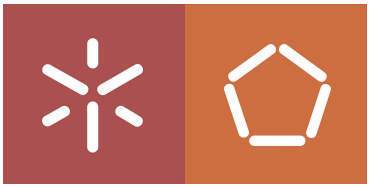

Universidade do Minho
Departamento de Informática

Computação Gráfica

Ano Letivo 2023/2024

Trabalho Prático
Fase 3 - Curvas, Superfícies Cúbicas e VBOs

Grupo 13

Ema Martins - A97678

Gonçalo Costa - A100824

Henrique Malheiro - A97455

Marta Rodrigues - A100743

Índice

Introdução

A terceira fase deste projeto, realizado no âmbito unidade curricular de Computação Gráfica, resumiu-se na adaptação das tarefas implementadas nas fases anteriores, principalmente no Engine e no Generator.

De tal modo, para o Engine fizemos alterações nas estruturas de dados para suportar novos tipos de translações e rotações. Estas mudanças às transformações geométricas permitem animar os modelos e cenas, dependente dos novos parâmetros que podem incluir, como o tempo e curvas de Catmull-Rom. Já para o Generator acrescentou-se a capacidade de gerar novos modelos através de *patches* de Bezier. Adicionalmente, criamos uma nova cena de demonstração dinâmica do Sistema Solar que preparamos na fase anterior, em prol de apresentar os acréscimos feitos.

Todas as metodologias e abordagens que permitiram a implementação desta etapa serão detalhas ao longo deste relatório.

Generator

Leitura dos pontos de controlo

O Generator deverá ser capaz de criar novos modelos baseados em *patches* de Bezier. Para tal, recebe como argumentos o nome do ficheiro *.patch* com os pontos de controlo de Bezier e o nível de *tessellation*. Deste modo, começamos por fazer a leitura do ficheiro fornecida para organizar os pontos dados por *patch*.

O ficheiro com os pontos de controlo está organizado da seguinte forma:

1. Número de *patches*;
2. Índices dos pontos para cada *patch*;
3. Número de pontos de controlo;
4. Pontos de controlo.

Assim, obtemos inicialmente o número de *patches* existentes no ficheiro, de seguida criamos uma variável *vector<vector<int>>* encarregada de armazenar os indíces para os 16 pontos de todos os *patches*. Proseguimos então para a obtenção do número de pontos de controlo e com esses, e os indíces adquiridos anteriormente, construímos os *patches* organizando cada ponto consoante os indíces que cada *vector<int>* guardou e obtendo o respetivo ponto para o adicionar ao *patch*.

O resultado desta função é, assim, um valor de tipo *vector<vector<vector<float>>>* que se traduz num conjunto de *patches* e cada *patch* é um conjunto de pontos, sendo que um ponto é representado pelo o último *vector<float>* (coordenadas *x*, *y* e *z*).

Com os pontos de controlo organizados por *patch*, pudemos implementar o calculo dos pontos para as superfícies cúbicas de Bezier.

Criação das superfícies cúbicas de Bezier

Para criar as superfícies de Bezier, precisamos de determinar os pontos que as compõem, partindo dos pontos de controlo fornecidos, usando para tal a seguinte equação:

$$p(u,v) = UMPM^TV$$

Sendo $U = [u^3 \quad u^2 \quad u \quad 1]$, $M = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 1 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$, $P = \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix}$ e $V = \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix}$

Note-se que, neste caso, a matriz M^T é igual a M, motivo pelo qual no código trabalhamos apenas com uma matriz. Como mencionado anteriormente, por cada patch existente no ficheiro *.patch* temos 16 pontos de controlo que constituem os pontos presentes na matriz P.

Para calcular os pontos que constituem a superfície de Bezier, por cada patch calcularam-se N*N pontos, sendo N o *tesselation*. O valor *tesselation* é fornecido quando se cria o ficheiro .3d. Assim, iremos variar o valor de *u* e de *v* com as diferentes combinações possíveis deles, utilizando-se dois ciclos para o fazer.

Inicialmente realizamos o cálculo literalmente como fornecido na equação anterior, mas tivemos bastantes problemas uma vez que as contas eram muito complexas, nomeadamente na multiplicação da matriz P com o resultante da multiplicação das matrizes M^T e V, uma vez que a matriz P tem 3 dimensões. Para facilitar os cálculos, optou-se por se fazer 3 matrizes a partir da matriz P, cada uma contendo uma das componentes dos pontos de controlo (*x*, *y* e *z*). Isto fez com que reduzisse-mos uma dimensão à matriz, apenas tendo multiplicações de vetores e matrizes de duas dimensões ou entre vetores. Achamos assim pertinente criar duas funções que fizessem cada um dos cálculos mencionados. Com esta abordagem, fazemos o cálculo separado de cada uma das componentes do ponto resultante.

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

O cálculo apresentado foi então o que realizamos para fazer a componente x do ponto resultante. Um cálculo semelhante foi realizado para obter as outras duas componentes do ponto.

Os pontos foram calculados tendo em conta a ordem com que os teríamos de inserir no vetor para posteriormente formar os triângulos base da figura.

O exemplo fornecido para teste pretendia que desenhássemos um teapot, tendo-se obtido o seguinte resultado:

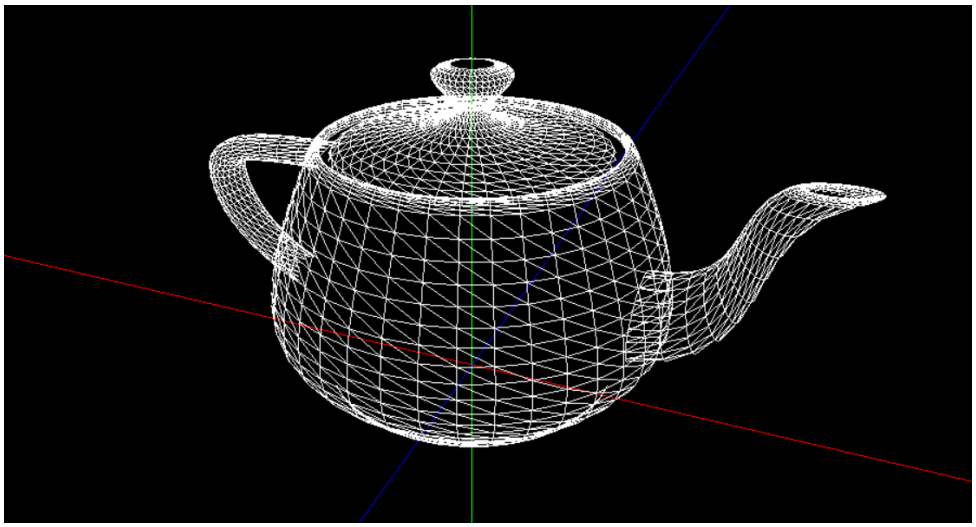

Figure 1: Teapot gerado

Engine

Foi também necessário alterar os elementos de translação e rotação. O objetivo destas transformações é poder realizar animações aos modelos gerados com base em curvas e tempo, respetivamente. Adicionalmente, a fim de alcançarmos estes objetivos foi preciso alterar as estruturas de dados que possuiamos anteriormente.

Alterações às estruturas de dados

Devido à alteração dos modelos de XML fornecidos, que agora permitem ter translações e rotações a demorar certo tempo a serem realizados, houve a necessidade de adicionar "gets" desses atributos na árvore de parse, bem como a sua adição à estrutura Transformation (para poder guardar os valores de time e align adicionados).

Quanto aos pontos em que o modelo vai percorrer durante essa translação, decidimos criar uma nova estrutura que guarde os valores das suas coordenadas, adicionando cada um dessa estruturas à lista de pontos da estrutura Transformation a que pertence.

A *struct* Transformation passou, assim, a ter a seguinte estrutura:

```
struct Transformation {
    Type type;
    float x, y, z, angle = 0, time;
    bool align;
    std::list<Point> points;
};
```

Figure 2: Struct para transformações

Translações dirigidas por curvas de Catmull-Rom

A fim de executar animações em translações, estas devem receber um conjunto de pontos de controlo para definir a curva cúbica de Catmull-Rom e o tempo em segundos que o modelo deve demorar para percorrer toda a curva. Pode também ser indicado se o objeto deverá estar alinhado à curva através de um valor *align*. Assim, para calcular os pontos da curva foram criadas duas funções a *getGlobalCatmullRomPoint* e *getCatmullRomPoint*.

Para obter a proporção do instante de tempo atual, com o tempo da curva (tempo fornecido na configuração XML), recorre-se à seguinte fórmula, onde *time* é o tempo fornecido:

$$gt = (glutGet(GLUT\_ELAPSED\_TIME) / 1000.0) / time$$

A função *getGlobalCatmullRomPoint* serve para calcular, a um dado instante de tempo, a posição de um certo objeto e devolve os pontos da posição e da derivada da posição. Esta recolhe os quatro pontos de controlo da curva para o valor de tempo no momento e fornece-os à função *getCatmullRomPoint*.

A função *getCatmullRomPoint* está encarregue de calcular então a posição do ponto na curva de Catmull-Rom. Com os quatros pontos de controlo fornecidos, *P*, podemos determinar a sua posição através da formula seguinte:

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

A derivada é obtida através da equação que se segue:

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

Caso o campo *align* seja aplicável a uma translação, precisamos definir a orientação dos eixos para alinhar à curva. Esta orientação é calculada com as fórmulas:

$$X_i = p'(t)$$

$$Y_i = Z_i \times X_i$$

$$Z_i = X_i \times X_{i-1}$$

Assumi-mos a inicialização do eixo Y (Y_0) como o vetor de coordenadas (0,1,0). Nestas fórmulas $p'(t)$ representa a derivada da curva no instante t, '$\times$' o produto vetorial e Y_{i-1} a orientação do eixo Y no instante anterior ao atual. Com os cálculos podemos então determinar uma matriz de rotação com a seguinte estrutura:

$$M = \begin{bmatrix} X_x & Y_x & Z_x & 0 \\ X_y & Y_y & Z_y & 0 \\ X_z & Y_z & Z_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Para visualizar a linha das trajétórias das curvas, adicionamos também a função *renderCatmullRomCurve* que recebe os pontos de controlo da translação e calcula as posições de todos os pontos da curva de 0.01 em 0.01 segundos (dentro de um tempo de 0 a 1 e um nível de *tesselation* de 100) através da *getGlobalCatmullRomPoint*.

Caso o tempo não seja indicado nas configurações XML, esse é considerado como 0 e a translação vai ser estática, isto é, sem animação.

Para complementar a justificação, toda a implementação das translações encontra-se na seguinte figura:

```
float pos[3], deriv[3];
float gt = ( glutGet(GLUT_ELAPSED_TIME) / 1000.0) / transformation.time;

getGlobalCatmullRomPoint(gt,transformation.points,pos,deriv);
renderCatmullRomCurve(transformation.points);

glTranslatef(pos[0], pos[1], pos[2]);
if (transformation.align) {
    normalize(deriv);

    float z[3];
    cross(deriv, prev_y, z);
    normalize(z);

    float y[3];
    cross(z, deriv, y);
    normalize(y);

    prev_y[0] = y[0];
    prev_y[1] = y[1];
    prev_y[2] = y[2];

    float m[16];
    buildRotMatrix(deriv, y, z, m);

    glMultMatrixf(m);
}
```

Figure 3: Implementação das translações com time

Rotações dependentes do tempo

Para obtermos uma rotação animada é necessário fornecer o tempo que o objeto demora a realizar uma rotação de 360 graus sobre o eixo pretendido. A partir desse valor, podemos deduzir o ângulo de rotação a qualquer instante da animação.

A todos os momentos da execução, calculamos o ângulo de rotação tendo em conta o tempo decorrido desde do início de execução do programa, obtido através do função *glutGet(GLUT_ELAPSED_TIME)* e do tempo de rotação definido na configuração XML. Assim, determinamos a seguinte fórmula, sendo o t o tempo decorrido e t_t o tempo definido para a rotação.

$$\alpha = \frac{360° * (t/1000)}{t_t}$$

Visto que os ângulos não devem, ou podem, ultrapassar os 360°, sempre que acontece isso retira-se 360 ao resultado.

Sistema Solar

A cena de demonstração requirida para esta fase é um sistema solar dinâmico que inclui todos os planetas, o sol, os satélites naturais da Terra e Marte e um cometa. Este cometa tem forma de um teapot, construido com os *patches* de Bezier fornecidos para o mesmo, e possui a sua trajetória definida com uma curva de Catmull-Rom.

O movimento de translação dos planetas e das luas é feito através de curvas dependentes do tempo. O cometa Teapot possui uma orbita elíptica à volta do Sol executadas com translações dirigidas por curvas de Catmull-Rom.

A seguinte figura representa o Sistema Solar com o cometa:

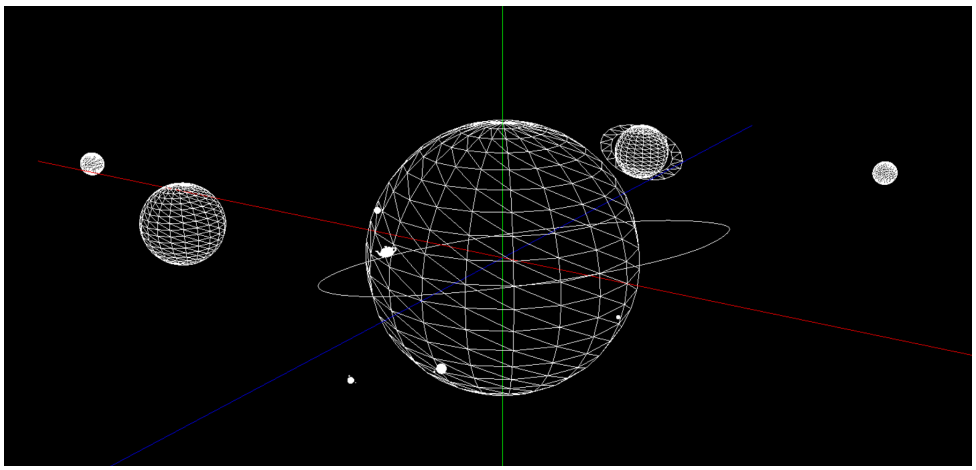

Figure 4: Sistema Solar com Teapot

Conclusão

Dado por concluída a terceira fase do projeto, consideramos que conseguimos aplicar os conhecimentos adquiridos relativos aos conceitos objetivo, nomeadamente curvas de Catmull-Rom e superfícies cúbicas de Bezier.

Apesar das dificuldades enfrentadas, principalmente com as superfícies de Bezier, sentimos que fomos capazes de as superar e atingir tudo o que foi proposto para esta fase. É importante lembrar que apesar de ser apenas explícito para os modelos passarem a ser desenhados através de VBOs nesta fase, já haváamos feito na primeira fase.

Finalmente, consideramos que reunimos os elementos necessários, com o desenvolvimento desta fase, para avançar para a última fase do projeto, na qual pretendemos, pelo menos, adicionar iluminação, texturas, um novo modo de câmera e novas cenas de demonstração.