Universidade do Minho
Escola de Engenharia

Computação Gráfica

3ª Fase

João Pedro da Santa Guedes A89588
Luís Pedro Oliveira de Castro Vieira A89601
Carlos Miguel Luzia Carvalho A89605
Bárbara Ferreira Teixeira A89610


A89588


A89601


A89605


A89610

2 de maio de 2021

Índice

1 Introdução

1.1 Contextualização

No âmbito da Unidade Curricular de Computação Gráfica foi nos pedido o desenvolvimento de um projeto dividido em 4 fases, com o objetivo de através de ferramentas como o C++ e o OpenGL criar um mini mecanismo 3D baseado num cenário gráfico. O relatório presente é referente à terceira fase, na qual o objetivo é 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.

1.2 Resumo

Este relatório diz respeito à terceira fase do projeto prático desenvolvido na unidade curricular de Computação Gráfica. Tratando-se da terceira fase, várias funcionalidades implementadas na primeira fase e na segunda foram mantidas e outras alteradas, de modo a melhor cumprirem os requisitos correspondentes a esta fase. Assim sendo, esta terceira fase traz consigo várias novidades relativas ao generator e ao engine.

Começando pelo generator, nesta terceira fase, este torna-se capaz de criar um novo tipo de modelo baseado em Bezier patches, passará a poder receber como parâmetros o nome de um ficheiro, sendo neste que se encontram definidos pontos de controlo dos vários patches, assim como um nível de tecelagem, retornando assim a partir destes o conteúdo de uma lista de triângulos que definem essa superfície.

Já no engine os elementos translate e rotate, presentes no ficheiros XML, sofreram algumas modificações, sendo que agora temos como objetivo a criação de algumas animações. Assim sendo, também os processos de parsing e processamento dos ficheiros terão de ser alterados. Por último, mas não menos importante, o engine sofrerá ainda outra alteração, sendo esta relacionada com os modelos gráficos, uma vez que estes serão agora desenhados com o auxilio de VBOs, ao contrario da fase anterior onde eram desenhados de forma imediata.

Todas estas alterações tem como finalidade a geração de forma mais eficaz de um modelo do Sistema Solar mais realista, uma vez que este passará com esta implementação a um modelo dinâmico.

2 Arquitetura do Projeto

Sendo esta fase a continuação das duas fases anteriores, grande parte do código se manteve inalterado, porém houve necessidade de modificar alguma parte deste e ou ainda acrescentar outro, tendo em vista os requisitos necessários a esta fase.

2.1 Aplicações

Nesta secção serão apresentadas as aplicações fundamentais que permitem gerar e exibir os cenários pretendidos. Na realização desta terceira fase existiram alterações significativas em ambas as aplicações do projeto (generator e engine) tendo em conta os requisitos propostos.

2.1.1 Generator

generator.cpp - Tal como enunciado nas fases anteriores, esta é a aplicação onde estão definidas as estruturações das diferentes formas/modelos geométricos a desenvolver de forma a gerar os respetivos vértices para serem renderizados pelo engine. Para além das primitivas gráficas anteriormente desenvolvidas, nesta fase foi introduzido um novo método de construção de modelos com base em curvas de Bezier, sendo então necessário acrescentar ao generator novas funcionalidades.

Figure 1: Menu Generator

2.1.2 Engine

engine.cpp - O engine é a aplicação que possui as funcionalidades principais, este permite a apresentação da janela exibindo os modelos pretendidos e ainda a interacção com estes através de comandos. Com os requisitos implementados durante a realização da presente fase foram surgindo alterações a esta aplicação relativamente á fase anterior. As mudanças relativamente à estruturação do ficheiro de configuração XML, obrigaram a pequenas remodelações no parsing de leitura e consequentemente na estruturação dos dados armazenados durante a leitura do respectivo ficheiro.

```
#------------** MENU **-------------#
|         Modo de utilização:        |
|         ./engine {file.xml}        |
|------------------------------------|
|               Teclas:              |
|------------------------------------|
|         Mudar Entre Câmeras:       |
|          Right Mouse Button        |
|------------------------------------|
|           Modo Estático:           |
|      Mover câmera: W A S D UA DA   |
|------------------------------------|
|         Modo First Person:         |
|    Mover câmera: Mouse + W A S D   |
|------------------------------------|
|          ativar GL_Point : P       |
|          ativar GL_Line : L        |
|          ativar GL_Fill : F        |
|      Ativar/Desativar Órbitas: O   |
|------------------------------------|
|       nota: .xml tem de estar      |
|        na pasta "/src/Files/"      |
|        nota2: UA -> Up Arrow       |
|       nota2': DA -> Down Arrow     |
|      nota2'': LA -> Left Arrow     |
|     nota2''': RA -> Right Arrow    |
#------------------------------------#
```

Figure 2: Menu Engine

2.2 Classes

A realização desta fase, ao contrário da anterior que exigiu a criação de enumeras classes, foi marcada pela alteração de parte destas com a exceção da

criação de uma única.

2.2.1 Transforms

transforms.h - Classe responsável por guardar as transformações realizadas num determinado modelo sendo assim necessária a existência de variáveis x,y,z

```
#ifndef GENERATOR_TRANSFORMS_H
#define GENERATOR_TRANSFORMS_H
#include <vector>
#include "point.h"

class Transform{
    float x_value;
    float y_value;
    float z_value;

    public:
        Transform();
        Transform(float x, float y, float z);
        float getX();
        float getY();
        float getZ();
        virtual void execute(){ };

};

class Translation : public Transform{
    float time;
    float up[3];
    std::vector<Point*> curve_points;
    std::vector<Point*> catmull_points;

public:
    Translation(float x, float y, float z, float t);
    void constructCurve();
    int getCatSize();
    void insertPoint(Point*);
    void drawCurve();
    void execute();
};
```

```
class Rotation : public Transform{
    float angle;
    float time;
public:
    Rotation(float a, float x, float y, float z, float t);
    void execute();
};

class Scale : public Transform{
public:
    Scale();
    Scale(float x, float y, float z);
    void execute();
};




#endif //GENERATOR_TRANSFORMS_H
```

Figure 3: transforms.h

2.2.2 Parser

parser.h - Classe responsável por dar parse, com o auxílio do tinyxml2, aos ficheiros que servem de input ao engine, e que utiliza como forma de guardar os dados a classe group, figure e transforms.

Para a realização desta fase, foi necessário alterar o parser, de modo a inserir corretamente todas as translações e rotações. Assim, aquando do parsing da translação, é verificado se existem pontos suficientes para a geração de uma curva de Catmull, adicionando a um vetor **orbits** caso esta condição se verifique.

Em relação à rotação, foi só preciso adicionar um parsing relativamente ao *time*.

```
#ifndef GENERATOR_PARSER_H
#define GENERATOR_PARSER_H

#include "tinyxml2.h"
#include <string>
#include <regex>
#include <fstream>
#include <sstream>
#include "point.h"
#include <iostream>
#include "group.h"


int readXML(string file, Group* group, vector<Translation*>* orbits);

#endif //GENERATOR_PARSER_H
```

Figure 4: parser.h

2.2.3 Camera

camera.h - Classe responsável por guardar as informações referentes a câmera como as reacções a eventos.

Nesta fase, decidimos generalizar um pouco mais a câmera de modo a que seja mais fácil, no futuro, acrescentar mais opções de câmeras. Assim, decidimos separar as 2 câmeras diferentes em 2 subclasses: **CameraStatic** e **CameraFPS**.

Para que esta modificação fosse bem sucedida, foram criados vários métodos obrigatórios para que a classe Engine não sofresse muitas alterações. Nesta classe principal, foi criado um array de câmeras (inicialmente constituído por 2), onde, com o auxílio de um inteiro **camOption** (0 se for estática, 1 se for first person) e do menu que altera esta variável, conseguimos com sucesso alternar entre as câmeras sem quaisquer problemas.

```
#ifndef GENERATOR_CAMERA_H
#define GENERATOR_CAMERA_H
#define _USE_MATH_DEFINES
#include "point.h"


class Camera{
    private:
        Point* position;
        Point* direction;

    public:
        Camera();
        Camera(float,float,float);
        Camera(float,float,float,float,float,float);

        virtual Point* getPosition();
        Point* getDirection();

        virtual Point* getFocus();
        void setPosition(float,float,float);

        virtual void move(unsigned char){};
        virtual void turn(float, float){};
        virtual void specialKey(int){};

};
```

```
class CameraFPS : public Camera{
    float yaw;
    float pitch;
    float speed;
    float rotationSpeed;
public:
    CameraFPS();
    void turn(float,float);
    void move(unsigned char );

};

class CameraStatic : public Camera{
    float alpha;
    float beta;
    float speed;
public:
    CameraStatic();
    void move(unsigned char);
    void specialKey(int key_code);
    Point* getPosition();
    Point* getFocus();

};

#endif //GENERATOR_CAMERA_H
```

Figure 5: camera.h

2.3 Ficheiros Auxiliares

2.3.1 Figures

figures.h - Classe responsável por armazenar os dados relativos a cada figura, sendo estes posteriormente utilizados pelo engine para desenhar as figuras.

A única alteração a esta classe nesta fase contribuiu para a simplificação da mesma, não precisando mais de passar um *GLenum* à função **draw**.

```
#ifndef GENERATOR_FIGURES_H
#define GENERATOR_FIGURES_H

#include <vector>
#include "point.h"
#ifndef __APPLE__
#include <GL/glew.h>
#endif

class Figure{
    private:
        float buffer_size;
        GLuint buffers[1];

    public:
        Figure();
        void setUp(std::vector<Point*> vertexes);
        void draw();
};

#endif //GENERATOR_FIGURES_H
```

Figure 6: figures.h

3 Generator

3.1 Bézier Patches

3.1.1 Processo de Leitura do ficheiro de Input

Antes de procedermos à explicação do processo de leitura do ficheiro de configuração (input), devemos entender o formato do mesmo. O ficheiro apresenta um formato relativamente simples:

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

No final do processo de leitura do ficheiro é inserido no array *toDraw* os pontos necessários para o desenho de cada patch. Depois de obtidos o número de patches (n_patches), vão sendo lidos os índices de cada ponto de forma a obter o respetivo ponto de controlo. Depois de obtido este ponto de controlo, resta somente inseri-lo no array *toDraw*.

3.1.2 Processamento dos patches

Para entender como 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.

Ora, podemos tratar esta curva como uma 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 estes valores é exatamente 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 da mesma curva. Assim, se queremos visualizar a curva paramétrica, tudo o que precisamos de fazer é calcular o resultado da equação à medida que aumentamos o valor de t em intervalos fixos, conseguindo assim obter os vários pontos da curva.

Como tal, se quisermos obter um resultado mais uniforme e preciso, devemos diminuir o tamanho desses mesmos intervalos e dessa forma aumentar o número de pontos calculados. No entanto, é necessário saber como podemos calcular estes pontos. Como referido previamente, a forma da curva é resultante da combinação dos vários pontos de controlo juntamente com alguns valores, surgindo assim a equação da curva:

$$p(x) = b0 * p1 + b1 * p2 + b2 * p3 + b3 * p4 \tag{1}$$

Na equação apresenta, p1,p2,p3 e p4 são os pontos de controlo da curva de Bezier e b0,b1,b2,b3 são os coeficientes que irão ponderar a contribuição de cada um desses pontos de controlo para o cálculo de uma dada posição na curva.

Quando t=0, como é fácil de supor, o resultado da equação, o primeiro ponto da curva, coincide com o ponto de controlo P1. O mesmo acontece quando t=1, o resultado da equação, o último ponto da curva, coincide com o ponto de controlo P4.

Por outro lado, quando t se encontra entre 0 e 1, o seu valor será utilizador para calcular os diferentes coeficientes da seguinte forma:

- it = 1.0 - t
- b0 = it * it * it
- b1 = 3 * t * it * it
- b2 = 3 * t * t * it
- b3 = t * t * t

Desta forma, quando queremos calcular a posição da curva para um determinado t, somos obrigados a substituir o t nas 4 equações, de forma a calcular os 4 coeficientes, que serão posteriormente multiplicados pelos pontos de controlo.

Agora podemos perceber o que são e como funcionam os Bezier patches de forma mais facilitada, sendo o princípio parecido com o mesmo das Bezier curves, passando a ter no entanto 16 pontos de controlo em vez de 4, mas que poderão ser vistos como uma grela de 4x4 pontos de controlo.

No caso das curvas, nós tínhamos apenas um parâmetro t para nos movimentarmos ao longo da mesma. No caso das patches passamos a ter dois parâmetros: $\boldsymbol{u}$ para nos movimentarmos na horizontal da grela e $\boldsymbol{v}$ para nos movimentarmos na vertical da grelha, sendo que ambos variam também entre 0 e 1.

Resta agora saber como poderemos calcular os pontos correspondentes às diversas coordenadas (u,v) do patch, sendo uma das maneiras mais comuns e a qual nós utilizamos, considerar cada linha da grelha 4x4 como Bezier curves

independentes. Fazendo uso do algoritmo previamente apresentado, substituiremos t por um dos parâmetros (u) para calcular o ponto correspondente em cada uma destas curvas.

Através deste processo, obtemos 4 pontos, os quais podem ser vistos como os 4 pontos de controlo de uma nova curva de Bezier orientada na direção de v. Utilizando agora como parâmetro v, calculamos o ponto final definido por essa mesma curva, o qual corresponde à posição da superfície de Bezier para um dado par de valores (u,v).

Por fim, estamos aptos para a criação de modelos baseados em Bezier patches e, dependendo do nível de tecelagem, *tess*, 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.

Quanto maior o nível de tecelagem, maior será o número de divisões de u e v e maior será o número de pontos calculados, resultando numa superfície mais perfeita e próxima da realidade.

4 Engine

4.1 VBOs

Uma das alterações que surigu 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 é oferecer uma perfomance significativamente 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 consequentemente a sobrecarga do sistema.Através deste método, os *frames per second* observados serão inevitavelmente superior àqueles que seriam observados utilizando o método de renderização da fase anterior.

Em termos mais técnicos, para a implementação efetiva dos VBOs, foi necessário criar aquilo a que chamamos de *vertex buffers*, que são nada mais do que arrays, nos quais serão inseridos todos os vértices que constituem o modelo a desenhar.

Em termos de código, tudo se processa na class *Figures*, mais especificamente nas funções Figure::setUp(std::vector¡Point*¿ vertexes), a qual recebe um array de vértices lido pelo parser do ficheiro correspondente à figura, e Figure::draw(), que estando a informação sobre os vértices disponibilizada na placa de vídeo, gera o respetivo modelo, fazendo uso da função *glDrawArrays* disponibilizada pelo OpenGL.

4.2 Curva Catmull-Rom

4.2.1 Rotação

De forma a implementarmos a animação de rotação de cada figura, foi necessário adicionar uma variável *time* a qual indica o tempo, em segundos, necessários para fazer uma rotação de 360 graus. De modo a auxiliar este cálculo usamos o valor obtido a partir da função *glutGet(GLUT_ELAPSED_TIME)*, que corresponde ao tempo decorrido desde a execução do *glutInit*.

De forma a aplicar o movimento de rotação dos objetos, usamos as seguintes fórmulas:

- elapsed = glutGet(GLUT_ELAPSED_TIME) % (int) (time*1000)
- ang = (elapsed * 360) / (time * 1000);

O resto da divisão pelo tempo dado multiplicado por 1000 serve como limitador do tempo, uma vez que este aumenta constantemente durante a execução do projeto. Com este valor, obtemos um decimal que será o coeficiente que,

ao multiplicar por 360, dá a amplitude que o objeto deverá rodar no momento. Desta forma, o valor de *ang* dá-nos o ângulo a aplicar a função *glRotate*.

No caso de aumentarmos o valor do tempo, a rotação torna-se mais lenta, e se diminuirmos, torna-se mais rápida.

4.2.2 Translação

Para além da rotação, também foi necessário implementar uma nova forma de translação. Para tal foi necessário adicionar, além da variável *time* previamente mencionada, um array *up*

3

, que servirá para alinhar o objeto com a curva (trajetória).

De forma a implementarmos a translação foi também necessário usar dois arrays auxiliares:

- pos[3] : ponto para a próxima translação na curva
- deriv[3] : derivada do ponto anterior

Desta forma, para armazenar os valores nos respetivos arrays, utilizamos a função *getGlobalCatmullRomPoint*, que recebe os dois arrays, um valor **gt**, calculado previamente, da mesma forma que é calculado o valor de **elapsed** na rotação e com as mesmas limitações, e o conjunto de pontos dados no ficheiro *.xml*.

Assim, 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 utilizar a matriz M:

$$M = \begin{bmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1.0 & -2.5 & 2.0 & -0.5 \\ -0.5 & 0.0 & 0.5 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \end{bmatrix}$$

E os vetores tt e dtt:

$$tt = \begin{bmatrix} t*t*t & t*t & t & 1 \end{bmatrix}$$

$$dtt = \begin{bmatrix} 3*t*t & 2*t & 1 & 0 \end{bmatrix}$$

Sendo **p** o array com os valores dos pontos, desta forma, com a fórmula tt * 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 dtt * 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 *curveRotate* que utiliza os valores dos arrays **deriv** e **up**, previamente definido.

4.2.3 Desenho das curvas Catmull-Rom

De seguida passaremos a explicar a forma como foi implementado o desenho das trajetórias dos objetos, como por exemplo, as órbitas.

Aquando do parsing, é passado como argumento um vetor de órbitas, para que sejam inseridos os vários pontos de cada curva que constam no **.xml**. Ao mesmo tempo é gerada a curva constituída por 100 pontos e guardada assim no vetor de pontos **curve_points** de cada translação para que, mais tarde, seja mais fácil o desenho da mesma.

No engine, é na renderScene onde é feito esse desenho: percorre-se o vetor com a chamada da função **drawCurve** que vai desenhar as órbitas com o *GL_LINE_LOOP*.

5 Resultados Obtidos

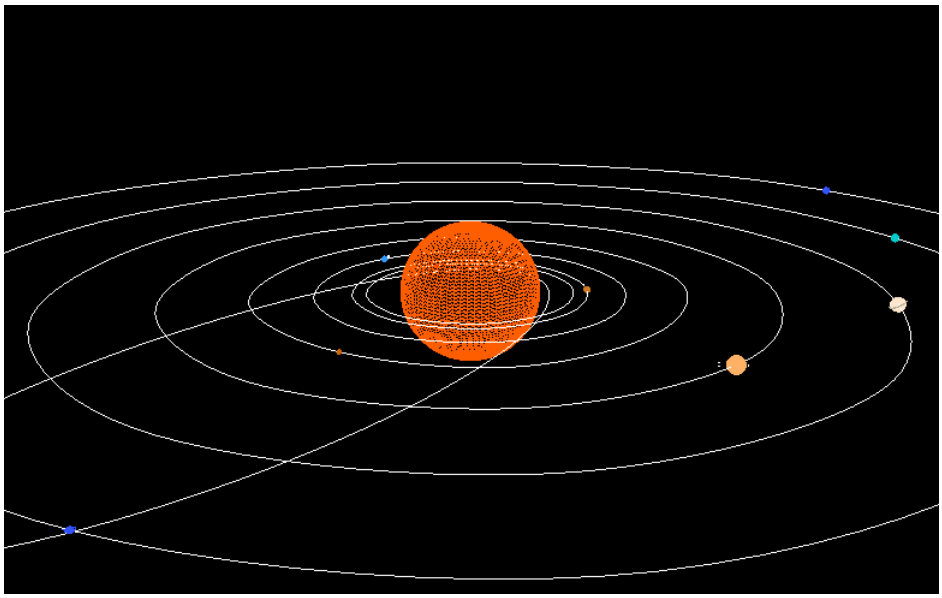

Figure 7: Sistema Solar

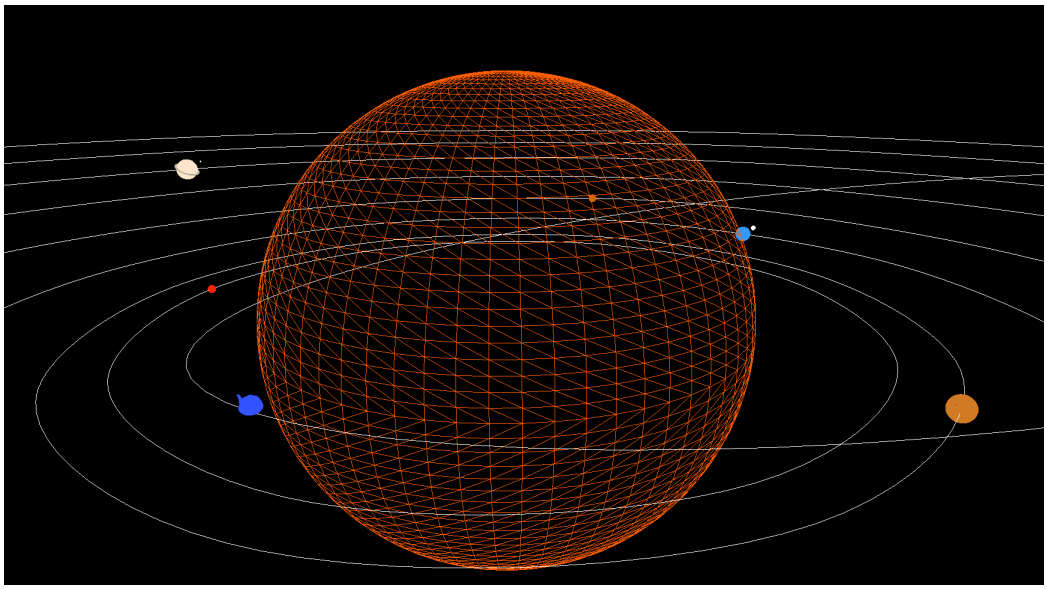

Figure 8: Sistema Solar Aproximado

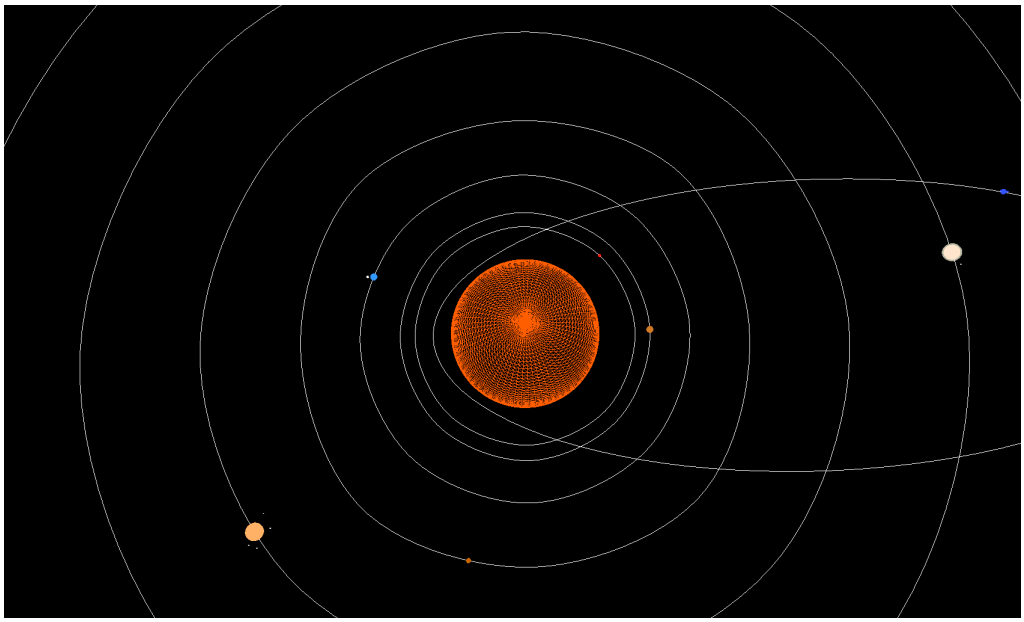

Figure 9: Sistema Solar visto de Cima

6 Conclusão e Trabalho Futuro

Ao contrário da fase anterior, a elaboração desta fase foi bastante mais complexa, devendo-se maioritariamente ao facto do nível de complexidade dos requisitos desta fase ser maior mas também ao maior número destes.

Durante a elaboração desta terceira fase deparamos-nos com várias dificuldades, sendo uma destas o facto de nunca termos trabalhado com Bezier patches sendo este um conceito novo para o grupo, assim como a implementação dos VBOs, porém com os conhecimentos adquiridos nas aulas práticas foi nos possível de melhor forma estruturar os passos necessários para a resolução dos requisitos apresentados nesta fase.

Assim sendo, o grupo considera o resultado final desta fase adequado, na medida em que conseguimos desenvolver um modelo dinâmico do Sistema Solar, assim como pedido no enunciado, desta forma esperamos que na próxima e última fase este modelo se torne cada vez mais realista e visualmente agradável.