Universidade do Minho

Escola de Engenharia

Computação Gráfica

Licenciatura em Engenharia Informática

Fase 3 - Curves, Cubic Surfaces and VBO's

Grupo 12

Ana Pires - [A96060]
Mariana Marques - [A93198]

Maio, 2023

Conteúdo

1. Introdução

1.1 Contextualização

O presente relatório tem como objetivo documentar e formalizar o procedimento efetuado para a implementação da terceira fase do Trabalho Prático de Computação Gráfica, constituinte do 2º semestre do 3º ano de Engenharia Informática.

1.2 Objetivos 3.ª Fase

Como está explicito no enunciado do presente trabalho prático, esta fase exige um conjunto de objetivos, tais como:

1. Atualizar o *generator* de forma a ser possível criar modelos baseados em *patches* de *Bezier* - Superfícies de *Bezier*;
2. Atualizar o *engine* para gerar os modelos através do uso de *VBOs*;
3. Extender as operações de rotação e translação através da *tag* ***point*** e os atributos ***time*** e ***align***, assim, com o auxílio do *engine* e com o algoritmo das **Curvas de *Catmull-Rom*** é possível obter animações relativas aos modelos.

2. Generator

Como já foi referido anteriormente, foi necessário atualizar o *generator* de forma a ser possível criar superfícies de *Bezier* baseados em *patches*.

2.1 Primitivas Gráficas

Para ser possível completar o objetivo referente ás superfícies de *Bezier*, foi necessário acrescentar a primitiva ***Bezier*** ás restantes existentes.

Assim, é possível listar as primitivas criadas até ao momento:

- **Plano**
- **Caixa**
- **Esfera**
- **Cone**
- **Torus**
- **Bazier**

2.1.1 *Bezier*

Como o próprio nome indica, esta primitiva permite a implementação de **Superfícies de *Bezier*** com recurso a *patches* e requer novos parâmetros, tais como:

- ***string file***: Nome do ficheiro *.patch*;
- ***int tesselation***: Representa a precisão do modelo a ser gerado, através dos incrementos bidirecionais de valor $\frac{1}{tesselation}$;
- ***map<int, vector<Point>> patches***: Mapa responsável por guardar as *patches* definidas no ficheiro, onde cada uma contém a sua identificação e o vetor de pontos de controlo respetivo.

De forma a ser possível a correta implementação desta primitiva, foram desenvolvidas as respetivas funções:

void parsePatch();

Responsável pelo *parsing* do ficheiro .patch correspondente e povoamento dos atributos da primitiva.

Point getBezierPoint(int p, float u, float v);

Responsável por calcular o ponto relativo aos parâmetros **u** e **v** na superfície cúbica de *Bezier* dada por **p(u,v)**.

$$p(u,v) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_{00} & P_{10} & P_{20} & P_{30} \\ P_{01} & P_{11} & P_{21} & P_{31} \\ P_{02} & P_{12} & P_{22} & P_{32} \\ P_{03} & P_{13} & P_{23} & P_{33} \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix}$$

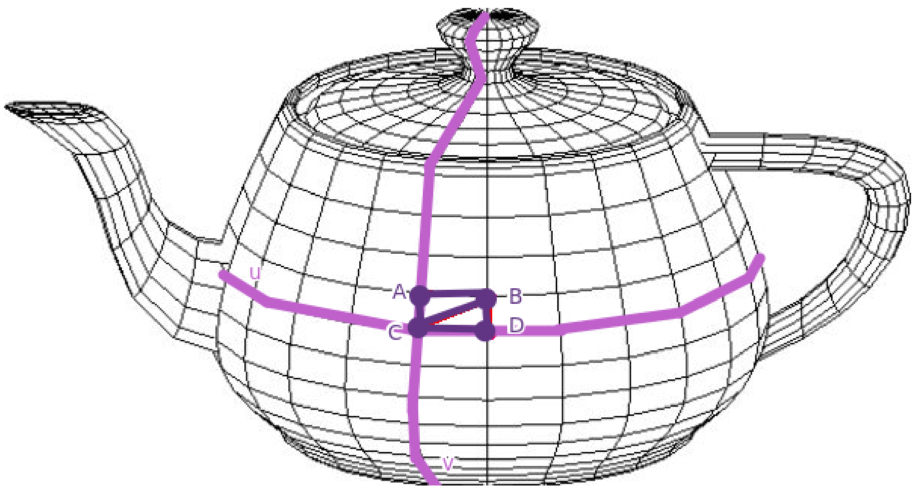
Figura 2.1: **Forma Matricial** da Superfície Cúbica de Bezier

Tendo por base a seguinte fórmula, define-se então a matriz de coeficientes referentes à superfície cúbica de **Bezier**, sendo posteriormente multiplicada pelos parâmetros u, v e, por fim, pela matriz referente aos pontos de controle, caracterizada pela multiplicidade 4 x 4, uma vez que cada *patch* é formado por 16 pontos de controle.

Esse processo é realizado para cada uma das coordenadas dos pontos de controle (**x**, **y** e **z**).

vector<Point>point_generator()

Responsável por gerar os vértices da primitiva, através da iteração pelo número de *patches* e pela *tesselation* bidirecional, **u** e **v** - horizontal e vertical, respetivamente.



Tendo em conta a imagem, determina-se que o triângulo superior é formado pelos pontos ABC e o triângulo inferior é formado pelos pontos BCD.

- **Ponto A**: getBezierPoint(patch, u + $\frac{1}{tesselation}$, v);
- **Ponto B**: getBezierPoint(patch, u + $\frac{1}{tesselation}$, v + $\frac{1}{tesselation}$);
- **Ponto C**: getBezierPoint(patch, u, v);
- **Ponto D**: getBezierPoint(patch, u, v + $\frac{1}{tesselation}$);

3. Engine

Como já foi referido anteriormente, foi necessário atualizar o *engine* para gerar os modelos através do uso de *VBOs* e extender as operações de rotação e translação através da *tag* ***point*** e os atributos ***time*** e ***align***, de forma a ser possível implementar as ***Curvas de Catmull-Rom***.

3.1 VBOs

De modo a melhorar a eficiência da renderização, realizou-se a implementação de **VBO's (*Vertex Buffer Object*)** ao gerar os modelos.

map<string, pair<unsigned int,unsigned int>> modelsVBOs

Para catalogar as *VBO's*, é criado um mapa, dado por ***modelsVBOs***, onde o elemento chave é o nome do ficheiro relativo ao modelo a ser gerado, e o elemento valor é um par que referencia, como primeiro elemento, o identificador do *VBO* e, como segundo elemento, o número de pontos do modelo.

No que toca à criação da *VBO*, foram utilizadas funções do *OpenGL*, tais como:

- **glGenBuffer**: Responsável por gerar o identificador do *buffer*;
- **glBindBuffer**: Responsável por vincular o *buffer* ao modelo a ser gerado;
- **glBufferData**: Responsável por copiar os dados para o *buffer*.

É de salientar que uma *VBO* só é criada caso o modelo em questão ainda não tenha sido lido, o que confere eficiência ao programa.

No que toca à renderização do modelo, foram utilizadas funções do *Glew* e *OpenGL*, tais como:

- **_glewBindBuffer**: Responsável por vincular o *buffer* ao alvo;
- **glVertexPointer**: Responsável por interpretar os dados dos vértices;
- **glDrawArrays**: Responsável por renderizar os modelos.

3.1.1 Estruturas de Dados

De forma a ser possível implementar esta funcionalidade, foram feitas alterações em relação às estruturas de dados, tais como:

1. **Model**

   Em relação à classe Model, foi adicionado o atributo *pairVBOs*, que se trata do elemento valor do mapa.

   Listing 3.1: Class Model

   ```
   class Model {
       private:
           string file;
           vector<Point>* points;
           pair<unsigned int, unsigned int> pairVBOs; <<<<<
   }
   ```

2. **Tree**

   Em relação à classe Tree, foi adiciona o atributo *modelsVBOs*, que se trata do mapa que guarda todas as *VBOs*.

   Listing 3.2: Class Tree

   ```
   class Tree {
       private:
           Window window;
           Camera camera;
           Group groups;
           map<string, pair<unsigned int,unsigned int>> modelsVBOs;
   }
   ```

3.2 Translação

No que toca à translação, foi necessário criar novos atributos para ser possível implementar o algoritmo correspondente ás curvas de *Catmull-Rom*, tais como:

- **float time**: Segundos que o modelo demora a percorrer a curva;
- **bool align**: Informa se o modelo tem de estar alinhado com a direção da curva;
- **vector <Point >points**: Responsável por guardar os pontos da curva;

3.2.1 Curvas de *Catmull-Rom*

Considerando um conjunto de 4 pontos, através das curvas de *Catmull-Rom*, obtém-se uma curva que expressa o segmento intermédio:

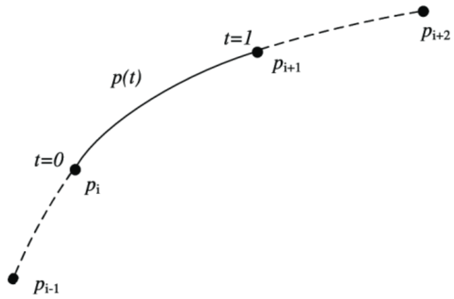


Figura 3.1: Segmento Intermédio

Desta forma, sabendo que p(t) = $at^3 + bt^2 + ct + d$, representa a expressão relativa às curvas de *Catmull-Rom*, é possível concluir:

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

Figura 3.2: p(t) em forma matricial

E, por fim, obtém-se as seguintes equações que permitem calcular as curvas:

$$C_0(t) = -0.5t^3 + t^2 - 0.5t$$
$$C_1(t) = 1.5t^3 - 2.5t^2 + t$$
$$C_2(t) = -1.5t^3 + 2t^2 + 0.5t$$
$$C_3(t) = 0.5t^3 - 0.5t^2$$

Considerando os conceitos teóricos acabados de definir, foram desenvolvidas as respetivas funções:

void getCatmullRomPoint(float t, Point p1..p4, Point *pos, Point *deriv);

Considerando que as curvas de *Catmull-Rom* são dadas por **p** e as respetivas derivadas por **p'**, tem-se:

1. p(t) = $at^3 + bt^2 + ct + d$;
2. p'(t) = $3at^2 + 2bt + c$.

Tendo por base a forma matricial da curva, é feita a multiplicação das coordenadas de cada ponto (p1..p4) pela matriz de coeficientes, e, por fim, é feita a substituição da matriz resultante em *p(t)* e *p'(t)*. Assim, é atingido o objetivo de calcular as coordenadas do ponto posição e derivada.

void getGlobalCatmullRomPoint(float gt, Point *pos, Point *deriv);

Permite um conjunto grande e pontos que, posteriormente, é divido em 4 sub-conjuntos de forma a respeitar as expressões definidas anteriormente.

void renderCatmullRoomCurve();

Responsável pela renderização da curva com auxílio da função *glBegin(GL_LINE_LOOP)* e *glEnd()*, onde, a cada iteração, é calculado o ponto global da curva.

void alignModel(Point *deriv);

Responsável pelo cálculo da direção do modelo, de forma, a igualar à direção da curva.

3.3 Rotação

No que toca à rotação, foi necessário criar novos atributos, tais como:

- **float angle**: Ângulo pelo qual a rotação irá ocorrer;
- **float time**: Segundos que o modelo demora a fazer uma volta de 360 graus;

Assim, antes de efetuar a ação, verificamos se o valor do atributo *time* é diferente de -1 (valor default, significa sem rotação periódica) e iremos obter o ângulo, multiplicando o valor do atributo *time* por 360, e após isto, aplicar a rotação.

4. Sistema Solar

Para assemelhar o Sistema Solar criado ao real, recorreu-se a todas as novas funcionalidades implementadas até ao momento, tornando tudo mais dinâmica e aprazível, uma vez que é possível observar cada planeta a girar sob as suas órbitas.

Para além de todos os corpos celestes anteriormente implementados, acrescentou-se ainda o cometa adicional, que era requisitado: cometa na forma de *teapot*. Neste também é possível observar a sua órbita e o seu movimento perante a mesma.

Uma vez que não é possível demonstrar a representação das órbitas e rotações dos corpos, apresenta-se apenas a representação estática do cenário:

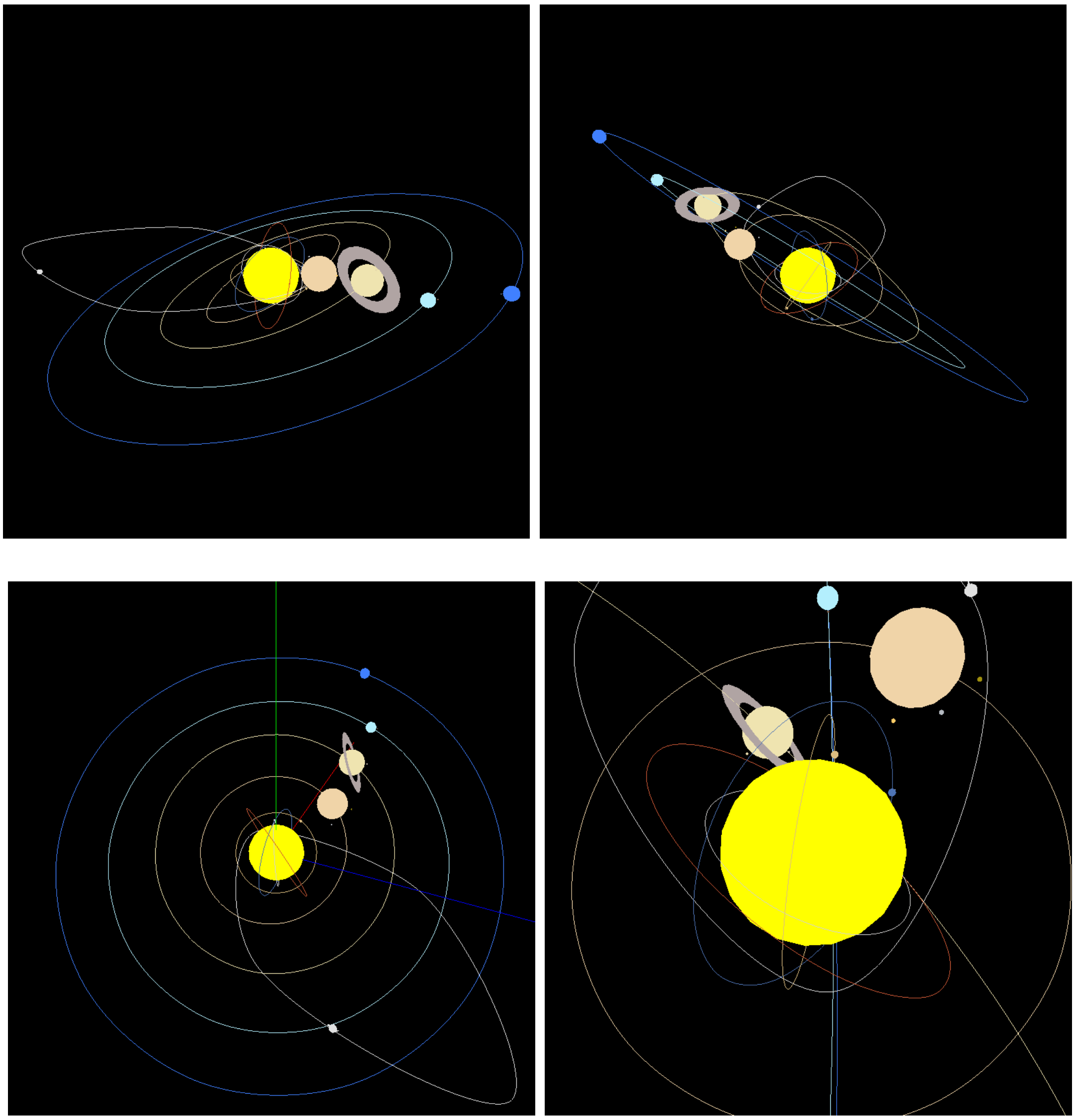


Figura 4.1: Sistema Solar visto de diferentes perspectivas

É de salientar que foi utilizado um programa em python de forma a obter as coordenadas dos pontos que pertencem ás órbitas dos planetas.

5. Testes

Ao executar os testes fornecidos no enunciado do trabalho prático, verificou-se que se realizaram todos com sucesso, como se confirma pela apresentação dos resultados obtidos.

5.1 Teste 1 - Curvas de *Catmull-Rom*

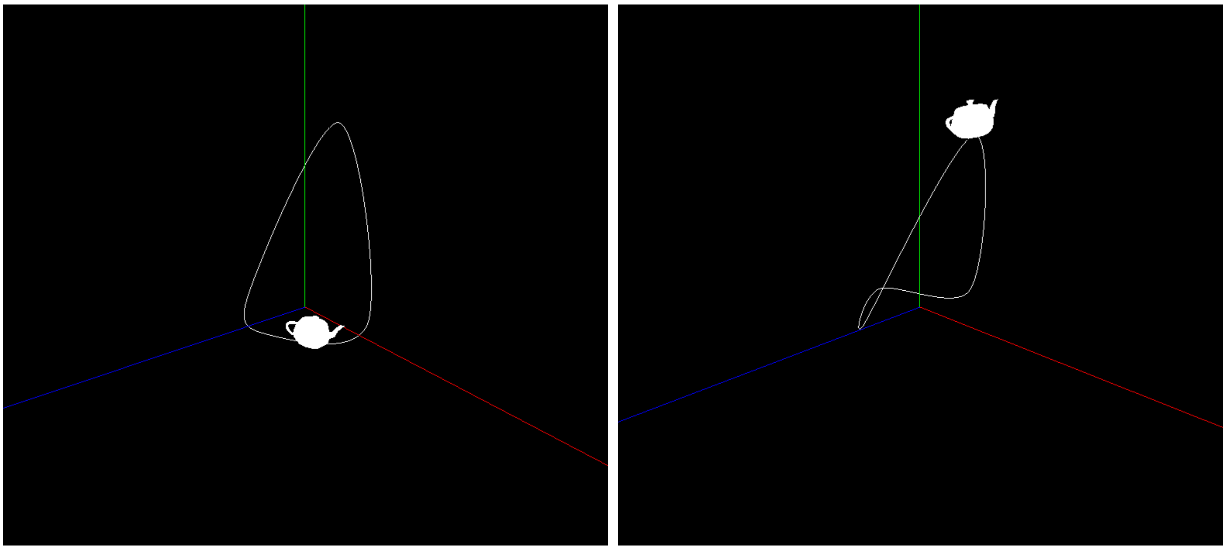


Figura 5.1: Teapot com rotação sob a curva de Catmull-Rom

5.2 Teste 2 - Superfícies de *Bezier*

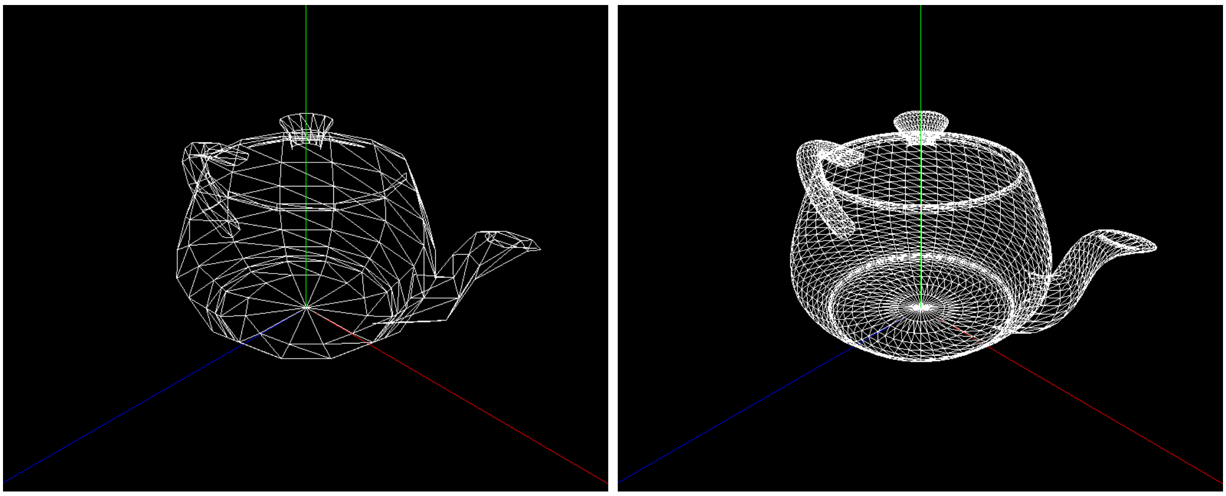


Figura 5.2: Teapot com níveis de tesselação de valor 3 e 10, respetivamente

6. Conclusão

Com esta fase do projeto realizada, verifica-se uma notória melhora na *performance* dos modelos, devida à implementação de *VBO's*. Para além disto, a extensão das operações de rotação e translação, também coloca a cena mais agradável visivelmente. Algo também bastante aprazível, é com certeza, notar na maior aproximação ao real do Sistema Solar criado, sendo isto apenas possível recorrendo ao auxílio das curvas de *Bezier* e *Catmull-Rom*.

Esta etapa apresentou-se, então, bastante satisfatória uma vez que foi possível colocar em prática os novos conhecimentos adquiridos em aula, e também, visto que foi possível cumprir todos os requisitos propostos.