial.

Como foi referido anteriormente, apenas serão estudadas curvas bicúbicas, ou seja, $n = m = 3$. Visto isto, podemos dizer que a nossa equação final é:

$$P(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} B_i^3(u)B_j^3(v)P_{ij}$$

Só nos resta saber o que é a tesselação.
A tesselação é apenas um fator que determina o quão detalhada (ou suave) queremos que seja a nossa

superfície. Usamos-la para descobrir o fator de iteração dos parâmetros u e v. Este fator é calculado da seguinte forma:

$$f = \frac{1}{tessellation}$$

2.2 Curvas de Catmull-Rom

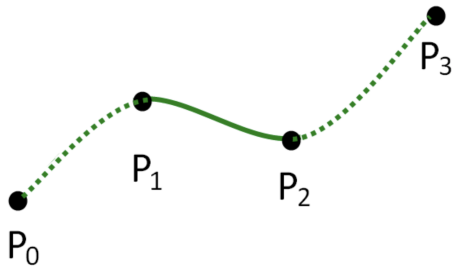

Figura 2.11: Curva de Catmull-Rom

Uma curva de Catmull-Rom, tal como uma curva de Bezier, tem um parâmetro `t` que se move ao longo da curva, e varia entre `0` e `1`.
O cálculo de um ponto na curva é dado pelo seguinte algoritmo:

$$P(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

De notar que este algoritmo funciona apenas para curvas com 4 pontos, ou mais especificamente, curvas cúbicas.
Visto isto, é necessária a criação de um algoritmo "global"que possibilite o desenho de curvas com mais pontos. A estratégia passa por definirmos uma variável global, à qual chamaremos `gt`, que tem a mesma função que o `t`, mas percorre a curva inteira.
Agora que temos esta nova variável, é possível descobrir o ponto de controlo anterior ao ponto na curva onde se encontra a variável. Com isto, sabemos qual o ponto de controlo anterior, e quais os 2 pontos de controlo seguintes, tendo assim 4 pontos.
Agora que temos os 4 pontos de controlo, podemos executar o algoritmo acima descrito, encontrando assim o ponto pretendido.

Capítulo 3

Implementação da Solução

3.1 Gerador

O **gerador** é um programa que recebe como *input* o nome da primitiva e os argumentos necessários para gerar os seus pontos.
Existem 2 funções envolvidas na criação do ficheiro 3d. Uma que se assegura que o número de argumentos está correto, e a outra que escreve para o ficheiro passado como parâmetro o número de vértices e os vértices. Esta última implementa o que foi explicado no capítulo 2, através de ciclos, tendo o cuidado de representar todos os polígonos apresentados como triângulos. (Ex: Um trapézio corresponde a dois triângulos). Na última fase do trabalho prático, é também necessário imprimir os vetores normais de cada vértice e as suas coordenadas de textura.

3.1.1 Superficie de Bezier

Para ser possível o desenho dos vértices de uma superfície de Bezier, é primeiro necessário a interpretação do ficheiro com as informações relativas às superfícies.
O ficheiro tem a seguinte estrutura:

- A primeira linha contêm o número de superfícies que serão desenhadas;
- As linhas seguintes, uma para cada superfície, contêm os índices dos pontos de controlo (16 para cada superfície);
- A linha seguinte indica o número total de pontos de controlo;
- E por último, as linhas restantes representam os pontos de controlo (1 por linha).

De forma a simplificar a interpretação do ficheiro, foram criadas as seguintes estruturas, `Point` e `Patch`:

```
1 struct Point {
2         float x;
3         float y;
4         float z;
5
6         Point(float x, float y, float z) {
7                 this->x = x;
8                 this->y = y;
9                 this->z = z;
```

```
10          }
11
12          string toString() {
13                  return "(" + to_string(this->x) + ","
14                  + to_string(this->y) + ","
15                  + to_string(this->z) + ")";
16          }
17  };
18
19
20  struct Patch {
21      vector<int> indices;
22  };
```

A estratégia utilizada começa por guardar os índices de cada superfície num vetor de inteiros relativo à sua superfície.

```
1  vector<Patch> patches;
2  char delimiter = ',';
3  for(int i = 0; i < number_of_patches; i++) {
4          Patch p;
5
6          string indices;
7          getline(in_file, indices);
8
9          string token1;
10         istringstream token_stream1(indices);
11
12         while(getline(token_stream1, token1, delimiter)) {
13                 p.indices.push_back(stoi(token1));
14         }
15
16         patches.push_back(p);
17 }
```

Depois disto, é criado um vetor de `Pontos`, que contem todos os pontos de controlo do ficheiro.

```
1  vector<Point> control_points;
2  for(int i = 0; i < number_of_control_points; i++) {
3          Point a = Point(0,0,0);
4
5          string coordinates;
6          getline(in_file, coordinates);
7
8          string token2;
9          istringstream token_stream2(coordinates);
10
11         // x
12         getline(token_stream2, token2, delimiter);
13         a.x = stof(token2);
14
15         // y
16         getline(token_stream2, token2, delimiter);
```

```
17          a.y = stof(token2);
18
19          // z
20          getline(token_stream2, token2, delimiter);
21          a.z = stof(token2);
22
23          control_points.push_back(a);
24  }
```

Com os vetores de índices e o vetor de pontos de controlo, podemos construir os vetores com os pontos de controlo de cada superfície, e consequentemente desenhar a superfície, ou neste caso, escrever para o ficheiro os vértices.

```
1  for(Patch p : patches) {
2          // criar o vetor de pontos final do patch
3          vector<Point> fpoints;
4          for(int index : p.indices) {
5                  fpoints.push_back(control_points[index]);
6          }
7
8          // desenhar o patch
9          outputBezierPatchVertices(fpoints, tessellation, 3, 3, fildes);
10 }
```

O cálculo dos vértices é apenas uma implementação da fórmula acima descrita, e é executado pela seguinte função:

```
1  Point bezierPatch(vector<Point> points, float u, float v, int n, int m) {
2
3          Point pf = Point(0,0,0);
4
5          int i, j;
6          for(i = 0; i <= n; i++) {
7                  for(j = 0; j <= m; j++) {
8                          Point p = points[i * (n+1) + j];
9                          float bi = bernsteinPol(n, i, v);
10                         float bj = bernsteinPol(m, j, u);
11
12                         pf.x += bi * bj * p.x;
13                         pf.y += bi * bj * p.y;
14                         pf.z += bi * bj * p.z;
15                 }
16         }
17
18         return pf;
19 }
```

3.1.2 Normais

No que toca à iluminação, tudo o que é relativo ao cálculo de normais encontra-se aqui, no gerador. Implementámos o cálculo das normais para todas as primitivas exceto as superfícies de Bezier.

Plano

As normais nesta primitiva são imediatas. Como o plano é desenhado no plano $Y = 0$, temos apenas uma normal: $(0, 1, 0)$, que é a normal de todos os vértices.

Box

Temos 6 normais diferentes nesta primitiva, uma por cada face da caixa.

- $(1, 0, 0)$ para a face X;
- $(-1, 0, 0)$ para a face $-X$;
- $(0, 0, 1)$ para a face Z;
- $(0, 0, -1)$ para a face $-Z$;
- $(0, 1, 0)$ para a face Y;
- $(0, -1, 0)$ para a face $-Y$;

Cone

Como o cone está desenhado no plano $Y = 0$, as normais dos vértices da base é $(0, -1, 0)$. As normais da face lateral são calculadas pelo produto vetorial.

Esfera

Como a esfera está centrada na origem, não é necessário o uso do produto vetorial para o cálculo do vetor normal a um ponto P. Apenas basta dividir o vetor da origem a P pelo raio.

$$\vec{n} = \frac{P - O}{raio}$$

Anel

As normais do `anel` são semelhantes às do plano, com a diferença que o anel é desenhados dos dois lados. Como tal como o plano é desenhado no plano Y, as suas normais são: $(0, 1, 0)$ e $(0, -1, 0)$.

3.1.3 Texturas

As texturas apenas são calculadas para a primitiva `esfera`. O mapeamento de um ponto P para coordenadas UV é feito pelas seguintes fórmulas:

$$u = \frac{\alpha}{2\pi} \qquad v = \frac{\beta}{\pi} + 0.5$$

sendo α e β os ângulos que foram utilizados para calcular o ponto previamente.

3.2 Motor

Em concordância com o enunciado, é necessária a criação de uma aplicação que seja capaz de interpretar ficheiros de configuração **XML** com o correspondente formato.

Denomeamos esta aplicação de **Motor** e opera lendo um nome de ficheiro, passado como primeiro argumento na execução do programa, e interpretando conforme o pedido.

O **Motor** lê o ficheiro **XML** iterando todas as *tags* `model` dentro da *tag* `scene` que contêm o nome do ficheiro 3d gerado pelo **Gerador** no atributo `file` e de seguida criando um objeto `Model` e, através do método `Model::Import`, o **Motor** lê e armazena os diferentes vértices numa variável de instância da classe `Model`, `m_vertices`.

Após esta fase inicial de interpretação e armazenamento, o **Motor**, com recurso às bibliotecas **GLUT** e **OpenGL**, inicializa uma cena 3d e renderiza os vértices, recorrendo ao método `Model::Render`, respeitando sempre a obrigação de desenhar apenas triângulos, ou seja, desenhando 3 vértices de cada vez com a função do **OpenGL** `glVertex3f`.

Era necessário a criação de uma estrutura de dados para armazenar a informação pertinente a cada grupo e também uma estrutura de dados que descrevesse e identificasse cada uma das diferentes transformações geométricas.

Tendo isto em conta, criamos a classe base GeoTransform que contém o tipo da transformação e 3 variáveis floats x,y,z que são caracteristicas de todas as transformações geométricas que era necessário implementar.

A transformação geométrica de Rotação requer um campo extra, o angle, para representar quanto é efetuada a Rotação.

Cada transformação geométrica é uma classe derivada da classe base cada uma com a sua especificação de tipo para sermos capaz de as identificar.

Criamos também a classe Group, que contém um vetor de transformações e diversos métodos para obter as respetivas transformações geométricas e o respetivo grupo pai.

Tendo isto, faltava apenas a interpretação das novas tags, que fizemos nas funções parse_group_tag e outras funções semelhantes para interpretar as tags das transformações geométricas.

Depois de serem interpetadas e armazenadas as tags dos Grupos, restou apenas atribuir a cada Modelo o grupo em qual se integra, e modificar a função Render para ter em conta o seu grupo e todos os grupos pais, e aplicar as transformações geométricas de cada grupo, começando pelas transformações geométricas dos pais até, por fim, aplicar as transformações geométricas do grupo do próprio Modelo.

3.2.1 VBO's

Os **VBO**'s são inicializados pelo seguinte pedaço de código,

```
1         // assign vbos to each model
2         size_t n = models.size();
3         std::vector<GLuint> temp_vbos(n);
4         glGenBuffers(n, temp_vbos.data());
5
6         for (auto i = 0u ; i < n; i++) {
7                 models[i].SetVBO(temp_vbos[i]);
8         }
```

Podemos ver que a cada modelo é atribuído um **VBO** diferente, e que estes, depois de inicializados, são preenchidos com a informação relativamente ao modelo, através da função `SetVBO`.

```
1         void Model::SetVBO(GLuint index) {
2                 m_vbo = index;
3                 glBindBuffer(GL_ARRAY_BUFFER, m_vbo);
4                 glBufferData(GL_ARRAY_BUFFER, m_vertices.size() *
5                         sizeof(TVertexData), m_vertices.data(), GL_STATIC_DRAW);
6         }
```

3.2.2 Curvas de Catmull-Rom

Para ser possível a animação das primitivas ao longo das curvas de Catmull-Rom, é primeiro necessária a interpretação do ficheiro `XML` em relação às novas *tags*.
Quando é encontrada uma *tag* de translação com o atributo `time`, que representa o tempo que o modelo demora a efetuar a translação ao longo da curva, o motor dá *parsing* às *tags* de pontos dentro da *tag* de translação, que representam os pontos de controlo da curva.

```
if (time) {
        t->time = std::stof(time);
        t->translation_type = TTranslate::DYNAMIC_TRANSLATE;
        if (!parse_points(e, t->translation_points))
        {
                t->type = GT_INVALID;
        }
        return t;
}
```

Aqui temos a função `parse_points`, que dá preenche o vetor de pontos de controlo da curva.

```
bool parse_points(XMLElement* elem, std::vector<Point>& points) {
        for (XMLElement* e = elem->FirstChildElement("point"); e != NULL; e = e->
            NextSiblingElement("point"))
        {
                Point p;
                points.push_back(p);
        }

        return true;
}
```

Agora que temos um vetor com todos os pontos de controlo da curva, é possível encontrar qualquer ponto, e com isto executar translações consecutivas ao modelo, criando assim a animação desejada.
A função que calcula um ponto global da curva é a implementação da estratégia acima discutida.

```
        // given  global t, returns the point in the curve
        void getGlobalCatmullRomPoint(float gt, std::vector<Point> p, float *pos,
            float *deriv) {

        size_t POINT_COUNT = p.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;

        float* ps[4] = {
                p[indices[0]].toVector(),
```

```
                p[indices[1]].toVector(),
                p[indices[2]].toVector(),
                p[indices[3]].toVector(),
        };

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

Sendo que a função `getCatmullRomPoint` é apenas a implementação do algoritmo de Catmull-Rom.

```
void getCatmullRomPoint(float t, float *p0, float *p1, float *p2, float *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 quad = t * t;
        float cubo = t * t * t;

        float P[4][4] = { {p0[0], p0[1], p0[2], p0[3]},
                {p1[0], p1[1], p1[2], p1[3]} ,
                {p2[0], p2[1], p2[2], p2[3]} ,
                {p3[0], p3[1], p3[2], p3[3]}
        };


        float C[4];
        float T[4] = { cubo, quad, t, 1 };

        // C = m * T
        multMatrixVector(m, T, C);

        // pos = C * P
        multMatrixVector(P, C, pos);

        float D[4];
        float Tdash[4] = { 3 * quad,  2 * t, 1, 0 };

        // D = m * T'
        multMatrixVector(m, Tdash, D);

        // deriv = D * P
        multMatrixVector(P, D, deriv);
}
```

3.2.3 Iluminação

Foi necessário atualizar o motor de forma a que fosse possível ler as *tags* de luz, que podem ser dos seguintes tipos:

- **POINT** - que também recebe a posição da luz;
- **DIRECTIONAL** - recebe a direção da luz;
- **SPOT** - recebe a posição da luz, a direção da luz e o cone da luz;

Foram alteradas algumas funções e estruturas de fases anteriores como o caso da estrutura "Model" para suportar as diferentes luzes.
Foi necessário também adicionar 4 componentes (floats) para cada tipo de luz (RGBA).
Infelizmente, o motor não consegue suportar várias luzes ao mesmo tempo, e por essa razão não conseguimos produzir exemplos bem iluminados.

Normais

A classe Modelo recebe um VBO para guardar as normais, que foram lidas do ficheiro gerado pelo gerador. Isto é necessário para o funcionamento correto da iluminação.

3.2.4 Texturas

Para a aplicação de texturas, foi necessária também a criação de um VBO na classe Modelo, proveniente do ficheiro .3d. Foi criada a função *setTexture* para carregar e atribuir uma estrutura a um modelo.
Ambas as implementações das normais e das texturas envolveram funções de OpenGL, na função render dos modelos para serem carregados para buffers do OpenGL, e consequentemente serem renderizadas.

Capítulo 4

Conclusão

Depois de concluído, olhamos para este projeto de forma positiva.
Apesar das partes iniciais do trabalho serem executadas com sucesso, existiram algumas dificuldades nesta fase final, mais nomeadamente na iluminação. De resto achamos que os objetivos globais foram alcançados com sucesso.
Desta forma, com o desenvolvimento deste trabalho consideramos que o nosso conhecimento em relação à disciplina e àrea de Computação Gráfica foi enriquecido, bem como a utilização das diferentes ferramentas usadas, como por exemplo o OpenGL ou o GLUT.