

UNIVERSIDADE DO MINHO

MESTRADO INTEGRADO DE ENGENHARIA
INFORMÁTICA

Curvas, Superfície Curvas, VBOs

Fase III

Autores:

Sara Pereira	A73700
Filipe Fortunato	A75008
Frederico Pinto	A73639

29 de Abril de 2018

Conteúdo

1	Introdução	2
2	Arquitetura do Código	2
2.1	Aplicações	2
2.1.1	Gerador	2
2.1.2	Motor	2
2.2	Classes	3
2.2.1	Translação	3
2.2.2	Rotação	4
2.2.3	Transforms	5
2.2.4	Camara	6
3	Gerador	7
3.1	<i>Bézier</i>	7
3.1.1	Processamento do ficheiro input	7
3.1.2	Estruturas de Dados	7
3.1.3	Processamento dos <i>patches</i>	8
4	Motor	9
4.1	VBOs	9
4.2	Curva Catmull-Rom	10
4.2.1	Rotação	10
4.2.2	Translação	10
4.2.3	Desenhar as curvas Catmull-Rom	11
5	Resultados Obtidos	12
5.1	Teapot	12
5.2	Sistema Solar	14
6	Conclusão	15

1 Introdução

Foi-nos proposto, no âmbito da UC Computação Gráfica, que no seguimento do trabalho já feito nas fases anteriores que alterássemos o trabalho já feito de maneira a acrescentar novas funcionalidades, mais concretamente **translações** e **rotações** sendo que a translação está definida por pontos de uma curva e por tempos para percorrer a curva e modificar a rotação para esta também ser associada ao tempo. Para conseguirmos adicionar estas novas funcionalidades, usamos o algoritmo de *Bézier*. Desta maneira, alterámos o Gerador para que este seja capaz de criar uma nova primitiva baseada nas curvas de Bézier. Para além disso vamos também recorrer ao VBOs para nos ajudar no desenho de todos os diferentes modelos. Consequentemente, no presente relatório explicamos quais e como foram as alterações feitas ao nosso trabalho.

2 Arquitetura do Código

Devido ao facto de este trabalho ser no seguimento dos anteriores, tendo sido feitas diversas alterações necessárias a cada uma das seguintes aplicações a serem cumpridos os novos requisitos.

2.1 Aplicações

No presente tópico é apresentado todas as alterações feitas às duas principais aplicações do trabalho. Foram necessárias algumas alterações ao ficheiro **XML** e consequentemente alterações no motor de processamento deste ficheiro. Foi também alterado o gerador de maneira a suportar o algoritmos de Bézier.

2.1.1 Gerador

Indo ao encontro do que foi feito nas fases anteriores, gerador destina-se a gerar vários pontos constituintes das várias primitivas gráficas conforme os parâmetros fornecidos. Nesta fase, para além das primitivas gráficas anteriores, foi introduzido um novo método de construção de modelos com base em curvas de Bézier, sendo que foi necessário adicionar novas funcionalidades ao gerador em relação às fases anteriores.

2.1.2 Motor

O objetivo desta aplicação continua a ser o mesmo: permitir a apresentação de uma janela e exibição dos modelos requisitados. Além disso, permite, também, a interação com os mesmos a partir de certos comandos. Tal como na versão anterior, existe um ficheiro **XML** que vai ser interpretado. No entanto, para esta fase do projeto, a arquitetura deste ficheiro evolui de maneira a cumprir os requisitos. Desta forma, foram feitas algumas alterações nos métodos de *parsing* e consequentemente alterações nas medidas de armazenamento e também

de renderização. Para além disso é também implementado o *Catmull-Rom Cubic Curves* e de *Vertex Buffer Objects*.

2.2 Classes

Para esta fase do projeto, foi apenas necessário fazer alterações nas classes criadas já nas fases anteriores. De seguida mostramos em quais classes fizemos alterações e explicando as mesmas.

2.2.1 Translação

Classe que guarda toda a informação referente a uma translação (x_eixo, y_eixo, z_eixo) a todos os pontos de controlo de uma curva e a um conjunto de pontos finais de translação, após ser calculada a curva de maneira a criar as mesmas com o método **CatmullRom**.

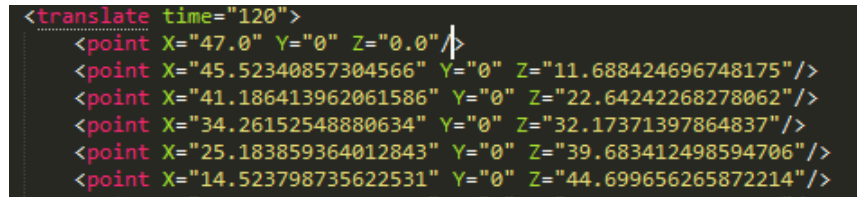
```
1
2 #include <vector>
3 #include <stdlib.h>
4 #include <math.h>
5 #include "Ponto.h"
6 #ifdef __APPLE__
7 #include <GLUT/glut.h>
8 #else
9 #include <GL/glut.h>
10 #endif
11
12
13 using namespace std;
14
15 #ifndef PROJECT.TRANSLACAO.H
16 #define PROJECT.TRANSLACAO.H
17
18 class Translacao{
19     float x_eixo;
20     float y_eixo;
21     float z_eixo;
22
23     float time;
24     int size;
25     float cima[3];
26     vector<Ponto> transl;
27     vector<Ponto> curvas;
28
29 public:
30     Translacao();
31     Translacao(float x, float y, float z, float ti, int s, vector<
32     Ponto> t);
33     float getX(){ return x_eixo; }
34     float getY(){ return y_eixo; }
35     float getZ(){ return z_eixo; }
36     float getTime(){ return time; }
37     int getSize(){ return size; }
38     float* getCima(){ return cima;}
39     vector<Ponto> getTransl(){ return transl; }
```

```

39 vector<Ponto> getCurvas(){ return curvas; }
40 void setX(float x){ x_eixo = x;}
41 void setY(float y){ y_eixo = y;}
42 void setZ(float z){ z_eixo = z;}
43 void setTime(float x){ time = x;}
44 void setSize(float y){ size = y;}
45 void setTransl(vector<Ponto> t){ transl = t;}
46 void setCurvas(vector<Ponto> c){ curvas = c;}
47 void getCatmullRomPoint(float t, int* pos, float* deriv, float*
    res, vector<Ponto> tr);
48 void getGlobalCatmullRomPoint(float t, float* deriv, float* res
    , vector<Ponto> tr);
49 void renderCatmullRomCurve( vector<Ponto> pontos, float r,
    float g, float b);
50 void normaliza(float* f);
51 void cruz(float* f, float* f2, float* res);
52 void rodaCurva(float* deriv, float* cima);
53 vector<Ponto> encurvar();
54 bool semTranslacao();
55 };
56 #endif //PROJECT.TRANSLACAO.H

```

Ao mesmo tempo, teve que ser alterado o ficheiro XML de maneira a ser adicionado o campo **time** numa **translação**.



```

<translate time="120">
  <point X="47.0" Y="0" Z="0.0"/>
  <point X="45.52340857304566" Y="0" Z="11.688424696748175"/>
  <point X="41.186413962061586" Y="0" Z="22.64242268278062"/>
  <point X="34.26152548880634" Y="0" Z="32.17371397864837"/>
  <point X="25.183859364012843" Y="0" Z="39.683412498594706"/>
  <point X="14.523798735622531" Y="0" Z="44.699656265872214"/>

```

Figura 1: Exemplo de alteração no XML adicionando o campo time

2.2.2 Rotação

Classe que guarda toda a informação referente a uma rotação. Sendo assim, é composta por quatro variáveis **angle**, **x_eixo**, **y_eixo** e **z_eixo**, que identificam em qual dos eixos vai ser efetuada a rotação. Nesta fase adicionamos também uma variável **time** que representa o número em segundos que demora a fazer uma rotação de 360° sobre o eixo especificado.

```

1
2 #ifndef PROJECT.ROTACAO.H
3 #define PROJECT.ROTACAO.H
4
5
6 class Rotacao {
7     float time, angle, x_eixo, y_eixo, z_eixo;
8
9 public:
10     Rotacao();
11     Rotacao(float time, float angle, float x, float y, float z);
12     float getTime(){ return time;}

```

```

13     float getAngle() { return angle; }
14     float getX() { return x_eixo; }
15     float getY() { return y_eixo; }
16     float getZ() { return z_eixo; }
17     void setTime(float a) { time = a; }
18     void setAngle(float a) { angle = a; }
19     void setX(float x) { x_eixo = x; }
20     void setY(float y) { y_eixo = y; }
21     void setZ(float z) { z_eixo = z; }
22     bool semRotacao();
23 };
24
25
26 #endif //PROJECT.ROTACAO.H

```

2.2.3 Transforms

Classe que guarda toda a informação relativa a um conjunto de transformações totais de um grupo, incluindo os subgrupos (filhos). Sendo assim, é constituída por um **tipo**, a transformação que irá ser aplicada **Transformacao t**, um vetor com os filhos **vector<Transforms> subgrupo** e os pontos para o desenho da figura **vector<Ponto> pontos**. Para além disto, nesta fase acrescentamos também um **buffer** para podermos implementar os **VBOs**.

```

1
2 #include <vector>
3 #include "Transformacao.h"
4 #include "Ponto.h"
5 #include <fstream>
6 #include <iostream>
7 #include <string>
8
9 #ifdef __APPLE__
10 #include <GLUT/glut.h>
11 #else
12 #include <GL/glew.h>
13 #endif
14
15
16
17 using namespace std;
18 #ifndef PROJECT_TRANSFORMS.H
19 #define PROJECT_TRANSFORMS.H
20
21
22 class Transforms{
23     string tipo;
24     Transformacao t;
25     vector<Ponto> pontos;
26     GLuint buffer[3];
27     int pos;
28     vector<Transforms> subgrupo;
29
30 public:
31     Transforms();

```

```

32     Transforms(string tipo, Transformacao t, vector<Transforms> sub
    , vector<Ponto> pontos);
33     string getTipo(){ return tipo; }
34     Transformacao getTransformacao(){ return t; }
35     vector<Transforms> getSubgrupo(){ return subgrupo; }
36     vector<Ponto> getPontos(){ return pontos; }
37     void setTipo(string t){ tipo = t; }
38     void setTrans(Transformacao trans){ t = trans;}
39     void setSubgrupo(vector<Transforms> sub){ subgrupo = sub; }
40     void setPontos(vector<Ponto> p){ pontos = p;}
41     void toVertex();
42     void setVBO();
43     void draw();
44     void push_child(Transforms t){subgrupo.push_back(t);}
45 };
46
47
48 #endif //PROJECT.TRANSFORMS_H

```

2.2.4 Camara

Classe que guarda toda a informação relativa à câmara do nosso trabalho. Esta é constituída por um **rotY** e **rotX** que indicam o ângulo da rotação da câmara, três variáveis **X Y Z** que indicam a posição da câmara, duas variáveis **speed** e **rotSpeed** indicando a velocidade da câmara e a velocidade da rotação da mesma, uma variável **frame**, uma variável **timebase** para o modelo ser apresentado em **FPS** e quatro valores booleanos **cima**, **baixo**, **esq**, **dir** de maneira a ser possível o uso de duas teclas ao mesmo tempo para movimentar a câmara, o que permite, assim, um movimento mais fluido por parte da mesma.

```

1
2
3 #ifndef PROJECT.CAMARA_H
4 #define PROJECT.CAMARA_H
5
6 #endif //PROJECT.CAMARA_H
7 #ifdef __APPLE__
8 #include <GLUT/glut.h>
9 #else
10 #include <GL/glut.h>
11 #endif
12
13 #include <stdio.h>
14 #include <math.h>
15
16 class Camara{
17     float rotY, rotX;
18     float lX, lY, lZ;
19     float llX, llY, llZ;
20     float speed, rotSpeed;
21     int frame = 0;
22     int timebase = 0;
23
24     bool cima = false, baixo = false, esq = false, dir = false;
25 public:

```

```

26     Camara();
27
28     float getRotY(){ return rotY;}
29     float getRotX(){ return rotX;}
30     float getLX(){ return lX;}
31     float getLY(){ return lY;}
32     float getLZ(){ return lZ;}
33     float getLLX(){ return llX;}
34     float getLLY(){ return llY;}
35     float getLLZ(){ return llZ;}
36     float getSpeed(){ return speed;}
37     float getRotSpeed(){ return rotSpeed;}
38     void setLX(float lx){ lX = lx;}
39     void setLY(float ly){ lY = ly;}
40     void setLZ(float lz){ lZ = lz;}
41     void setLLX(float llx){ llX = llx;}
42     void setLLY(float lly){ llY = lly;}
43     void setLLZ(float llz){ llZ = llz;}
44     void setSpeed(float s){ speed = s;}
45     void setRotSpeed(float rspeed){ rotSpeed = speed;}
46
47     void displayFPS();
48     void mouseMove(int x, int y);
49     void camaraMove();
50     void pressKeys(unsigned char key, int x, int y);
51     void releaseKeys(unsigned char key, int x, int y);
52 };

```

3 Gerador

3.1 *Bézier*

3.1.1 Processamento do ficheiro input

Antes de processar o ficheiro de input (*.patch*), tivemos que entender o formato do ficheiro para posteriormente gerar a figura. Podemos então concluir que o ficheiro (*.patch*) não representa uma grande complexidade:

- Na primeira linha surge o número de patches(**nPatches**);
- As próximas linhas que são no total **nPatches**, têm cada uma, 16 números inteiros que correspondem aos índices de cada um dos pontos de controlo que fazem parte desse patch;
- Posteriormente aparece um inteiro que representa o número de pontos de controlo (**cPoints**);
- Por fim surgem os pontos de controlo que, no total, são (**cPoints**);

3.1.2 Estruturas de Dados

Após termos analisado a estrutura do ficheiro de input, decidimos guardar toda a informação fornecida em arrays de arrays.

Guardamos os 16 números inteiros correspondentes a um determinado patch num array de arrays que armazena inteiros, $index[i][j]$, onde i representa o índice do patch em questão, varia entre $0 \leq i < nPatches$. Por outro lado, j varia entre $0 \leq j < 16$.

Para guardar os pontos de controlo, utilizamos a mesma técnica. Foi criado um array de arrays capaz de armazenar floats, $points[i][j]$, onde i representa o índice do ponto de controlo que varia entre $0 \leq i < cPoints$. Já j , representa as várias X, Y e Z dos pontos, ou seja, j varia entre $0 \leq j < 3$.

Armazenando nessas estruturas os dados fornecidos no ficheiro de input permitiu-nos avançar para a fase em que fazemos um processamento de todos os dados.

3.1.3 Processamento dos *patches*

Antes de passarmos à explicação do algoritmo que traduz o ficheiro de input num modelo geométrico, temos primeiro que entender como funcionam as curvas de *Bézier*.

Para a criação de uma curva de Bézier é necessário 4 pontos, definidos num espaço 3D, ou seja, constituídos por três coordenadas: **X**, **Y**, **Z**.

Esses pontos só por si não geram a curva, para processar a curva é preciso combinar estes pontos com alguns coeficientes.

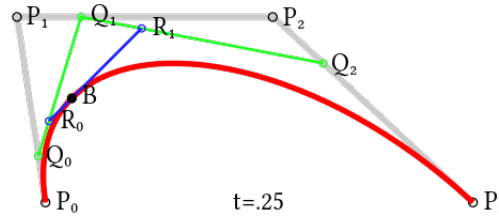


Figura 2: Exemplo de uma curva de Bézier

Sendo esta curva definida por uma equação, existe uma variável t , que varia entre $[0,1]$. O resultado da equação de qualquer t corresponde a uma determinada posição na curva. A equação é a seguinte:

$$B(t) = (1 - t)^3 * P0 + 3 * t * (1 - t)^2 * P1 + 3 * t^2 * (1 - t) * P2 + t^3 * P3$$

Onde **P0**, **P1**, **P2** e **P3** são os pontos de controlo. Resolvendo a equação várias vezes substituindo t , por vários valores entre 0 e 1 conseguimos descobrir toda a curva. A partir deste momento, já estamos capazes para desenvolver o algoritmo para resolver este problema. Este problema apresenta um princípio parecido com aquele que é utilizado nas curvas de Bézier, só que em vez de termos quatro pontos de controlo temos dezasseis.

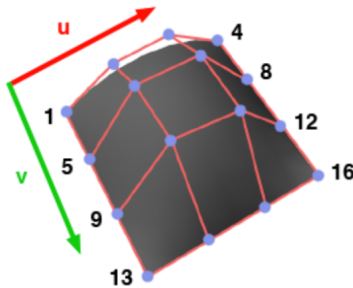


Figura 3: Bezier Patch e os seus pontos de controlo

Para tal iremos aplicar a fórmula de Bézier mas em vez de termos só um parâmetro (t), temos dois, um xx horizontal e yy vertical. Ambos variam entre 0 e 1.

Posteriormente aplicamos uma função que calcula a curva de Bézier para cada coordenada (xx, yy) no sentido de yy iterando até ($yy = \text{valor tecelagem}$), após esse cálculos é aplicada a fórmula de Bézier aos resultados obtidos anteriormente. Este ciclo pertence a outro ciclo que está a iterar sobre xx até ($xx = \text{valor tecelagem}$). Em cada iteração que xx quer yy , incrementam para ($1.0/\text{valor tecelagem} * \text{variável}$).

Após isso é só imprimir os pontos no respetivo ficheiro. É de salientar que quanto maior for o número de tecelagem melhor a figura irá ser.

4 Motor

4.1 VBOs

Tal como já referido anteriormente, uma das mudanças feitas no nosso trabalho foi o implementar os VBOs para desenhar os diferentes modelos. Os *Virtual Buffer Objects* são uma funcionalidade oferecida pelo OpenGL, os quais nos permitem inserir informação sobre os vértices diretamente na placa gráfica do nosso dispositivo. Estes fornecem-nos uma performance bastante melhor, devido ao facto de a renderização ser feita de imediato pois a informação já se encontra na placa gráfica em vez de no sistema, diminuindo assim a carga de trabalho no processador e assim os os pontos em vez de serem desenhados um a um, passando a estar todos guardados num *buffer*. Passando para a explicação do que foi alterado no nosso código, as maiores alterações foram na classe **Tranforms** que agora possui três novos métodos que serão usados nas funções *RenderScene* e *initGL*.

- **setVBO()** método que após criar e preencher um array com os pontos para desenhar os triângulos, preenche um *buffer* de forma ordenada com os pontos do array previamente criado.

- **draw()** método que desenha os triângulos com os pontos guardados no *buffer*.
- **toVertex()** método que através da função **glBegin(GL_TRIANGLES)** desenha todos os triângulos correspondentes aos pontos que se encontram no vetor desse objeto.

4.2 Curva Catmull-Rom

4.2.1 Rotação

Para podermos implementar as novas formas de rotação, foi necessário acrescentar uma variável **time** para podermos calcular o tempo que demora a fazer uma rotação de 360°. Para isto, aplicamos as seguintes fórmulas para aplicar a rotação ao objeto:

```
float r = glutGet(GLUT_ELAPSED_TIME) % (int) (subrot.getTime() * 1000);
float tempo = (r * 360) / (subrot.getTime() * 1000);
glRotatef(tempo, subrot.getX(), subrot.getY(), subrot.getZ());
```

Usamos a função *glutGet* na *renderScene* para determinarmos o tempo decorrido na execução do programa, mas, como este vai cada vez mais aumentando ao longo do tempo precisamos de estabelecer um limite para este valor, sendo que, usamos o resto da divisão pelo tempo dado e multiplicamos por 1000 (tempo em milissegundos). Conseguimos assim de seguida determinar a amplitude que o objeto vai rodar no momento simplesmente dividindo o valor calculado anteriormente multiplicado por 360 (graus) por o tempo multiplicado por 1000. Desta forma, o valor **tempo** é aplicado à função **glRotatef** fazendo o objeto rodar durante uma porção de tempo.

4.2.2 Translação

Além da nova forma de representar uma rotação, também foi criada uma nova representação para uma translação. Assim, foi necessário adicionar uma nova variável **time** e dois arrays auxiliares: *res[3]* (pontos para a próxima translação na curva), *deriv[3]* (derivada do ponto anterior). A inserção dos valores nos arrays foi feita na função **getGlobalCatmullRomPoint**. Os parâmetros da mesma são os dois arrays a preencher, em conjunto com uma variável **tempo** e os pontos dados no ficheiro XML. O valor tempo é calculado de maneira semelhante ao da rotação:

```
float t = glutGet(GLUT_ELAPSED_TIME) % (int) (trl.getTime() * 1000);
float tempo = t / (trl.getTime() * 1000.0);
```

$$M = \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad T = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix},$$

$$T' = \begin{bmatrix} 3*t^2 & 2*t & 1 & 0 \end{bmatrix},$$

Então, os arrays são preenchidos com o auxílio da função **getCatmullRomPoint** que obtém os valores para os mesmos através da multiplicação de uma matriz **M** por dois vetores e pontos:

Se **P** for o o vetor com os pontos, então a fórmula $T*M*P$ dá-nos os valores para preencher o array **res** e $T'*M*P$ resulta nos valores para preencher o vetor **deriv** com a derivada no ponto. Após a obtenção destes resultados podemos utilizar a função **glTranslatef** para aplicar a translação.

4.2.3 Desenhar as curvas Catmull-Rom

Para a implementação desta secção foi criada uma nova função denominada **encurvar** que, com o resultado da **getGlobalCatmullRomPoint** gera os pontos da curva a partir dos pontos dados no ficheiro XML. O resultado da auxiliar perminte-nos obter as coordenadas do próximo ponto da curva para um dado valor **t**. Desta forma, foi implementado um ciclo que passa por 100 pontos da curva. Por fim, é utilizada a função **renderCatmullRomCurve** que desenha a curva pretendida com a respetiva cor.

5 Resultados Obtidos

5.1 Teapot

Nesta parte apresentamos o *teapot* cujo ficheiro de input é fornecido no enunciado do trabalho prático.

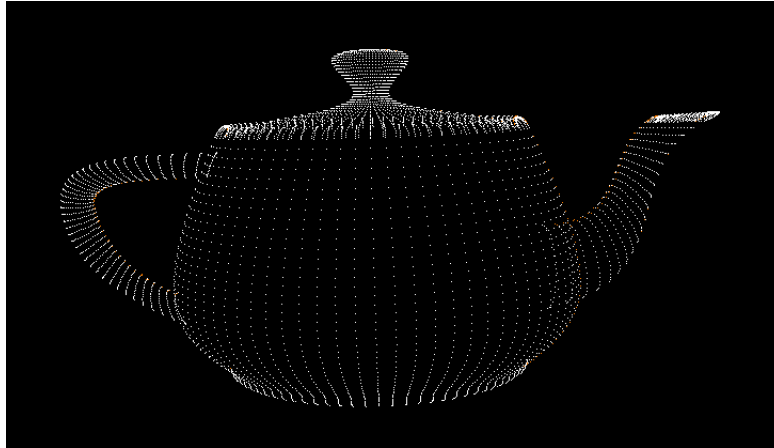


Figura 4: Teapot apenas com pontos (*Comando P*)



Figura 5: Teapot apenas com linhas (*Comando L*)

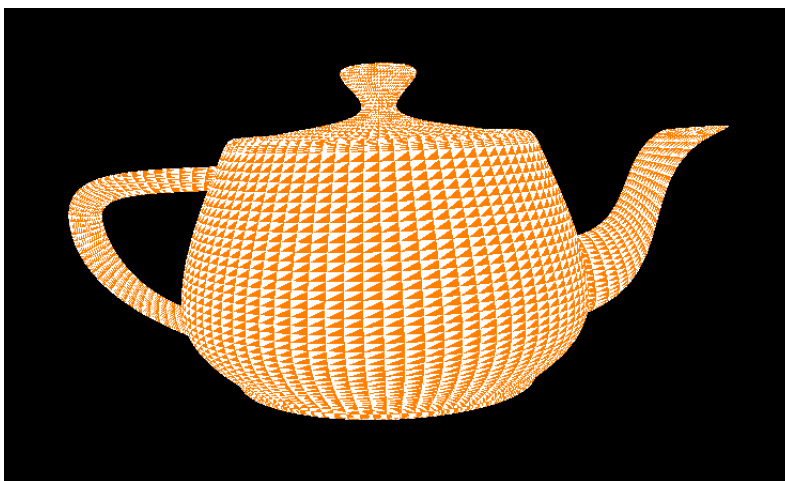


Figura 6: Teapot preenchido (*Comando F*)

5.2 Sistema Solar

Nesta secção visualizamos os resultados da criação do Sistema Solar.

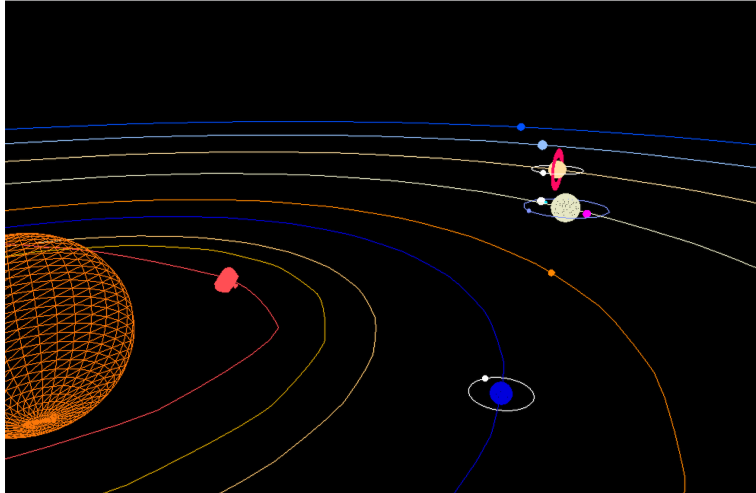


Figura 7: Visão aproximada do Sistema Solar

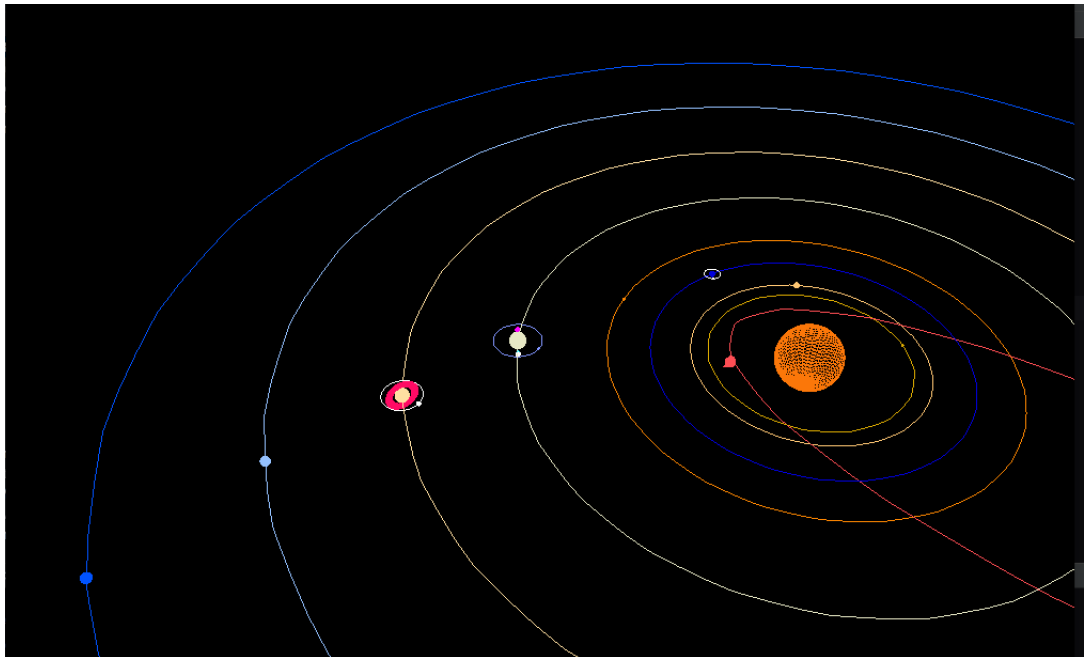


Figura 8: Visão geral do Sistema Solar

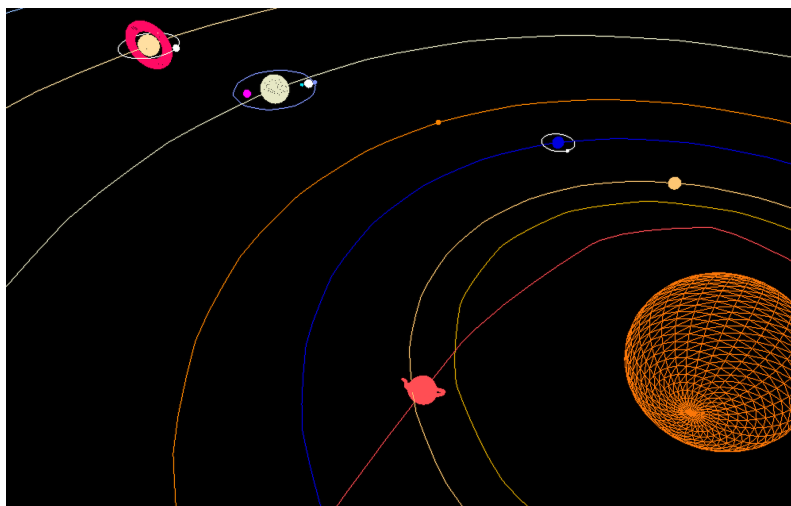


Figura 9: Visão geral aproximada do Sistema Solar

6 Conclusão

A realização desta fase do projeto revelou-se bastante mais trabalhosa e demorada, devido ao facto de ao contrário da fase anterior, esta ter bastante mais requisitos e estes apresentam um nível de complexidade bastante maior. No decorrer da elaboração do projeto tivemos diversas dificuldades, principalmente com tudo o que diz respeito com os Bezier patches pois é um tema que nunca tínhamos ouvido falar e que implicou que tivéssemos que fazer alguma pesquisa para conseguirmos, através desta, elaborar um algoritmo capaz de desenhar os novos modelos de forma correta. A implementação tanto dos VBOs como das Catmull-Rom curves foram mais um desafio que tivemos de contornar, no entanto com os conhecimentos adquiridos nas aulas práticas conseguimos implementá-los com sucesso. Posto isto, achamos que o resultado final desta fase está positivo pois, tal como era pedido no enunciado, conseguimos criar um Sistema solar dinâmico.