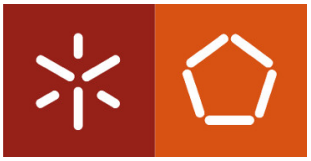
Universidade do Minho

Escola de Engenharia



Computação Gráfica

Licenciatura em Engenharia Informática

Curves, Cubic Surfaces and VBOs

Trabalho Prático - Fase 3 - Grupo 7

João Silva - [A89293]
Francisco Novo - [A89567]
João Vieira - [A93170]
Tiago Ribeiro - [A93203]

Maio, 2022

Conteúdo

Introdução

No âmbito da Unidade Curricular de Computação Gráfica, foi desenvolvida a terceira fase do trabalho prático proposto pelos docentes. O principal objetivo desta fase era o uso de superfícies de Bezier para a geração de uma primitiva a partir de um ficheiro, a implementação de transformações (translações e rotações) com o tempo, sendo que as translações são feitas com recurso a curvas de Catmull-Rom e alterar o motor de modo a que o desenho dos modelos faça uso dos VBO's (*Vertex Buffer Object*).

Para isto, foi necessário proceder à alteração do trabalho feito nas fases anteriores. O ficheiro XML pode ter novos formatos, tendo sido para o efeito criadas classes que permitiram armazenar o conteúdo do ficheiro, para depois serem desenhados os modelos e as transformações temporizadas. O desenho das primitivas foi alterado, de modo a que este seja feito pela placa gráfica. Relativamente ao ficheiro dos patches, foi implementada a leitura do mesmo para proceder ao cálculo dos vértices da primitiva.

A *demo scene* do sistema solar foi alterada com o intuito de serem representadas as órbitas dos planetas e um cometa com a trajetória definida a partir das curvas de Catmull-Rom.

Alterações na estrutura do projeto

Nesta fase do trabalho, o grupo teve de trabalhar com novas transformações que irão usar curvas, logo será necessário manipular matrizes e guardar informação destas mesmas transformações.

Package Matrices

Para este efeito foi criado o *package* Matrices que contêm funções disponibilizadas nas aulas práticas pelos docentes que nos permitem fazer operações com matrizes.

```
namespace matrices{
    void multMatrixVector(float*, float*, float*);
    void buildRotMatrix(float*, float*, float*, float*);
    void cross(float *, float *, float *);
    void normalize(float *);
}
```

Figura 1: Funções presentes no *package* Matrices

Struct das transformações com tempo

A *struct* utilizada para lidar com as transformações com tempo tem o seguinte formato:

```
enum class timedTransformation {
    rotate,
    translate
};

class timedTransform {
    int time;
    float x;
    float y;
    float z;
    bool align;
    timedTransformation trans;
    std::vector<point> points;
    std::vector<point> curvePoints;
```

Figura 2: Struct *timedTransformation*

Para indicar o tipo da estrutura é utilizada uma classe *enum*, tal como nas transformações da fase 2. O resto dos parâmetros servem para guardar informação que irá ser recebida nos ficheiros XML, exceto o vetor de pontos *CurvePoints* que irá guardar, depois de calculados, os pontos que o objeto irá ter de percorrer na translação.

Alterações no gerador

Nesta fase o gerador tem de ser capaz de gerar primitivas a partir de *patches* de Bezier. O gerador irá receber um ficheiro com a informação destes patches e tem de ser capaz gerar os pontos para criar o ficheiro 3d da primitiva.

Primitivas com patches de Bezier

O gerador vai receber o nome do ficheiro de *patches* e o valor da tesselação. Depois de recebidos estes parâmetros a função *readBezierFile* irá fazer o parsing do ficheiro de *patches*. Esta função irá criar um vetor com os pontos de controlo dos *patches* e outro vetor com os indíces dos pontos, sabendo que a cada 16 indíces irá corresponder a um *patch*. De seguida, estes vetores e o nível de tesselação são passados à função *createBezier*.

A função *createBezier* irá criar percorrer o vetor dos indíces para aceder ao vetor que contêm os pontos de controlo dos *patches*. Depois irão ser feitos dois ciclos, onde irão criadas as matrizes de cada coordenada e determinados os valores de u e v, que dependem do valor da tesselação.

$$x_{i+1} = x_i + \frac{1}{tesselacao}, \quad i \in [0, tesselacao] \wedge x \in \{u, v\}$$

Irão ser calculados os quatro pontos do quadrado atual da grelha, que é determinado com os valores de u e v.

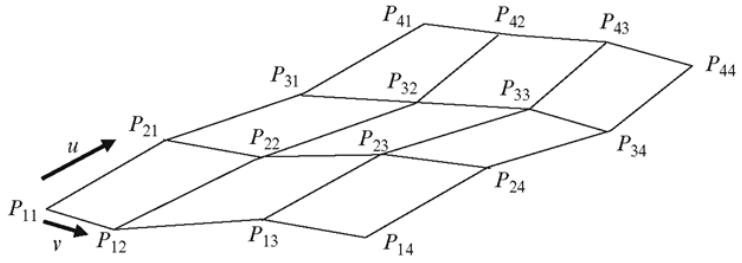


Figura 3: Exemplo da grelha de um patch

A função *calculateBezierPoint* irá determinar as coordenadas destes pontos. No início, a partir da matriz de Bezier e das matrizes dos pontos *patch* irá calcular uma matriz.

$$matrix = M.\begin{pmatrix} P_{00} & P_{10} & P_{20} & P_{30} \\ P_{01} & P_{11} & P_{21} & P_{31} \\ P_{02} & P_{12} & P_{22} & P_{32} \\ P_{03} & P_{13} & P_{23} & P_{33} \end{pmatrix}.M^T$$

Figura 4: Matriz calculada

A matriz M é a matriz de Bezier, sendo que ela é simétrica, por isso, a sua transposta é igual a ela mesma.

$$M = \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Figura 5: Matriz Bezier

Para obter as coordenadas de um ponto é usada a seguinte fórmula:

$$p(u,v) = (u^3 \quad u^2 \quad u \quad 1).matrix.\begin{pmatrix} v^3 \\ v^3 \\ v \\ 1 \end{pmatrix}, \quad u,v \in [0,1]$$

Figura 6: Fórmula para calcular as coordenadas de um ponto do *patch*

Depois de calculadas as coordenadas dos pontos de um quadrado da grelha do *patch*, irão ser criados os triângulos usando a regra da mão direita, sendo estes introduzidos no objeto *figure*. Este processo irá ser repetido para todos os quadrados do *patch*, sendo no final usada a função *createFile* para escrever no ficheiro 3d os dados contidos no objeto *figure*.

Alterações no motor

No motor foram implementadas as novas transformações que utilizam o tempo. Estas serão as translações com recurso a curvas de Catmull-Rom e as rotações em 360ª sobre um eixo.

Translações

A extensão das translações, fará uso das curvas de Catmull-Rom. Estas curvas são um tipo de curvas cúbicas, muito usadas na área da computação gráfica. São definidas por quatro pontos de controlo P_0, P_1, P_2, P_3, sendo a curva desenhada apenas entre os pontos P_1 e P_2.

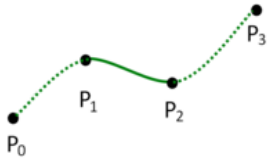


Figura 7: Exemplo de curva de Catmull-Rom

Primeiramente, é lida a duração da translação, se queremos que o nosso objeto fique alinhado com a curva ou não e os pontos do ficheiro XML que irão ser os pontos de controlo usados para o cálculo das curvas, sendo todos estes dados armazenados num objeto *timedTransformation*. Após isto é feito o cálculo dos pontos da curva a partir da função *calculateCurvePoints*, que dados os pontos de controlo, preenche o vetor *curvePoints* do objeto da transformação.

Se no ficheiro XML a *flag* que indica se o objeto deve estar alinhado com a curva for *true*, na função que irá desenhar os objetos irão ser realizadas normalizações, cálculos vetoriais e o cálculo de uma matriz de rotação, usando as funções do *package* Matrices, que irão efetuar o alinhamento da primitiva com a curva calculada.

Rotações

Para efetuar uma rotação com o tempo, são passados no ficheiro XML o tempo em que deve ser feita a rotação 360º e o eixo em que esta deve ser feita. Com isto, a partir da fórmula apresentada a seguir, é possível calcular o ângulo da rotação em relação ao tempo atual.

$$angulo = \frac{\frac{tempo\_atual}{1000} \times 360}{tempo\_rotacao}$$

Após isto, é feita uma rotação usando o *glRotatef* com o ângulo calculado, e os valores dos eixos referidos no ficheiro XML.

VBO's

Relativamente à implementação dos VBO's, optou-se por uma estratégia em que os VBO's são recolhidos e desenhados enquanto se efetua a leitura de um group no ficheiro XML.

Ao mesmo tempo que se efetua a leitura dos pontos no ficheiro *.3d*, estes são também armazenados num *buffer* simultaneamente, contando também o número de pontos armazenados (sendo que cada ponto corresponde a 3 coordenadas lidas).

O número de pontos lidos é depois registado e associado ao grupo a que pertence, mantendo assim a informação de quantos pontos estão contidos em cada grupo.

Feita a leitura e armazenamento destes dados, é possível desenhar os pontos contidos no VBO. Quando se termina a leitura de todos os models no ficheiro XML, é feita uma invocação à função *glDrawArrays*, sendo fornecido como ponto de início o ponto onde se parou anteriormente (por definição, se estivermos ainda a desenhar o conteúdo do primeiro *group*, começa-se a desenhar a partir do índice 0), e como ponto final do desenho é fornecido o número total de pontos lidos nesse mesmo *group*. A variável *pontosLidos* é depois atualizada com o valor do ponto de paragem que foi fornecido na função anterior, de forma a que no próximo group se comece a desenhar a partir do ponto onde se parou.

Ficheiros de teste

Foram corridos os ficheiros de teste fornecidos pelos docentes e os resultados foram os seguintes:

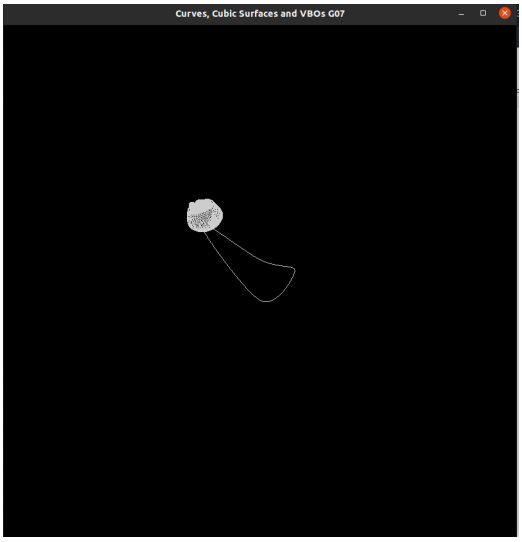


Figura 8: Ficheiro de teste *test_3_1.xml*

Neste teste a trajetória do *teapot* estava em rotação, sendo que este estava alinhado com a curva.

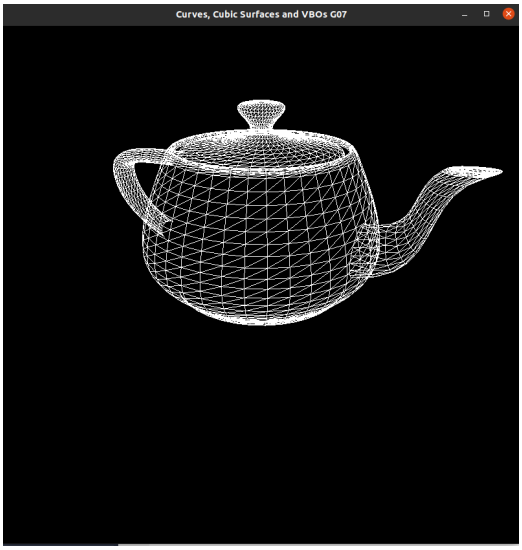


Figura 9: Ficheiro de teste *test_3_2.xml*

Neste teste foi desenhado um *teapot* com recurso a um ficheiro de *patches*.

Sistema Solar

No sistema solar é possível observar que os diferentes planetas já contém as suas respetivas órbitas.

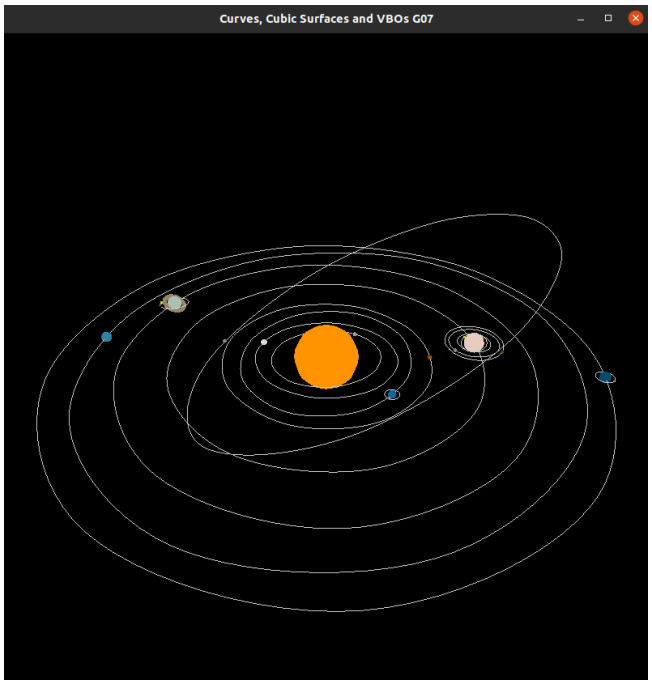


Figura 10: Sistema Solar

Para obter os pontos fornecidos no ficheiro XML para representar a órbita, foi utilizado o seguinte programa em Python:

```python
import math
import sys

x = float(sys.argv[1])
y = float(sys.argv[2])
z = float(sys.argv[3])
r = float(math.pi/4)
for i in range(0,8):
    new_x = str(math.cos(r*i)*x)
    new_y = str(math.sin(r*i)*y)
    new_z = str(-(math.sin(r*i)*z))
    print("<point x=\"" + new_x + "\" y=\"" + new_y + "\" z=\"" + new_z + "\"/>")
```

Figura 11: Programa em Python

As direções das rotações dos planetas são baseados na realidade dos mesmos, sendo que Vénus e Neptuno têm rotação no sentido oposto dos outros planetas. A velocidade de translação dos planetas é também relacionada com a velocidade real de translação, sendo Mercúrio o planeta que possui maior velocidade de translação, e Neptuno o mais lento.

O cometa utiliza a primitiva do *teapot* tal como especificado no enunciado, sendo a órbita deste uma elipse muito achatada que passa muito perto dos planetas interiores e perto do Sol. A órbita do cometa também é demorada, dando assim um ar mais realista, pois o formato e a duração da órbita seguem a mesma dos cometas na realidade.

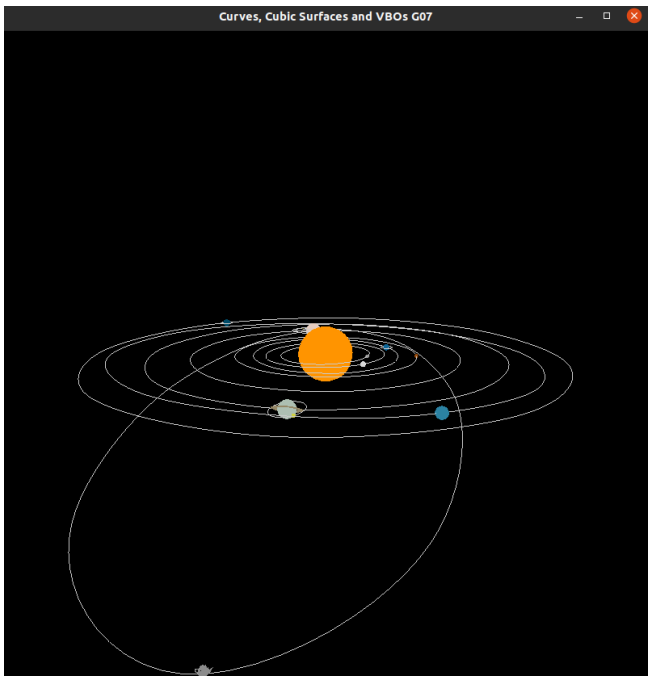


Figura 12: Órbita do cometa

Extras

Foram também adicionados alguns extras, nomeadamente a capacidade de ocultar o desenho das órbitas, ficando apenas desenhados os planetas, e a capacidade de mudar a renderização dos planetas para wireframe, para que seja mais fácil distinguir as rotações dos mesmos.

É possível ocultar/desenhar as órbitas dos planetas pressionando a tecla "o"no teclado.

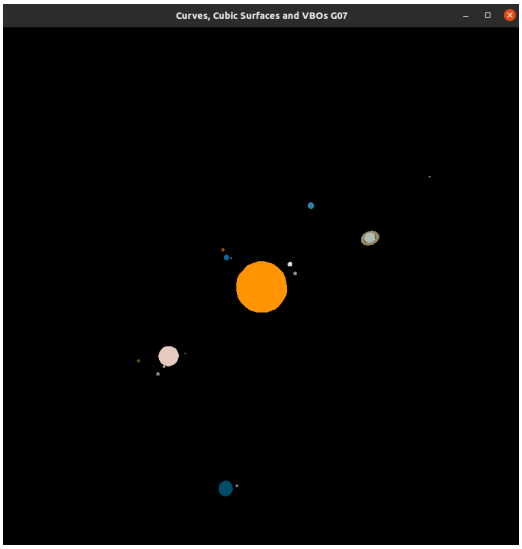


Figura 13: Sistema solar sem as órbitas

É possível renderizar os planetas em *wireframe* pressionando a tecla "w"no teclado.

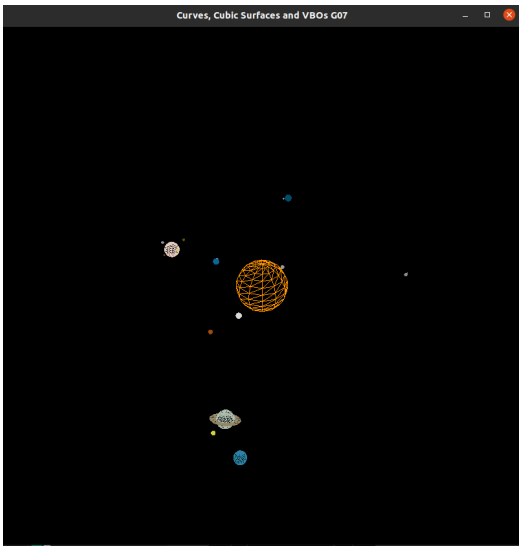


Figura 14: Sistema Solar desenhado em wireframe

Conclusão

O trabalho desenvolvido permitiu aprofundar o conhecimento sobre curvas cúbicas, mais concretamente as curvas de Catmull-Rom e de Bezier.

Verificamos que a aplicação gerador é capaz de gerar ficheiros 3d a partir de ficheiros de patches. A aplicação motor faz uso dos VBO's e é capaz de desenhar transformações com tempo, ou seja, efetuar as translações definidas como curvas de Catmull-Rom e as rotações de 360^{o} em torno de um eixo.

Em suma, o trabalho desenvolvido é considerado positivo, apesar de algumas dificuldades inicias em torno dos cálculos necessários para as curvas cúbicas.