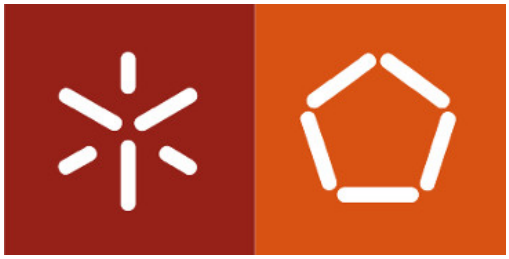

Universidade do Minho

Mestrado integrado em Engenharia Informática

Trabalho Prático - 3ª Fase

Grupo 35

André Rafael Olim Martins, A84347

Filipe Emanuel Santos Fernandes, A83996

José Diogo Xavier Monteiro, A83638

Susana Vitória Sá Silva Marques, A84167

Computação Gráfica

3º Ano, 2º Semestre

Departmento de Informática

2 de maio de 2021

Índice

1 Introdução

Nesta UC de Computação Gráfica, foi-nos proposto o desenvolvimento de um projeto dividido em 4 fases que consiste num mecanismo 3D. Nesta terceira fase, temos o objectivo de modificar o *Generator* de forma a que este consiga gerar modelos baseados nos **patches de Bezier** e o *Engine* nas rotações e translações de forma a que este use as curvas de **Catmull-rom**, desta forma criando as animações no modelo de demonstração. Nesta fase é também pedido a implementação VBOs para desenhar os modelos. O resultado será possível visualizar num ficheiro XML para demonstração de todas estas alterações.

2 Engine

Nesta terceira fase do trabalho o componente que passou por mais alterações foi o *Engine*, cujo qual teve que sofrer diversas modificações de forma a suportar diversas novas funcionalidades que foram pedidas.

O objetivo desta aplicação continua a ser o mesmo: permitir a apresentação de uma janela e exibição dos modelos requisitados, interpretando o ficheiro **XML**. No entanto, para obtermos os resultados pretendidos foram feitas algumas alterações nos métodos de *parsing* e consequentemente alterações nas medidas de armazenamento e também renderização.

2.1 Time Dependent Rotation

Implementar Rotações Dinâmicas ao longo do tempo implicou definir um novo tipo de transformação no nosso programa. De forma semelhante à Rotação Estática era necessário saber qual o eixo sobre o qual executar o movimento, mas agora em vez do ângulo exato, temos apenas um tempo, o qual nos indica a duração de uma rotação completa (360^{o}). Podemos assim considerar que ao longo do tempo consecutivas rotações são realizadas à medida que o ângulo é alterado,este pode aumentar ou diminuir dependendo do sentido da rotação.

Para isto aplicamos as seguintes fórmulas para a rotação do 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());
```

Usou-se a função *glutGet* na *renderScene* para determinarmos o tempo decorrido na execução do programa, mas, como este vai cada vez aumentando ao longo do tempo precisamos de estabelecer um limite para este valor, sendo que, utilizamos o resto da divisão pelo tempo dado e multiplicamos por 1000 (tempo em milisegundos). 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 esse determinado tempo.

2.2 Time Dependent Translation

Ao contrário das Rotações Dinâmicas, a implementação de Translações Dinâmicas representaram um maior desafio de adaptação, visto que não só era agora necessário ler mais informação do ficheiro fornecido ao *engine*, mas sobretudo porque as trajetórias destas translações passavam a ser definidas através de uma curva de *Catmull-Rom*. À semelhança dos *Patches de Bezier* (que iremos falar na secção do *Generator*) podemos simplesmente aplicar uma fórmula já definida para calcular os diversos pontos ao longo de uma curva de *Catmull*, com a única diferença que desta vez apenas incrementamos uma só variável.

Dado que esta translação evolui ao longo do tempo é assim necessário guardar constantemente o *gt* global, e à medida que avançamos no tempo incrementá-lo de forma a movermos-nos na curva. Ao fazer isto é também importante que este mantenha a sua orientação relativa a essa mesma curva, é assim preciso calcular a matriz de rotação que ajusta a rotação do objeto de acordo com a posição em que se encontra na curva.

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 *getGlobalCatmull-RomPoint*. Os parâmetros da mesma são os dois arrays a preencher, em conjunto com uma variável **tempo** e os pontos dados no ficheiro XML.

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

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

Figura 1: Curva Catmull-Rom

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 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 utilizador a função **glTranslatef** para aplicar a translação.

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);
```

2.3 Catmull-Rom Curves

Para a implementação do desenho das curvas foi criada uma nova função denominada **curve()** que, com o resultado da *getGlobalCatmullRomPoint* gera os pontos da curva a partir dos pontos dados no ficheiro XML (no ficheiro XML os pontos foram calculados recorrendo a coordenadas polares para o desenho das órbitas dos planetas e do cometa). O resultado da função permite-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.

2.4 Classes Alteradas

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 explicando-as.

2.4.1 Rotação

Nesta classe que guarda a informação referente a uma rotação e composta anteriormente por quatro variáveis **angle, x_eixo, y_eixo, z_eixo**, que identificam em qual dos eixos vai ser efetuada a rotação, adicionou-se nesta fase a variável **time** que representa o número em segundos que demora a fazer uma rotação de 360^{o} sobre o eixo especificado.

2.4.2 Translação

Nesta classe para além de se guardar, como anteriormente, toda a informação referente a uma translação (x_eixo, y_eixo, z_eixo), guarda-se ainda todos os pontos de controlo de uma curva (**vector<Vert> curvas**) e um conjunto de pontos finais de translação (**vector<Vert> verts**) após ser calculada a curva de maneira a cria-la com o método **CatmullRom**. Para além disso, como acontece com a classe **Rotação** adicionou-se também o campo **time** na classe e no ficheiro XML.

É nesta classe que se encontra todas as funções mencionadas acima para a contrução da *CatmullRomCurve*.

Os pontos que esta classe necessita são criados no ficheiro XML com ajuda de coordenadas polares para o desenho dos mesmos: gerou-se órbitas de planetas de forma circular desenhando pontos com diferença de 15 graus entre eles.

2.5 Performance

De modo a tornar mais eficiente o carregamento dos vértices para a placa gráfica foram utilizados VBO's . Para uma melhor noção da vantagem do uso de VBO's, corremos 3 *engine's*, um onde não eram utilizados VBO's, um *engine* onde eram utilizados VBO's para todos os objetos e, por fim, um *engine* onde não eram utilizados VBO's nas curvas *Catmull-Rom*. De forma a atingir os limites da placa gráfica foram utilizadas esferas com 120 camadas e fatias.
No primeiro, onde não foram utilizados VBO's, os vértices de todos os objetos estão sempre a ser carregados um a um para a gráfica, obtendo uma *framerate* de

24fps.

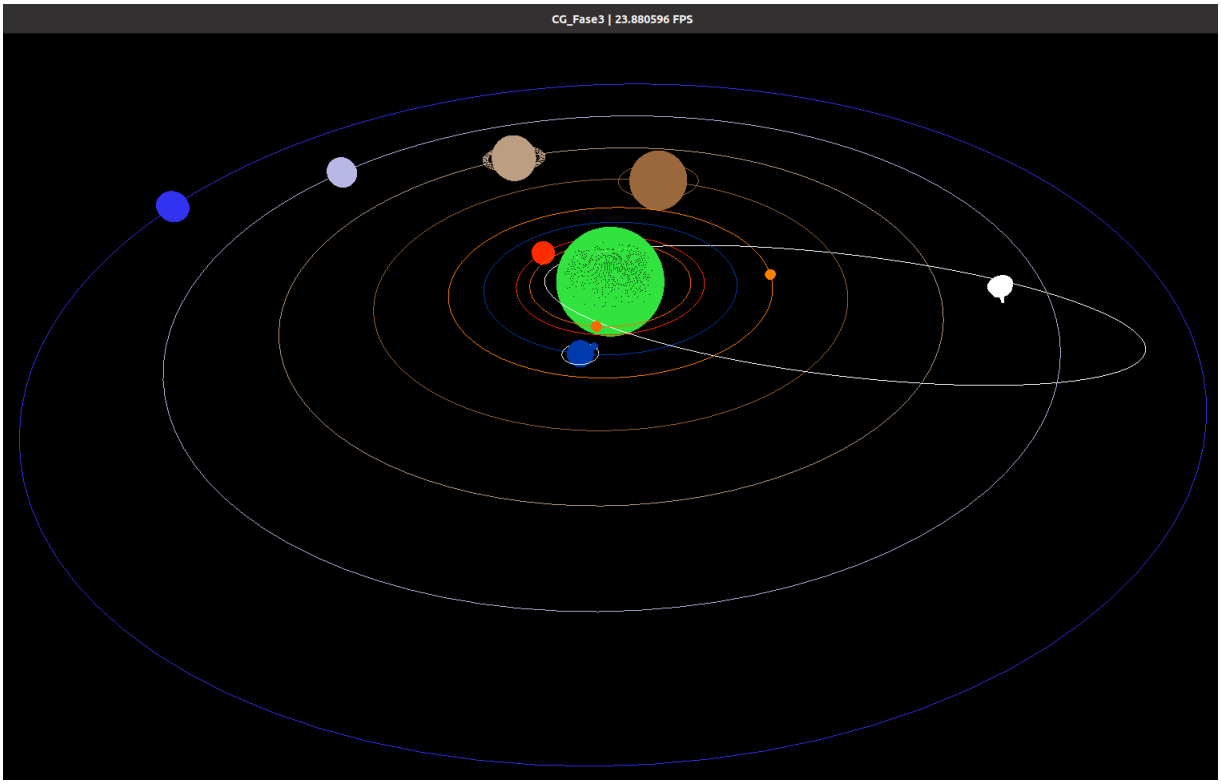

Figura 2: Sistema Solar, sem VBO's

Depois foi, utilizado o *engine* onde todos os objetos (planetas, luas, anéis, curvas *Catmull-Rom*, etc), eram carregados para a gráfica através de VBO's, isto é, os objetos são gerados no início do programa, depois do *parse*, e são guardados em buffer, depois na *renderScene* os buffers são carregados para a gráfica.
Houve uma clara melhoria no desempenho, obtendo cerca de 51fps. No entanto, como é possível ver na imagem, existe um *bug* na renderização das curvas *Catmull-Rom*, sendo visível uma espécie de raio para a curva da translação.

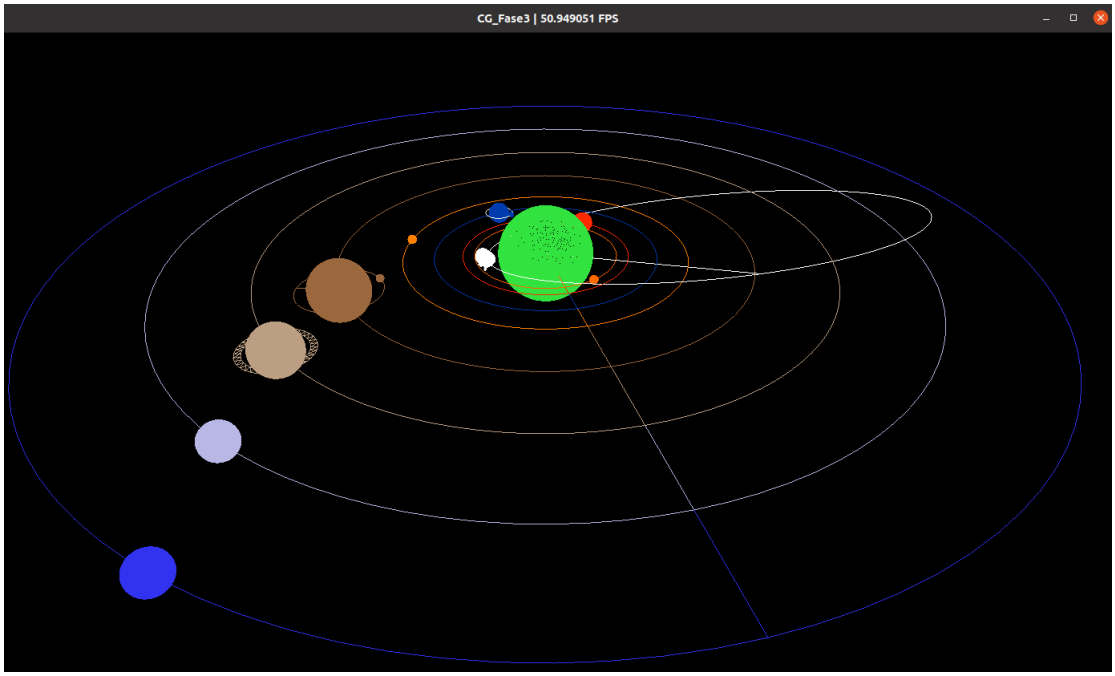

Figura 3: Sistema Solar, utilização de VBO's para todos os objetos

Por fim, foi utilizado o modelo apresentado como final. De forma a corrigir o *bug* do *engine* anterior, deixámos de utilizar os VBO's para as curvas *Catmull*, sendo estas enviadas para a gráfica vértice a vértice. De notar que devido a esta alteração houveram perdas significativas na performance (queda para os 36fps).

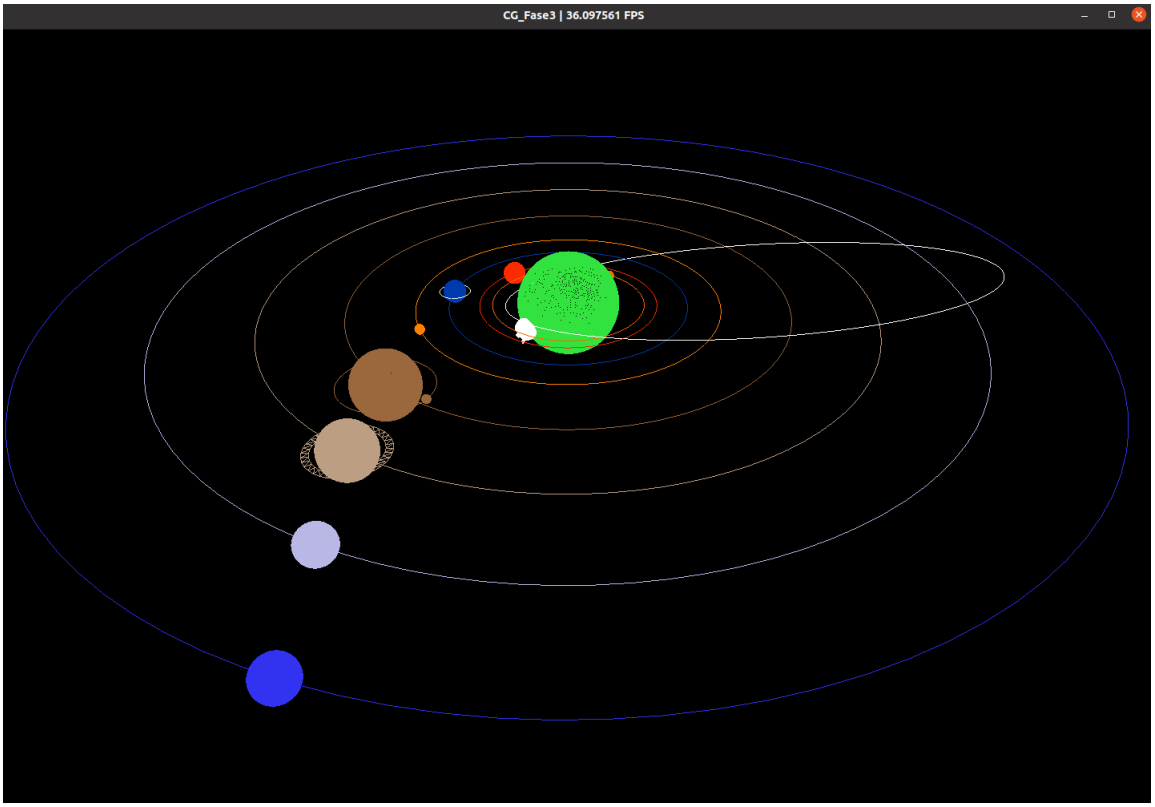

Figura 4: Sistema Solar, *Catmull* sem VBO's

3 Generator

Nesta 3ª fase do nosso projeto, foi nos pedido que realizássemos uma atualização ao nosso *Generator*. O foco desta atualização seria adicionar a capacidade de ler ficheiros .patch e, utilizando os métodos de ***Bézier*** leccionados, converter os valores recebidos num ficheiro .3d apto ao nosso *Engine*.

3.1 Criação de um Modelo com Superfícies de *Bézier*

A criação dos objectos 3D que utilizam as superfícies de *Bézier* passa pela utilização de ficheiros com a formatação indicada pelos docentes, no qual estão indicados todos os dados necessários, desde *patches* até pontos de controlo. De seguida, tendo sido compilado o *generator*, o método deve ser invocado com os seguintes valores:

```
generator bezierPatch {'file'.patch} {tessellation} {'file'.3d}
```

3.2 Curvas de *Bézier*

Na base da criação de uma curva de *Bézier* temos os 4 pontos de controlo, definidos num espaço 3D. Estes pontos serão os parâmetros que, quando aplicados na fórmula construtora da curva, multiplicados por um certo peso, irão estabelecer o formato da mesma. A seguinte equação ilustra esta interação que acabamos de explicar:

```
P(a) = P1*k1 + P2*k2 + P3*k3 + P4*k4
```

O parâmetro 'a' que aparece na equação acima será utilizado para calcular os valores dos coeficientes k1, k2, k3 e k4, através das seguintes equações:

```
K1(a) = (1-a)*(1-a)*(1-a)
K2(a) = 3(1-a)^2 * a
K3(a) = 3(1-a) * a^2
K4(a) = a^3
```

Variações no valor de 'a' serão utilizadas para calcular um ponto específico 3D ao longo da curva de *Bézier*.

3.3 Superfícies de *Bézier*

Agora que percebemos como se constrói uma curva de *Bézier*, vamos adicionar uma segunda dimensão, para passarmos a ter superfícies com 16 pontos de controlo, ou seja, 4 x 4. No exemplo das curvas, tínhamos a variável 't' para percorrer os valores ao longo da curva, neste caso, tendo adicionado outra dimensão, vamos precisar de dois: 'u' e 'v'. Decidimos considerar apenas a existência e definição de superfícies bicúbicas, ou seja, apenas as que são definidas através de 16 pontos de controlo. Para além de ser o tipo de geometria mais utilizada, vai de acordo com os valores fornecidos pelos docentes.

O cálculo de cada ponto de uma superfíce de *Bézier* irá depender de ambos os parâmetros 'u' e 'v', podendo ser definido como a soma de pontos de controlo e coeficientes vindos de ambas as curvas, ou seja, uma soma de curvas de *Bézier*.

3.4 Implementação de Mecanismos de *Bézier*

```
Vec3f calcBezierCurve(const Vec3f *P, float a) {
    float k1 = (1 - a) * (1 - a) * (1 - a);
    float k2 = 3 * a * (1 - a) * (1 - a);
    float k3 = 3 * a * a * (1 - a);
    float k4 = a * a * a;
    return P[0] * k1 + P[1] * k2 + P[2] * k3 + P[3] * k4;
}
```

Figura 5: Definição da Equação das Curvas de *Bézier*

```
Vec3f calcBezierPatch(const Vec3f *controlPoints, float u, float v) {
    Vec3f uCurve[4];
    for (int i = 0; i < 4; ++i) {
        uCurve[i] = calcBezierCurve(&controlPoints[4 * i], u);
    }
    return calcBezierCurve(uCurve, v);
}
```

Figura 6: Cálculo dos Pontos para a Superfície de *Bézier*

4 Demonstração

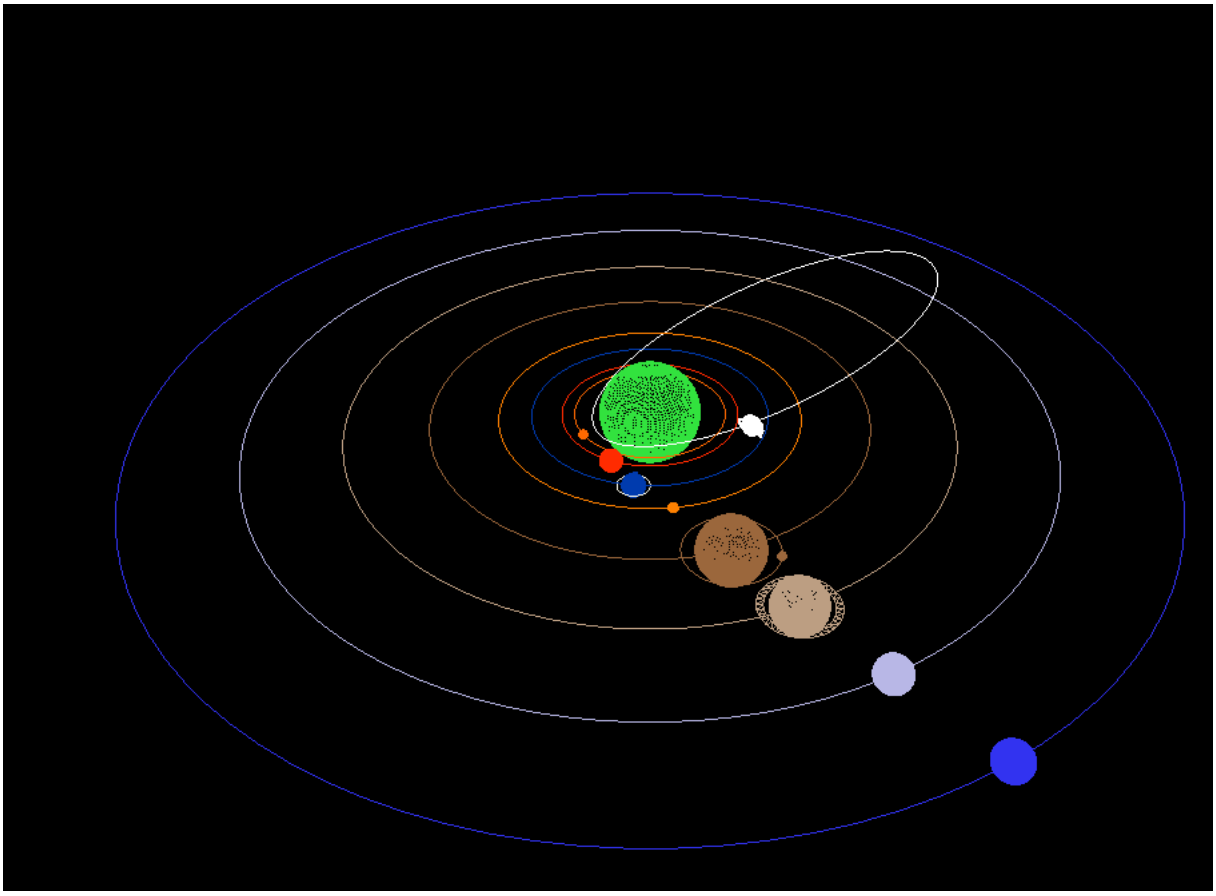

Figura 7: Demonstração do modelo SolarSystem6 no início

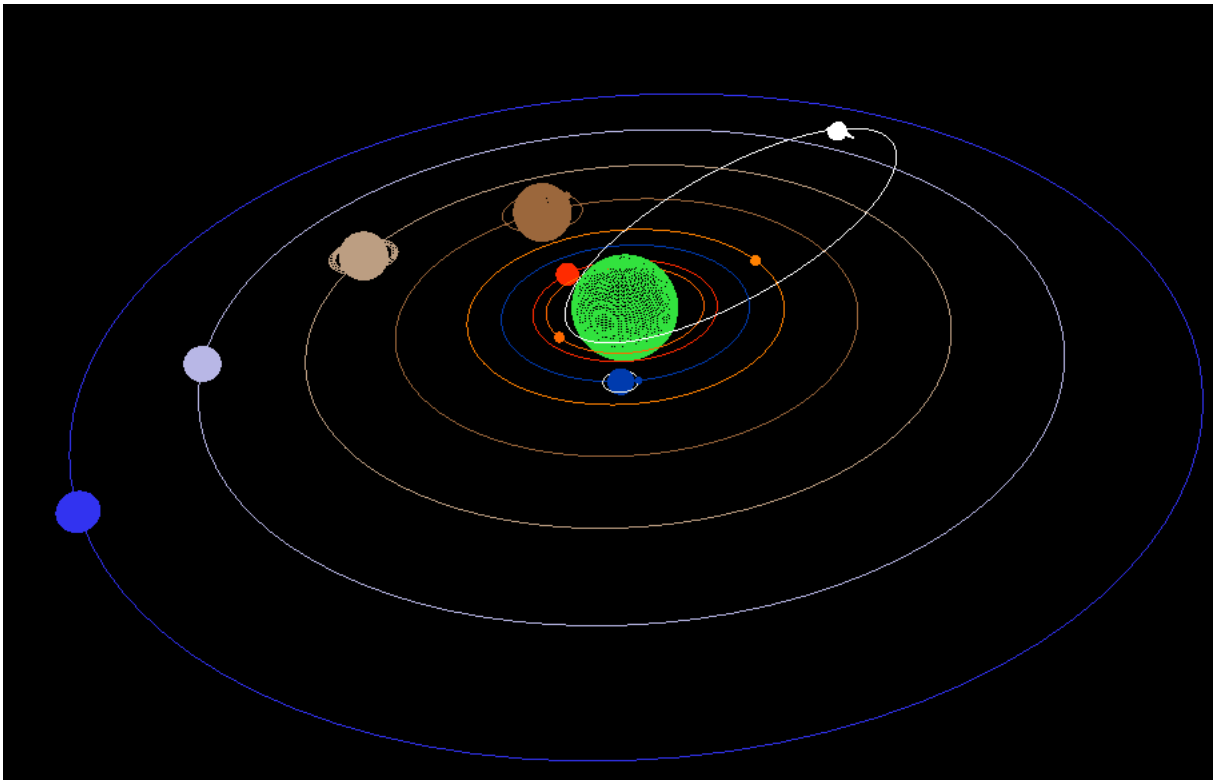

Figura 8: Demonstração do modelo SolarSystem6 após algum tempo

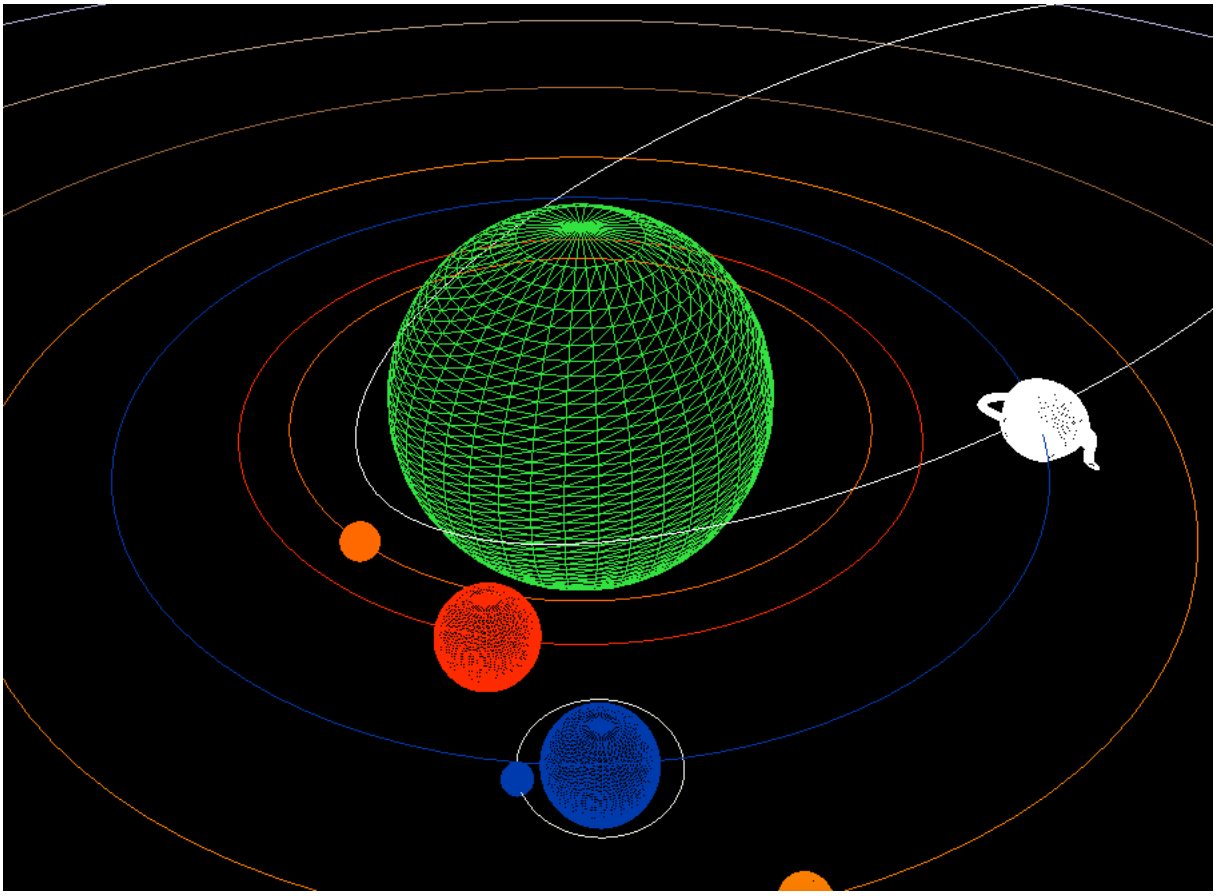

Figura 9: Demonstração da lua ao redor da terra e do cometa em forma de teapot

5 Conclusão

Nesta terceira fase do projecto, foi-nos pedido para realizar uma modificação ao *Generator* de forma a que este fosse capaz de produzir modelos baseados em **patches de Bezier**. Para o efeito usamos o ficheiro fornecido que constitui a informação necessária para gerar o **Teapot**.

No *Engine* foi modificado como eram feitas as rotações e translações de forma a incluir os requisitos necessários para o uso das curvas de **Catmull-rom**. Estas foram usadas para a animação dos modelos, mais especificamente no nosso modelo de demonstração, as translações dos planetas a volta do sol.

Além destas modificações também foi adicionado o mecanismo de **VBOs** para tornar o programa mais eficiente.

No ponto de vista do grupo, e após uma análise sucinta dos resultados obtidos, achamos que fomos capazes de pôr em uso os conhecimentos lecionados de forma a criar a demonstração e a implementar transformações de forma satisfatória.