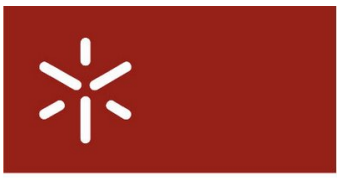


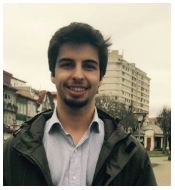
Universidade do Minho

Computação Gráfica

MIEI - 3º ano - 2º semestre
Universidade do Minho

Trabalho Prático - Parte 3


Guilherme Guerreiro
A73860


Dinis Peixoto
A75353


Ricardo Pereira
A74185


Marcelo Lima
A75210

22 de Maio de 2017

Conteúdo

1. *Introdução*

Foi-nos proposto, no âmbito da UC Computação Gráfica, a criação de um mini mecanismo 3D baseado num cenário gráfico sendo que para isso teríamos de utilizador várias ferramentas apresentadas nas aulas práticas entre as quais C++ e OpenGL.
Este trabalho foi dividido em quatro partes, sendo esta a terceira fase que basicamente tem como objetivo a inclusão de curvas e superfícies cúbicas ao trabalho anteriormente desenvolvido, tendo como finalidade a criação de um modelo dinâmico do Sistema Solar com um cometa incluído.

1.1 Resumo

Visto se tratar esta da terceira parte do projeto prático, é natural que se mantenham algumas das funcionalidades criadas na primeira e segunda fase e, por outro lado, algumas delas sejam alteradas e melhoradas, de modo a cumprir com os requisitos necessários.
Assim, esta fase traz consigo várias novidades que irão levar a várias alterações, tanto ao nível do engine como do generator.
Começando pelo generator, nesta fase, este passará a conseguir ser capaz de criar um novo tipo de modelo baseado em Bezier patches. Este passará a poder receber como parâmetros o nome de um ficheiro, no qual se encontram definidos os pontos de controlo dos vários patches, assim como, um nível de tecelagem e, a partir destes, retornará um ficheiro contendo uma lista de triângulos que definem essa mesma superfície.
Passando para o engine, este sofrerá várias modificações e, ao mesmo tempo, receberá também novas funcionalidades. Os elementos translate e rotate, presentes nos ficheiros XML, sofrerão algumas modificações. O elemento translation será agora acompanhado por um conjunto de pontos, que no seu todo, definirão uma catmull-rom curve, assim como o número de segundos para percorrer toda essa curva. Isto surge com o objetivo de passar a ser possível criar animações baseadas nessas mesmas curvas. Passando para o elemento rotation, o ângulo anteriormente definido poderá agora ser substituído pelo tempo, correspondente este ao número de segundos que o objeto demora para completar uma rotação de 360 graus em torno do eixo definido. Deste modo, terá que ser alterado não só o parser responsável por ler esses mesmos ficheiros, assim como, o modo como é processada toda a informação recebida com o intuito de gerar todo o cenário pretendido.
A última modificação, e não menos importante, que o engine sofrerá, está relacionada com os modelos gráficos, pois, estes agora passarão a ser desenhados com o

auxílio de VBOs, ao contrário da fase anterior, no qual estes eram desenhados de forma imediata.
Tudo isto tem como finalidade conseguirmos gerar eficazmente um modelo do Sistema Solar ainda mais realista do que o elaborado na fase anterior, pois este passará de um modelo estático a dinâmico com a implementação de todos estes novos requisitos.

2. *Arquitetura do código*

Visto este se tratar de uma continuação do trabalho desenvolvido nas fases anteriores, é normal que maior parte do código se tenha mantido inalterado e, por outro lado, alguma parte deste tenha sido modificado e outro tenha agora surgido, tendo em vista o cumprimento dos requisitos necessários propostos para esta fase.

2.1 Aplicações

Nesta secção são apresentadas as aplicações fundamentais que permitem gerar e exibir os diferentes cenários pretendidos. Na realização desta fase houve alterações significativas em ambas as aplicações do projeto, de modo a que os requisitos proprostos pela mesma fossem completamente concluídos.

2.1.1 Gerador

generator.cpp - Tal como explicado em fases anteriores, esta é a aplicação onde estão definidas as estruturações das diferentes formas/modelos geométricos a desenvolver de forma a gerar os respectivos vértices, para que, mais tarde, possam ser renderizados pelo motor (*engine*). Para além das primitivas gráficas anteriormente desenvolvidas, com a realização desta fase foi introduzido um novo método de construção de modelos com base em curvas de Bézier, sendo assim necessário acrescentar ao gerador novas funcionalidades relativamente às fases anteriores.

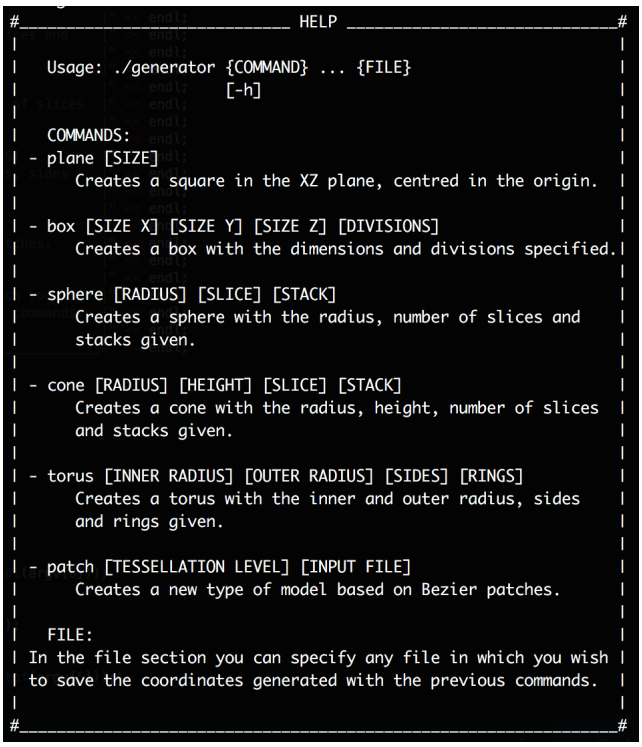


Figura 2.1: Menu de ajuda do Generator.

2.1.2 Motor

engine.cpp - O motor é a aplicação que possui as funcionalidades principais. Permite a apresentação de uma janela exibindo os modelos pretendidos e ainda a interação com estes através de diversos comandos. Com os requisitos implementados durante a realização da presente fase foram, inevitavelmente, surgindo alterações nesta aplicação relativamente à fase anterior.
As mudanças relativamente à estruturação do ficheiro de configuração (XML) obrigaram, uma vez mais, a pequenas remodelações nos métodos de *parsing* e consequentemente na estruturação dos dados armazenados durante leitura do respectivo ficheiro. A implementação de *Catmull-Rom Cubic Curves* e de *Vertex Buffer Objects*, por sua vez impulsionou também a mudanças nesta aplicação.

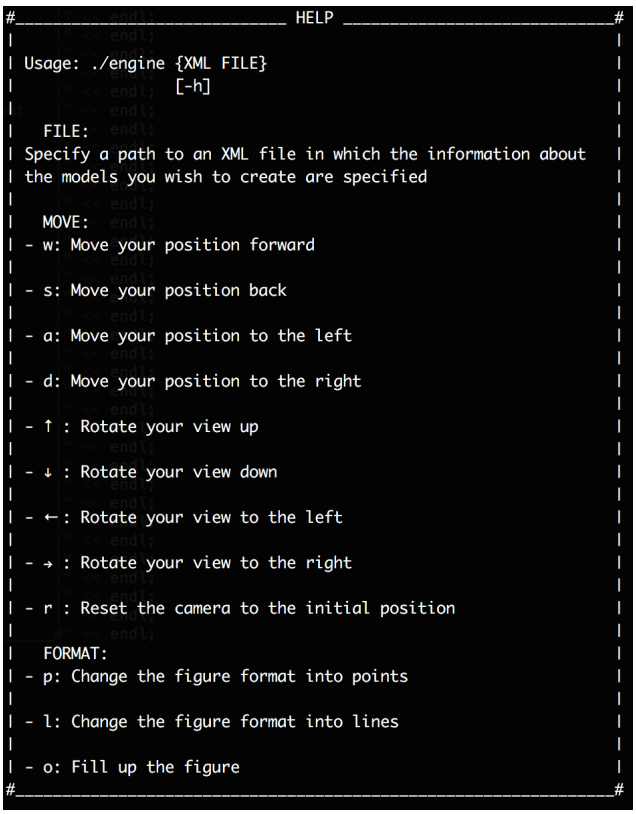


Figura 2.2: Menu de ajuda do Engine.

2.2 Classes

A realização desta fase foi distinta da outra neste ponto. Enquanto que durante a realização da fase anterior foi necessária a criação de inúmeras classes de modo a facilitar o processamento e organização de informação, esta fase é marcada pela alteração de parte destas e ainda pela criação de uma única.

2.2.1 Translation

Translation.h - Classe que armazena toda a informação necessária à execução de uma translação, sendo portanto obrigatória a existência das variáveis de instância x, y e z, representando o **vector aplicado na translação**. A implementação das curvas *Catmull-Rom* obrigou ainda à presença de um **conjunto de pontos** que a definam, em conjunto com um **tempo** correspondente ao número em segundos que demorará a percorrer toda a curva. O grupo optou pela criação de um **conjunto de pontos auxiliares**, representativo dos pontos resultantes da curva depois de efetuado o processamento desta. O conteúdo de cada um destes conjuntos de pontos será explicado detalhadamente mais adiante.

```
#ifndef __TRANSLATION_H__
#define __TRANSLATION_H__

#include <vector>
#include <math.h>

#include "Vertex.h"

using namespace std;

class Translation{

    float x;
    float y;
    float z;

    float time;
    vector<Vertex*> points_list;
    vector<Vertex*> points_curv;

    public:
        Translation();
        Translation(float,float,float,float);
        float getX();
        float getY();
        float getZ();
        float getTime();
        vector<Vertex*> getPoints();
        vector<Vertex*> getPointsCurv();
        vector<Vertex*> genPointsCurv();
        void getGlobalCatmullRomPoint(float, float*,float*, vector<Vertex*>);
        void setX(float);
        void setY(float);
        void setZ(float);
        void setTime(float);
        void setPoints(vector<Vertex*>);
        void addPoint(Vertex*);
        Translation* clone() const;
        virtual ~Translation();
};

#endif
```

Figura 2.3: Apresentação do ficheiro Translation.h.

2.2.2 Rotation

Rotation.h - Classe que armazena toda a informação pertinente à execução de uma rotação, sendo, assim, necessária a existência das variáveis de um plano cartesiano (x, y e z), representando o **eixo de aplicação** e ainda uma variável **ângulo** correspondendo ao ângulo de rotação sobre o vector. A implementação de movimento sobre os modelos exigiu a adição de uma variável **tempo** significando o número em segundos a percorrer uma rotação de 360º sobre o eixo especificado.

```
#ifndef __ROTATION_H__
#define __ROTATION_H__

class Rotation{

    float angle;
    float time;
    float x, y, z;

    public:
        Rotation();
        Rotation(float,float,float,float,float);
        float getAngle();
        float getTime();
        float getX();
        float getY();
        float getZ();
        void setAngle(float);
        void setTime(float);
        void setX(float);
        void setY(float);
        void setZ(float);
        Rotation* clone() const;
        virtual ~Rotation();
};

#endif
```

Figura 2.4: Apresentação do ficheiro Rotation.h.

2.2.3 Shape

Shape.h - Classe que armazena todo o conjunto de pontos necessários à representação de um determinado modelo, contendo, desta maneira um **conjunto de vértices** e o seu **nome**. Com a implementação de *VBOs* foi necessária a inclusão de um **buffer** destinado a tal.

```
#ifndef __SHAPE_H__
#define __SHAPE_H__
#include <string>
#include <vector>
#include <GL/glut.h>

#include "Vertex.h"

using namespace std;

class Shape{

    string name;
    GLuint vertex_buffer;
    vector<Vertex*> vertex_list;

    public:
        Shape();
        Shape(string,vector<Vertex*>);
        string getName();
        vector<Vertex*> getVertexList();
        GLuint getVertexBuffer();
        void prepare();
        void drawVertex3f(); // just for performance tests purposes
        void draw();
        virtual ~Shape();
};

#endif
```

Figura 2.5: Apresentação do ficheiro Shape.h.

2.2.4 Patch

Patch.h - Classe que armazena o **conjunto de pontos de controlo** associado a cada *patch* indispensáveis ao processamento das curvas de Bézier.

```
#ifndef __PATCH_H__
#define __PATCH_H__

#include <vector>

#include "Vertex.h"

using namespace std;

class Patch{

    vector<Vertex*> control_points;

    public:
        Patch();
        Patch(vector<Vertex*>);
        vector<Vertex*> getControlPoints();
        void setControlPoints(vector<Vertex*>);
        void addVertex(Vertex*);
        virtual ~Patch();

};

#endif
```

Figura 2.6: Apresentação do ficheiro Patch.h.

2.3 Ficheiros auxiliares

2.3.1 Figures

É neste ficheiro, **Figures.h**, que se encontram todos os métodos de criação de formas, isto é, todos os métodos utilizados pelo *generator* durante a criação de

uma dada figura, das disponíveis pelo mesmo:

- plano
- paralelipípedo
- cone
- esfera
- cilindro
- torus

Durante a realização desta fase foram adicionados alguns métodos, possibilitando este de gerar modelos a partir de patches fornecidos segundo um ficheiro de configuração com um formato específico.

```
#ifndef __FIGURES_H__
#define __FIGURES_H__

#define _USE_MATH_DEFINES

#include <math.h>
#include <vector>

#include "Patch.h"
#include "Vertex.h"

using namespace std;

// Figures
vector<Vertex*> createPlane(float size);
vector<Vertex*> createBox(float x, float y, float z, int div);
vector<Vertex*> createCone(float radius, float height, int slice, int stack);
vector<Vertex*> createSphere(float radius, int slice, int stack);
vector<Vertex*> createCylinder(float radius, float height, int slice, int stack);
vector<Vertex*> createTorus(float radiusIn, float radiusOut, int sides, int rings);

// Patches
Vertex* evalBezierCurve(float t, Vertex* p1, Vertex* p2, Vertex* p3, Vertex* p4);
Vertex* evalBezierPatch(float u, float v, vector<Vertex*> control_points);
vector<Vertex*> renderBezierPatch(int divs, vector<Patch*> patch_list);

#endif
```

Figura 2.7: Apresentação do ficheiro Figures.h.

3. *Generator*

3.1 Bézier patches

3.1.1 Processo de leitura do ficheiro input

Antes de explicar o processo de leitura do ficheiro de configuração (*input*), devemos primeiramente entender o formato do mesmo. O ficheiro apresenta um formato relativamente simples:

- Primeira linha contém o número de patches (*n_patches*).
- As restantes (*n_patches*) linhas contêm, para cada patch, uma sequência de 16 números correspondentes aos índices de cada um dos pontos de controlo constituintes desse mesmo patch.
- Segue-se uma linha contendo apenas o número de pontos de controlo (*n_points*).
- Por fim, estão representados cada um dos pontos de controlo fazendo corresponder os índices de 0 a *n_points*.

Durante o processo de leitura do ficheiro de configuração, foi preocupação do grupo criar um objeto **Patch** para cada patch lido do ficheiro, associando a este unicamente os seus 16 pontos de controlo correspondentes. Para tal, o primeiro passo foi efetivamente conseguir o número de patches presentes no ficheiro *input* (*n_patches*). Posteriormente, para cada um destes, fomos lendo os índices (*index*) e, sucessivamente recolher o respectivo ponto de controlo, através da leitura de uma linha (*line*) específica do ficheiro, dada pela equação:

$$line = 3 + n\_patches + index$$

Em que o 3 corresponde às linhas número de patches, número de pontos de controlo e ao facto dos índices começarem em 0 e não em 1.
Conseguido o ponto, restava apenas adicionar o mesmo à lista de pontos de controlo do respectivo patch.

3.1.2 Estruturas de dados

Assim como vimos na secção anterior, a estrutura de dados essencial ao armazenamento de informação para processamento dos *patches* passa simplesmente por associar, para cada um dos diferentes patches, os seus pontos de controlo.

```
#ifndef __PATCH_H__
#define __PATCH_H__

#include <vector>

#include "Vertex.h"

using namespace std;

class Patch{

    vector<Vertex*> control_points;

    public:
        Patch();
        Patch(vector<Vertex*>);
        vector<Vertex*> getControlPoints();
        void setControlPoints(vector<Vertex*>);
        void addVertex(Vertex*);
        virtual ~Patch();

};

#endif
```

Figura 3.1: Apresentação do ficheiro Patch.h.

3.1.3 Processamento dos *patches*

Para entender como é que funciona todo o algoritmo responsável por traduzir os Bezier patches para modelos geométricos, temos primeiro de saber o que são Bezier curves e como é que estas funcionam.
Para criar uma curva de Bezier precisamos apenas de 4 pontos. Estes pontos são designados pontos de controlo e estão definidos no espaço 3D, ou seja, são constituídos por 3 coordenadas: x, y e z. Visto que estamos apenas na posse de um conjunto de pontos, a curva propriamente dita ainda não existe até esta ser processada através da combinação destes mesmos pontos juntamente com alguns coeficientes.

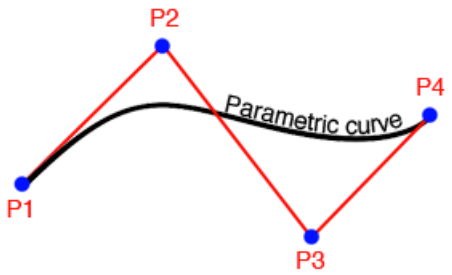


Figura 3.2: Bezier curve e os seus 4 pontos de controlo.

Ora, podemos tratar esta curva como uma curva paramétrica, isto é, definida por uma equação, e, portanto, é normal que esta contenha uma variável, que neste caso, é designada como parâmetro. O parâmetro é normalmente identificado pela letra t e, como estamos perante uma curva de Bezier, este varia entre 0 e 1. Descrevendo a equação da curva entre os valores de 0 a 1 é exactamente o mesmo que acompanhar a curva desde o seu início até ao fim. Desta forma, o resultado da equação para qualquer t contido no intervalo [0:1] corresponde a uma determinada posição no espaço 3D desta mesma curva. Assim, se nós queremos visualizar a curva paramétrica, tudo que precisamos de fazer é calcular o resultado da equação da curva à medida que vamos aumentando o valor de t em intervalos fixos, obtendo assim vários pontos da curva.

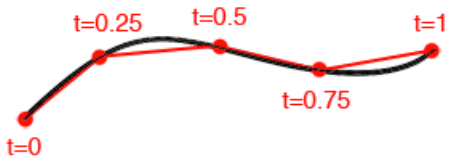


Figura 3.3: Curva com pontos calculados com apenas 4 valores de t.

Como tal, se quisermos obter um resultado mais uniforme e preciso, tudo o que temos de fazer é diminuir o tamanho desses mesmos intervalos e assim aumentar o número de pontos calculados.
No entanto, agora é necessário saber como é que podemos calcular estes tais pontos. Tal como dito anteriormente, a forma da curva corresponde ao resultado da combinação dos vários pontos de controlo juntamente com alguns valores. E como tal, surge assim a equação da curva:

$$P(t) = P1*k1 + P2*k2 + P3*k3 + P4*k4$$

P1, P2, P3 e P4 são os pontos de controlo da curva de Bezier e k1, k2, k3, k4 são os coeficientes que irão ponderar a contribuição de cada um desses pontos de controlo para o cálculo de uma dada posição da curva.
Como é fácil de supor, quando t=0, o resultado da equação, ou seja, o primeiro ponto da curva, coincide com o ponto de controlo P1 (k2=k3=k4=0). O mesmo acontece quando t=1, o resultado da equação, ou seja, o último ponto da curva, coincide com o ponto de controlo P4 (k2=k3=k4=0).
Por outro lado, quando o parametro t se encontra entre 0 e 1, o seu valor será utilizado para calcular os vários coeficientes, seguindo as seguintes equações:

$$k1(t) = (1-t) * (1-t) * (1-t)$$
$$k2(t) = 3(1-t)^2 * t$$
$$k3(t) = 3(1-t) * t^2$$
$$k4(t) = t^3$$

Desta forma, quando queremos calcular a posição da curva para um determinado t, somos obrigados a substituir o t nestas 4 equações, de forma a calcular os 4 coeficientes, os quais são depois multiplicados pelos 4 pontos de controlo.
Agora sim, estamos aptos a aprender o que são e como funcionam os Bezier patches. Na verdade, o princípio é parecido com aquele utilizado nas Bezier curves, no entanto, em vez de termos apenas 4 pontos de controlo, passamos a ter 16, os quais podem ser vistos como uma grelha de 4x4 pontos de controlo.

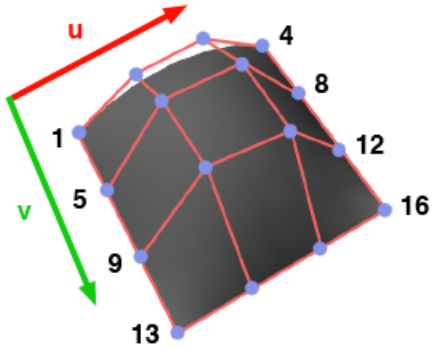


Figura 3.4: Bezier patch e os seus 16 pontos de controlo.

No caso das curvas, nós tínhamos apenas um parâmetro(t) para nos movimentarmos ao longo desta. Neste caso, passaremos a ter dois parâmetros: o parâmetro u, para nos movimentarmos na horizontal da grelha e o parâmetro v, para nos movimentarmos na vertical da grelha. Tal como em t, ambos estes parâmetros variam entre 0 e 1.
Resta agora saber como é que podemos calcular os pontos correspondentes às várias coordenadas (u,v) do nosso patch. Uma das maneiras mais comuns, e a qual nós utilizamos, passa por considerar cada linha da grelha 4x4 como Bezier curves independentes. De seguida, usaremos apenas um dos parâmetros(u) para calcular o ponto correspondente em cada uma destas curvas usando o algoritmo de cálculo anteriormente explicado, onde este u passará a ser tratado como o t do algoritmo anterior.

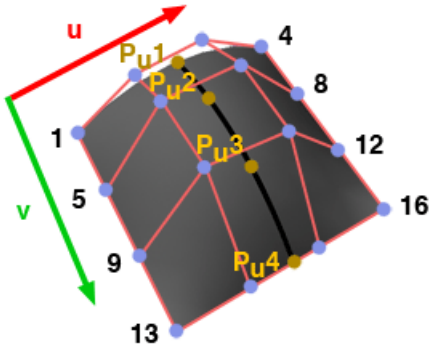


Figura 3.5: Pontos correspondentes a um dado valor de u para cada uma das curvas.

Através deste processo obtemos 4 pontos, os quais podem, desta forma, ser vistos como os 4 pontos de controlo de uma nova Bezier curve orientada na outra direção(v). Consequentemente, utilizando o segundo parâmetro v, podemos agora calcular, como da forma anterior, o ponto final definido por essa mesma curva. Este ponto corresponde assim à posição da superfície de Bezier para um dado par de valores (u,v).

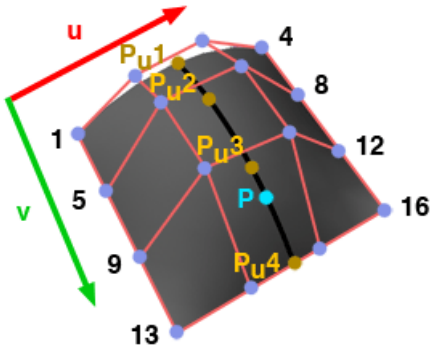


Figura 3.6: Ponto P final correspondente a um dado v e calculado a partir dos 4 pontos anteriormente encontrados.

Depois disto, já estamos completamente aptos a criar modelos baseados em Bézier patches. Dependendo do nível de tecelagem dado, serão calculados os pontos para os vários pares (u,v) dos vários patches e colocados num ficheiro numa dada ordem, de modo a formarem uma lista de triângulos que ilustre corretamente a superfície pretendida. Como é natural, quanto maior o nível de tecelagem, maior será o número de divisões de u e v e, portanto, maior será o número de pontos calculado, resultando numa superfície mais perfeita e próxima da realidade.

4. *Engine*

4.1 VBOs

Tal como dito anteriormente, uma das alterações que surgiu nesta fase foi a implementação dos VBOs para desenhar todos os modelos, os quais anteriormente eram desenhados de modo imediato.
VBO é a sigla para Vertex Buffer Object e esta é uma funcionalidade oferecida pelo OpenGL, a qual fornece métodos capazes de inserir a informação sobre os vértices diretamente na placa de vídeo do nosso dispositivo.
O principal objetivo dos VBOs é oferecer uma performance substancialmente superior aquela conseguida com o método de renderização imediato, primeiramente porque a informação reside na placa de vídeo em vez de na própria memória do sistema, podendo desta forma ser diretamente renderizada pela placa de vídeo, diminuindo significativamente a sobrecarga do sistema. Assim, através deste método, os fps(frames per second) observados serão inevitavelmente superiores àqueles que seriam observados utilizando a renderização imediata da fase anterior.
Em termos mais técnicos, para implementar VBOs foi necessário criar aquilo a que se chamam vertex buffers, que são nada mais do que arrays, nos quais serão inseridos todos os vértices que constituem o modelo que queremos desenhar.
Passando para a explicação do nosso código, basicamente tudo se processa na classe Shape, mais propriamente nas funcões Shape::draw() e Shape::prepare(). Esta última é responsável por criar e preencher esse mesmo array. Já a função draw() é a responsável por, depois dessa mesma informação já se encontrar na placa de vídeo, gerar o respectivo modelo, fazendo uso da função glDrawArrays disponibilizada pelo OpenGL.
Desta forma, já no engine, de modo a tornarmos tudo mais simples, no initGL é chamada a função initGroup, a qual irá invocar o prepare em todas as Shapes presentes no sistema e, desta forma, o renderScene só se encarrega de desenhar os vários modelos invocando a função draw().

4.2 Curva Catmull-Rom

4.2.1 *Rotate*

Para a implementação da nova forma de rotação, foi necessário adicionar uma variável *time* à classe **Rotation**. A nova variável indica o tempo, em segundos, necessários para uma rotação de 360 graus. Desta forma, para aplicar o movimento de rotação dos objectos, usamos as fórmulas:

r = (*Tempo decorrido desde a execução do glutInit*) $\%(time * 1000)$
$gr = (r * 360)/(time * 1000)$

Dado que o valor obtido pela função *glutGet(GLUT_ELAPSED_TIME)* é o tempo decorrido desde a execução do *glutInit*, este vai aumentar durante a execução do projecto. Por isso, é necessário, estabelecer um limite, usando o resto da divisão pelo tempo dado multiplicado por 1000 (tempo em milisegundos) ficando igual à unidade da função referida anteriormente. Com este valor, ao dividir pelo tempo multiplicado por 1000, dá-nos um valor decimal, que vai ser o coeficiente (porção) que ao mutiplicar por 360 (graus), dá a amplitude que o objecto vai rodar no momento. Desta forma, o valor de **gr** dá-nos o ângulo a aplicar a função *glRotate*, fazendo o objecto rodar uma certa porção num instante de tempo. Caso aumentemos o valor do tempo (*time*), o rodação torna-se mais lenta, e caso diminuirmos o valor do tempo, torna-se mais rápida.

4.2.2 *Translate*

Para além da rotação, também foi necessário implementar uma nova forma de translação. Para tal, foi necessário adicionar uma variável *time*, e uma *array up*[3] à classe **Translation**. A variável *time* corresponde ao tempo dado no ficheiro *.xml*, que representa o tempo, em segundos, necessário para o objeto completar uma volta na sua trajetória. O *array* é necessário para colocar e alinhar o objeto com a curva (trajétoria).
Para implementar a translação, foi necessário usar dois *arrays* auxiliares:

res[3] : ponto para a próxima translação na curva
$deriv$[3] : derivada do ponto anterior

Desta forma, para armazenar os valores nos respectivos *arrays*, utilizamos a função *getGlobalCatmullRomPoint*, que recebe os dois *arrays*, um valor **gt**, calculado previamente e o conjunto de pontos dados no ficheiro *.xml*. O valor de **gt** é conseguido como o valor *gr* no método da rotação, isto é, utilizando as seguintes fórmulas:

te = (*Tempo decorrido desde a execução do glutInit*) $\%(time * 1000)$
$gt = te/(time * 1000)$

Da mesma forma que na rotação, limitamos o valor recebido pela função *glutGet(GLUT_ELAPSED_TIME)*, e por fim, obtemos o valor de gt dividindo o valor por time*1000, obtendo um coeficiente que corresponde a uma porção da translação na curva.
Com isto, a função *getGlobalCatmullRomPoint*, preenche os *arrays* com os valores pretendidos, calculando o valor de t previamente e usando a função *getCatmullRomPoint* que obtém os valores para os *arrays*. A função *getCatmullRomPoint* para calcular os valores, utiliza a matriz M:

$$M = \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix},$$

Os vectores T e T':

$$T = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix},$$

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

Sendo P o array com os valores dos pontos, desta forma, com a fórmula $T * M * P$, obtemos os valores para o *array* **res**, obtendo assim as coordenadas do ponto.
Para além disso, para obter os valores para o *array* **deriv**, utilizamos a fórmula $T' * M * P$, obtendo a derivada no ponto.
Tendo os valores do *array* res podemos aplicar a translação, utilizando a função *glTranslatef*. No caso do *array* deriv, para aplicar a funcionalidade de o objeto seguir a orientação da curva, utilizamos a função *curveRotation* que utiliza os valores dos *arrays* deriv e up (definido inicialmente).

4.2.3 Desenhar curvas *Catmull-Rom*

Nesta secção, vamos falar na forma como implementamos o desenho das trajétorias dos objectos, como por exemplo, as órbitas dos planetas do sistema solar. Para tal, utilizamos a função *genPointsCurv*. A função, gera os pontos da curva a partir dos pontos recolhidos do ficheiro *.xml*. A geração dos pontos, passa por utilizar a função *getGlobalCatmullRomPoint*, que nos permite obter as coordenadas do próximo ponto na curva, para um dado valor t, como visto anteriormente. Desta forma, ao aplicar um ciclo, com o valor de t de 0 até 1, com o incremento de 0.01, passamos por 100 pontos da curva. Por fim com estes pontos, aplicamos uma função para desenhar as linhas que ligam os pontos, desenhando a pretendida curva (trajétoria do objeto).

5. *Resultados obtidos*

5.1 Bule

Apresentaremos, de seguida, o bule (*teapot*) gerado com o ficheiro de configuração de Patch do mesmo fornecido no enunciado da presente fase.

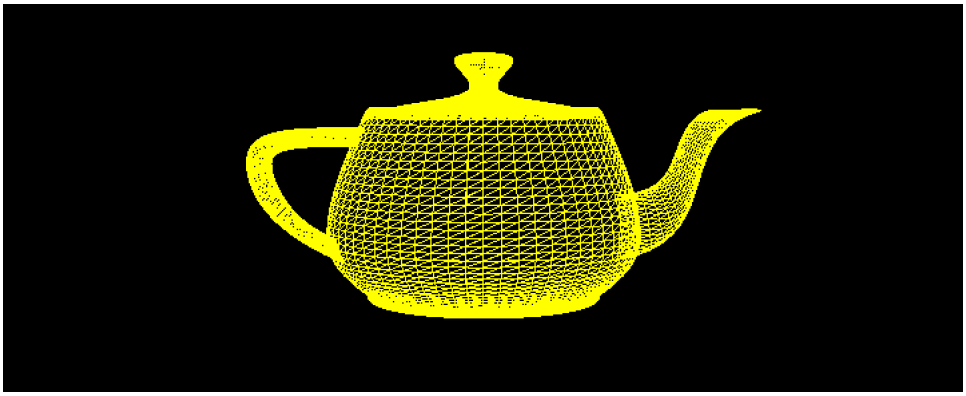


Figura 5.1: Visualização do *teapot* por linhas (comando L).

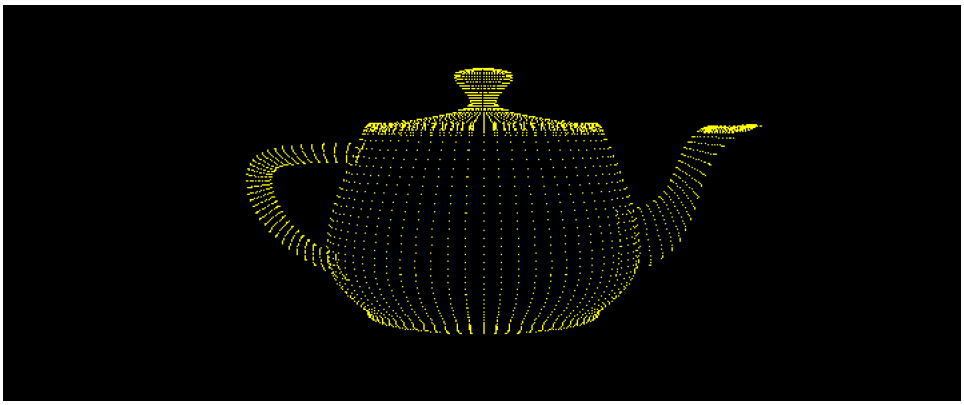


Figura 5.2: Visualização do *teapot* por pontos (comando P).

Figura 5.3: Visualização do *teapot* a preenchido (comando O).

5.2 Chávena e Colher

Para além do modelo anterior, o grupo preocupou-se em adquirir alguns exemplos extra de modo a poder testar mais completamente as funcionalidades das curvas de Bézier.

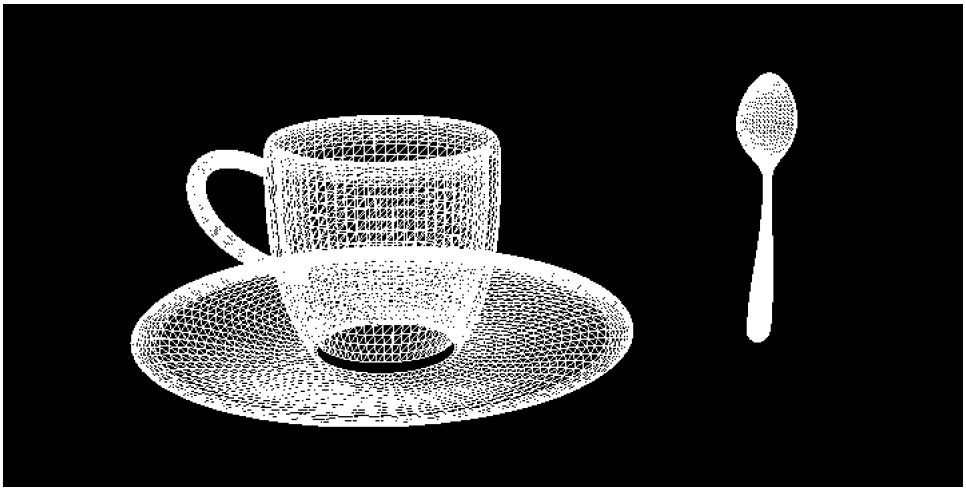


Figura 5.4: Visualização da chávena e colher por linhas (comando L).

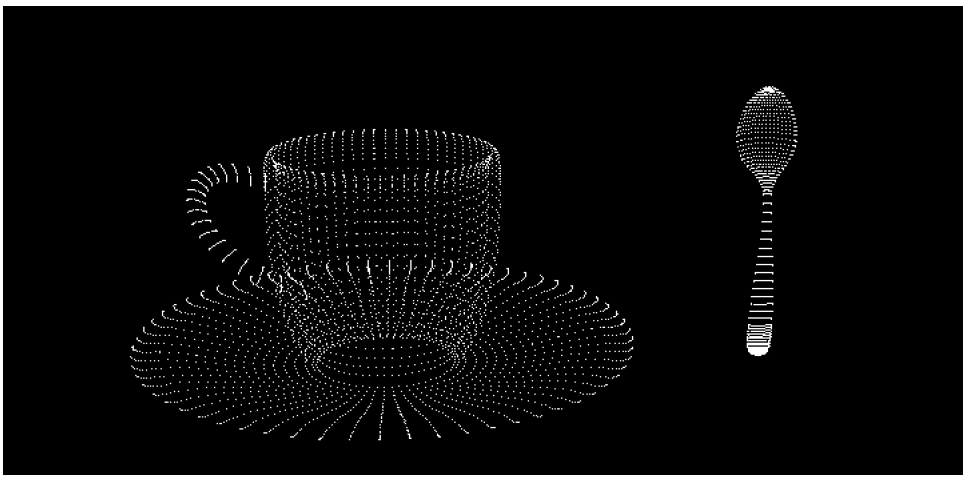


Figura 5.5: Visualização da chávena e colher por pontos (comando P).

Figura 5.6: Visualização da chávena e colher a preenchido (comando O).

5.3 Sistema Solar

O resultado final do sistema solar dinâmico, correspondeu ao esperado pelo grupo, todos os planetas foram representados o máximo à escala possível, os seus movimentos de rotação e translação foram representados tentando também aproximar o mais possível à realidade. Houve ainda a preocupação de adicionar um cometa gerado através de um ficheiro de configuração por *patches*.

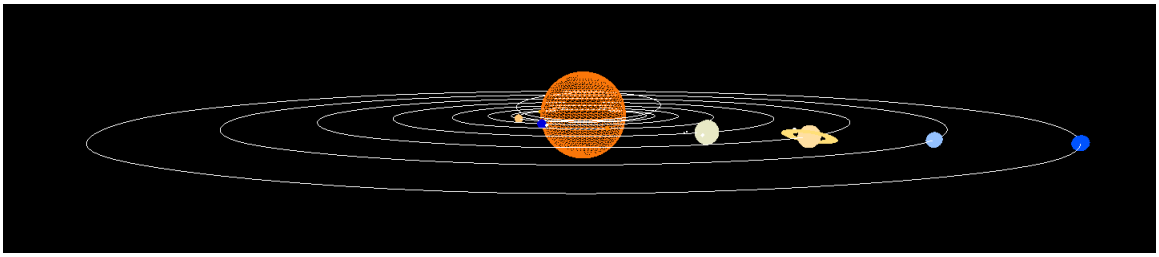


Figura 5.7: Visualização do sistema solar - perspectiva total.

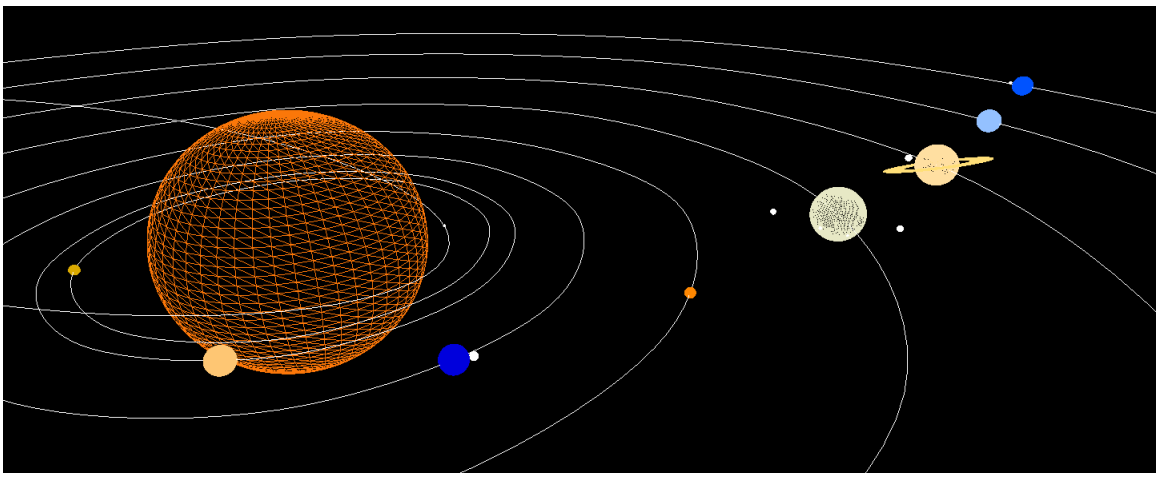


Figura 5.8: Visualização do sistema solar - perspectiva aproximada.

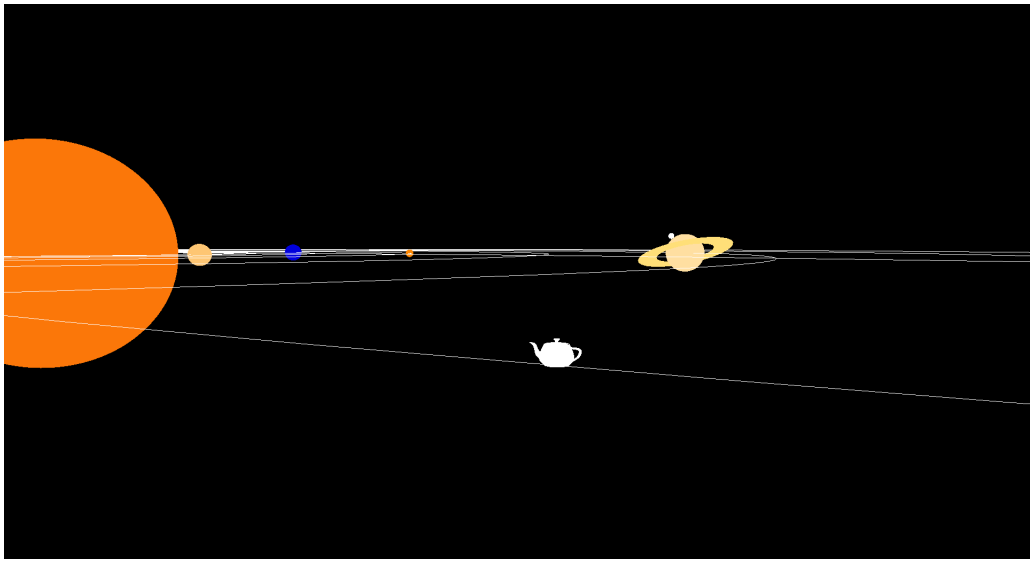


Figura 5.9: Visualização do sistema solar - perspectiva do cometa.

5.4 Boneco de Neve

De forma a aplicar ainda mais os nossos conhecimentos, decidimos criar um ficheiro xml que conseguisse gerar um boneco de neve. Nesta figura, foram utilizadas diversas formas geométricas previamente geradas, entre as quais o *torus*, a *teaspoon* e a *teacup*.

Figura 5.10: Visualização do boneco de neve.

6. *Conclusão/Trabalho futuro*

Ao contrário do que aconteceu na fase anterior, a elaboração desta terceira fase do projeto foi bastante mais demorada e complexa. Isto deve-se, não só ao facto de o número de requisitos desta fase ser relativamente superior, mas também ao nível de complexidade que estes apresentavam.
Durante a elaboração desta fase deparamo-nos com várias dificuldades. Uma das maiores dificuldades esteve relacionada com os Bezier patches e tudo aquilo que eles envolvem uma vez que nunca tínhamos trabalhado com estes e, desta forma, não possuíamos conhecimento suficiente para, a partir destes, elaborar um algoritmo capaz de projetar os modelos gráficos correspondentes. No entanto, depois de pesquisarmos bastante conseguimos chegar a uma solução que nos pareceu adequada e eficaz para o problema.
A implementação, tanto das Catmull-Rom curves como dos VBOs, foram outros dos desafios que tivemos de resolver, no entanto, com os conhecimentos adquiridos durante as aulas práticas, facilmente estruturamos os passos necessários para a sua correta resolução.
Assim, pensamos que o resultado final desta fase corresponde às expectativas, na medida em que conseguimos desenvolver um modelo, agora dinâmico, do Sistema Solar, tal como era pedido no enunciado.
Desta forma, esperamos que para a próxima e última fase que se avizinha, consigamos concluir o projeto de forma a obter um resultado final ainda mais realista e visualmente agradável possível.