Universidade do Minho
Escola de Engenharia

Computação Gráfica
Fase 3 - Grupo 11
2022/2023

Afonso Xavier Cardoso Marques a94940
Ana Filipa da Cunha Rebelo a90234
Tomás Cardoso Francisco a93193
Simão Paulo da Gama Castel-Branco e Brito a89482

Maio 2023




Índice

1 Introdução **4**

2 Generator **4**
2.1 Leitura dos Benzier patches 4
2.2 Cálculo dos pontos 5

3 Engine **7**
3.1 Alterações efetuadas ao código existene. 7
3.1.1 O novo transformation.h 7
3.1.2 Novo conteúdo do parser 8
3.2 Transformações com Tempo 8
3.2.1 Translação 8
3.2.2 Rotação 8
3.3 Implementação de VBOs 9

4 Conclusão **10**

List of Figures

1 Fórmula de Bernstein 5
2 Excerto de código da função generateBenzier 5
3 cálculo dos pontos 6
4 Algoritmo de Bezier 6

1 Introdução

A terceira fase do projeto teve como objetivo obter uma representação dinâmica do sistema solar, tornando-o mais próximo da realidade. Para tal foi necessária a implementação de novas funcionalidades, associadas ao Generator e ao Engine desenvolvidos nas fases anteriores, tais como curvas, superfíces cúbicas e VBOs. No Generator, foi nos proposta a conversão de um dado ficheiro em ficheiro 3d, a partir de um conjunto de patches de bezier e o nível de tecelagem.

Quanto ao Engine, foi nos pedida a implementação de transformações com tempo. A translação deve ser realizada através da curva cúbica, Catmull-Rom, enquanto que na rotação é possível efetuar uma rotação de 360 graus em torno de um dado eixo. Cada uma destas transformações ocorrem num determinado tempo. Foi-nos também pedida a alteração do modo de desenho da primitivas para VBOs.

No presente relatório, apresentamos e explicamos em detalhe todas as funcionalidades pedidas no enunciado de modo a ilustrar o raciocínio usado.

2 Generator

No generator foi adicionada a leitura de patches de Benzier, guardadas num ficheiro, bem como a sua transformação para pontos e a respetiva escrita num ficheiro 3d. De modo a ser possível armazenar e manipular os dados da curva de Bezier de forma eficiente e organizada implementamos a struct bezier_storage.

```
struct bezier_storage {
        int n_patches;
        vector<int> indices_por_patch;
        int n_control_points;
        vector<vector<float>> pontos_controlo;
};
```

Esta struct contém n_patches que representa o número de patches da curva de benzier, indices_por_patch que contém uma lista de índices para cada patch, n_control_points que representa o número de pontos de controlo e pontos_controlo que contém os pontos de controlo da curva de benzier.

2.1 Leitura dos Benzier patches

Para a leitura do ficheiro 'teapot.patch' recorremos à função generateBenzier que guarda as informações lidas na struct bezier_storage. O preenchimento na struct é feito através da leitura linha a linha do ficheiro. A função generateBenzier começa por calcular os coeficientes da curva de Bezier usando a fórmula de Bernstein.

$$B_{i,3} = \binom{3}{i} t^i (1-t)^{3-i}$$

Figure 1: Fórmula de Bernstein

Esta fórmula é usada para calcular os pesos de cada ponto de controlo em cada um dos pontos da curva. Os coeficientes de Bernstein são calculados uma única vez, antes do loop que gera os pontos da curva.

2.2 Cálculo dos pontos

O cálculo dos pontos foi feito no excerto de código apresentado na figura 2. Este excerto cria diferentes matrizes para cada coordenada x, y ou z de todos os pontos. De seguida é aplicada a função getBenzierPoint, apresentada na figura 3, em função da tecelagem, sendo esta incrementada sucessivamente. Esta segue o algoritmo de Bezier apresentado na figura 4.

```
for (int s = 0; s < bs.indices_por_patch.size(); s += 16) {

        for (size_t a = 0; a < 4; a++) {

                for (size_t b = 0; b < 4; b++) {
                        matrixX[a][b] = bs.pontos_controlo.at(bs.indices_por_patch.at(s + a * 4 + b))[0];
                        matrixY[a][b] = bs.pontos_controlo.at(bs.indices_por_patch.at(s + a * 4 + b))[1];
                        matrixZ[a][b] = bs.pontos_controlo.at(bs.indices_por_patch.at(s + a * 4 + b))[2];
                }
        }

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

                for (int j = 0; j < tesselation; j++) {
                        counter++;
                        u = t * i;
                        v = t * j;
                        float next_u = t * (i + 1);
                        float next_v = t * (j + 1);

                        getBezierPoint(u, v, (float**)matrixX, (float**)matrixY, (float**)matrixZ, pos[0]);
                        getBezierPoint(u, next_v, (float**)matrixX, (float**)matrixY, (float**)matrixZ, pos[1]);
                        getBezierPoint(next_u, v, (float**)matrixX, (float**)matrixY, (float**)matrixZ, pos[2]);
                        getBezierPoint(next_u, next_v, (float**)matrixX, (float**)matrixY, (float**)matrixZ, pos[3]);
```

Figure 2: Excerto de código da função generateBenzier

```
void getBezierPoint(float u, float v, float** matrixX, float** matrixY, float** matrixZ, float* pos) {
        float bezierMatrix[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 } };

        float vetorU[4] = { u * u * u, u * u, u, 1 };
        float vetorV[4] = { v * v * v, v * v, v, 1 };

        float vetorAux[4];
        float px[4];
        float py[4];
        float pz[4];

        float mx[4];
        float my[4];
        float mz[4];

        multMatrixVector((float*)bezierMatrix, vetorV, vetorAux);
        multMatrixVector((float*)matrixX, vetorAux, px);
        multMatrixVector((float*)matrixY, vetorAux, py);
        multMatrixVector((float*)matrixZ, vetorAux, pz);

        multMatrixVector((float*)bezierMatrix, px, mx);
        multMatrixVector((float*)bezierMatrix, py, my);
        multMatrixVector((float*)bezierMatrix, pz, mz);


        pos[0] = (vetorU[0] * mx[0]) + (vetorU[1] * mx[1]) + (vetorU[2] * mx[2]) + (vetorU[3] * mx[3]);
        pos[1] = (vetorU[0] * my[0]) + (vetorU[1] * my[1]) + (vetorU[2] * my[2]) + (vetorU[3] * my[3]);
        pos[2] = (vetorU[0] * mz[0]) + (vetorU[1] * mz[1]) + (vetorU[2] * mz[2]) + (vetorU[3] * mz[3]);

}
```

Figure 3: cálculo dos pontos

$$B(u,v) = [u^3 \quad u^2 \quad u \quad 1]M \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} M^T \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix}$$

Figure 4: Algoritmo de Bezier

3 Engine

No Engine, o objetivo foi expandir os elementos *translate* e *rotate*, sendo que a translação pode agora ter um conjunto de pontos que definem uma curva cúbica de Catmull-Rom, onde o modelo pode alinhar com dita curva, juntamente com o número de segundos que demora a percorrer a curva, e a rotação, em vez de ter um ângulo, pode ter uma quantidade de segundos que indica quanto tempo demora a realizar a rotação completa de 360 graus. De tal forma, a seguinte secção irá explicar algumas alterações feitas ao código das fases anteriores, juntamente com as novas secções exclusivas a esta fase.

3.1 Alterações efetuadas ao código existene.

3.1.1 O novo transformation.h

Originalmente, um *group* ou grupo era constituído por quatro elementos: um vetor de inteiros que serve para saber a ordem das instruções, um vetor de vetores de structs chamado de *transformations*, para representar as transformações geométricas, um vetor de vetores de strings para guardar o nome dos ficheiros para os modelos a desenhar e um vetor de outros grupos que representam as cenas adicionais. Embora muitos ficaram iguais, o struct de transformação foi transformado na sua própria classe, guardada no ficheiro transformation.h.

Esta nova classe é constituída por uma estrutura bastante simples, sendo constituída maioritariamente por funções *getters,setters* e *construtors*. Em termos de variáveis, existem algumas novas que temos de manter:

- **type** - Uma *string* que serve para identificar que tipo de transformação se trata.
- **angle** - Um *float* para as rotações que representam o ângulo.
- **time, timeNow** - Variáveis *float* que servem para as novas transformações que utilizam tempo. *time* serve para guardar o tempo lido no parser e *timeNow* para representar o tempo atual, ou tempo de vida.
- **align** - Uma *string* que serve para a nova translação e indica ao programa se o modelo deve ser alinhado com a linha Catmull-Rom que está a percorrer.
- **var_x, var_y, var_z** - Variáveis *float* que servem como variáveis gerais para as transformações, explicando a movimentação ou efeitos nos eixos x, y, z.
- **prev_y** - Variável *vector¡float¿* que serve como vetor iterativo para usar na translação e garantir que o modelo se encontra alinhado com a linha.
- **catmullPoints** - Variável que representa os pontos usados para criar a curva de Catmull, daí o nome, **catmullPoints**.

3.1.2 Novo conteúdo do parser

As funções do parser agora reconhecem os novos casos de translação e de rotação. Na condicional das funções, é analisado se os atributos do elemento XML possui a variável *time*. Isto é porque esta variável é apenas utilizada nas novas transformações, ou seja, é fácil de identificar de que transformação se trata. As transformações antigas permanecem-se inalteradas.

3.2 Transformações com Tempo

3.2.1 Translação

De forma a fazer a translação, dois requerimentos tinham de ser cumpridos: a curva de Catmull-Rom tinha de ser desenhada e o modelo tinha de ser alinhado com a curva, se tal fosse preciso.

Para tratar da primeira tarefa, foram utilizadas algumas funções aprendidas durante as aulas práticas, que formam uma sequência de instruções que criam a curva. Começando com a função **renderCatmullCurve**, esta função recebe os nossos principais pontos para desenhar a curva, os pontos de controlo. Esta função leva a textbfGlobalCatmullRomPoint que serve para determinar a secção da curva onde iremos desenhar os pontos de dita secção com a função **getCatmullPoint** a obter os pontos individualmente de forma a depois podermos executar uma instrução **GL_LINE_LOOP**, dentro da função **renderCatmullCurve** que irá desenhar a curva com os pontos.

Com a curva desenhada, obtemos um novo ponto usando a função **GlobalCatmullRomPoint** e metemos o modelo em cima da curva. A sequência de instruções a seguir encontra-se localizada dentro de uma condicional, pois refere ao facto de que, segundo a variável **align**, o modelo pode ou não estar alinhado com a curva. Nesta condicional, traduzimos para código as fórmulas matemáticas ensinadas nas aulas de forma a normalizar vetores e a construir a matriz de rotação necessária para manter o modelo alinhado.

Após esta condicional, começamos o desenho em si mesmo, onde utilizamos as variáveis **time**,**timeNow**. Segmentando a translação usando o tempo, determinamos quantas translações mais pequenas precisam de serem efetuadas de forma a que após o tempo especificado na variável **time**, o modelo está a realizar a próxima volta à curva. Como óbvio, quando uma destas pequenas translações ocorre, alteramos a variável **timeNow** de forma a que a próxima translação possa ocorrer.

3.2.2 Rotação

A rotação é um pouco mais simple de efetuar e explicar que a translação. Mais uma vez recorrendo às variáveis **time** e **timeNow**, iremos utilizar a mesma técnica de segmentação que utilizamos na translação, mas com o tempo a servir como o divisor de 360, desta forma obtendo um número de segmentos a percorrer. Juntamente com as variáveis **var_x**, **var_y** e **var_z** para nos indicarem a orientação da rotação, cada incremento irá simular a rotação em pequenos pedaços, até que quando o tempo atingir o valor em **time**, o modelo começa outra rotação de 360 ângulos.

3.3 Implementação de VBOs

Para implementar os VBOs recorremos à struct apresentada em baixo:

```
struct vbo_object
{
    GLuint buffer;
    GLuint verticeCount;
};
```

Esta struct irá ser utilizada quando utilizada num formato de mapa, onde a chave será o nome do ficheiro que representa, criando assim uma forma de rápido acesso aos buffers que contém a informação crítica para desenho. No entanto, esta informação só se encontrará disponível nos buffers, quando for invocada no começo do programa, logo após o parsing do ficheiro XML, a função **prepareData**, a função que irá guardar o formato da nossa informação, juntamente com a mesma, criando VBOs que depois serão chamados para este efeito.

Durante o processo de parsing, os ficheiros no sistema que contém a informação são guardados numa variável chamada **modelos_em_memoria**, assim chamada porque irá guardar os nossos modelos na memória central para depois transferir para memória gráfica, sendo a formatação dos dados depois tratados por várias funções que transformam as variáveis nos tipos necessários para efetuar os comandos de criação de VBOs.

Com os VBOs criados e armazenados, iremos no processo de desenhar um dos modelos, simplesmente chamar a função **fromFiletoShape**, uma função modificada de fases anteriores que tem agora duas funções: encontrar o especifico VBO com o nome do modelo que lhe é fornecido e desenhar a nossa figura, assegurando que em desenhos mais complexos, o programa se encontrará com uma *performance* superior.

4 Conclusão

Dada por concluída a terceira fase do projeto, consideramos importante realizar uma análise crítica, e ainda, realçar os pontos positivos e negativos.

Durante o desenvolvimento do projeto surgiram algumas dificuldades, o que tornou impossível para o grupo, de modo a cumprir os prazos, implementar a totalidade das funcionalidades. As funcionalidades do Generator estão a funcionar como é esperado mas a aplicação do Engine, apesar de estar completamente implementada, tem um erro relacionado com a inclusão das bibliotecas do Glut. Isto resultou na impossibilitação de testar os modelos XML. Assim sendo, o grupo compromete-se a ter este problema resolvido aquando da entrega da fase final. No entanto, estamos confiantes que este pequeno precalço não invalida as soluções encontradas pelo grupo.

Realçamos também que se encontra na entrega desta fase o XML correspondente ao sistema solar dinâmico.

Assim, na próxima fase começaremos por explorar novas formas de corrigir esse erro e efetuar a testagem das restantes funcionalidades que não puderam ser testadas devido ao mesmo. Por outro lado, apesar das dificuldades encontradas no Engine o grupo conseguiu implementar corretamente a totalidade das funcionalidades no Generator o que consideramos ser um ponto positivo do trabalho.