Universidade do Minho

Departamento de Informática

Computação Gráfica

Phase 3 – Curves, Cubic Surfaces and VBOs

Grupo 39

Gonçalo Araújo Brandão A100663
Maya Gomes A100822
Luís de Castro Rodrigues Caetano A100893

26 de abril de 2024

Conteúdos

1 Introdução

A produção dos modelos deste trabalho prático em OpenGL envolve diversos temas, entre os quais,transformações geométricas, curvas, superfícies cúbicas, iluminação e texturas. Neste documento serão relatados os procedimentos da terceira fase do projeto, sendo mais orientada para as transformações geométricas e curvas.

Nesta fase, o objetivo é aprimorar tanto o gerador de modelos como o mecanismo de animação de um sistema solar dinâmico. O gerador agora pode criar modelos baseados em patches de Bezier, recebendo como entrada um arquivo com pontos de controlo e um nível de tecelagem desejado. O resultado será um arquivo com uma lista de triângulos para desenhar a superfície.

A implementação dos modelos agora deve ser feita com VBOs. O cenário de demonstração inclui um sistema solar dinâmico, com destaque para um cometa cuja trajetória é definida usando uma curva de Catmull-Rom e construído com patches de Bezier, como os pontos de controlo fornecidos.

2 Descrição do problema

Nesta terceira fase do projeto, a meta é aprimorar as funcionalidades do Gerador, em particular, desenvolver um novo modelo baseado em superfícies de Bezier. Em outras palavras, o programa deverá ser capaz de receber como entrada um ficheiro contendo uma série de pontos de controlo.

Quanto ao *Engine*, as operações de rotação e translação não serão mais restritas a movimentos estáticos. O parâmetro "angle"poderá ser substituído por "time", indicando o tempo necessário em segundos para completar uma rotação completa. Já o campo de translação poderá conter uma lista de pontos representando uma curva Catmull-Rom. Adicionalmente, será possível especificar um campo "align"para garantir que o objeto permaneça alinhado com a curva ao longo do movimento.

Finalmente, será necessário modificar o xml do sistema solar gerado na fase anterior de forma a incorporar um modelo dinâmico do sistema solar, inclusive com a inclusão de um cometa cuja trajetória será definida usando as curvas de Catmull-Rom.

3 Estrutura da aplicação

3.1 Resolução do problema

3.1.1 Novas estruturas

Nesta fase do trabalho, sendo que um dos objetivos era implementar as curvas de Catmull-row foram feitas algumas alterações nas estruturas do projeto.
Numa primeira fase, criou-se a estrutura *CatmullCurve* que representa os pontos de controlo da curva:

```
struct CatmullCurve {
    std::vector<Point> controlPoints;
};
```

Figura 1: Estrutura *Catmull*.

Com isto também foram adaptadas algumas estruturas anteriores:

```
struct World {
    Camera camera;
    vector<Group> groups;
    map<string, vector<Point>> points;
    vector<string> files;
    std::vector<CatmullCurve> catmullCurves;
};
```

Figura 2: Estrutura *World*.

```
struct Group {
    Transforms transform;
    vector<Model> models;
    vector<Group> groups;
    std::vector<CatmullCurve> catmullCurves;
};
```

Figura 3: Estrutura *Group*.

Com o intuito de podermos fornecer cores no ficheiro XML, renderizando as primitivas com diversas cores, foi criada a seguinte estrutura e adaptada a estrutura Transforms de forma a guardarmos a informação relativa as cores:

```
struct Color {
    float x,y,z;
    Color(float x, float y, float z) : x(x), y(y), z(z) {}
};
```

Figura 4: Estrutura *Color*.

```
struct Transforms {
    Color *color;
    Transform scale;
    Transform translate;
    Transform rotate;
```

Figura 5: Estrutura *Transforms*.

3.1.2 Camâra

Nesta fase, procuramos melhorar um pouco a nossa camâra implementando duas funções que nos permitem interligar a mesma com o rato.
Assim, também o rato é utilizado na manipulação da câmara, porém desta vez para modificar o vetor direção D. Os ângulos α e β referem-se à orientação horizontal e vertical da câmara, respetivamente.
Essa funcionalidade resultante do movimento do rato é possível devido a utilização das duas funções: *processMouseButtons()* e *processMouseMotion()*. Estas são validadas pela função glutMotionFunc do GLUT.
É importante notar que as mesmas ainda precisam de alterações e melhorias que, por falta de tempo, ficarão para a próxima fase.

3.2 Gerador

3.2.1 Bezier Patch

Leitura dos patches Inicialmente, procuramos ler o ficheiro patch fornecido pela equipa docente. Primeiro identificamos quantos patches tem o ficheiro e guardarmos os índices de cada um num vetor de vetores, cada um deles correspondente a um patch. De seguida, identificamos o número de pontos de controlo e guardamos as suas coordenadas x, y e z noutro vetor. Desta forma, temos os seguintes vetores:

$$vector < vector < int >> patches; \quad vector < Point > points;$$

Cálculo dos pontos Para começar é iportante referimos que a matriz M de bezier é dada pela expressão seguinte:

$$\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Como M é uma matriz simétrica a sua transposta é igual a ela mesma.

Para formar os pontos constituintes do *teapot* começou-se por percorrer cada patch formando em cada iteração a matriz,que é calculada através da expressão abaixo.

$$M \cdot \begin{bmatrix} P00 & P10 & P20 & P30 \\ P01 & P11 & P21 & P31 \\ P02 & P12 & P22 & P32 \\ P03 & P13 & P23 & P33 \end{bmatrix} \cdot M^T$$

Desta forma, a matriz será alterada ao longo das diversas iterações à medida que se percorre o vetor patches construíndo-se assim as curvas de bezier.

De forma a realizar as multiplicações de matrizes referidas anteriormentes foi criada a função *matrixMultiplication()*.

Para além disso, foi necessário calcular-se cada ponto de bezier. Considerando que u e v pertencem ao interavalo [0,1] temos que:

$$p(u,v) = (u^3 u^2 u1) \cdot matrix \cdot \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix}$$

Para cada ponto formado, os parâmetros u e v vão aumentando conforme o nível de tesselação estabelecido.

$step = 1 \div tesselagem$

De forma a multiplicar o vetor u pela matriz *matrix* e o resultado da mesma pelo vetor v foi criada a função *multMatrixVector()*. O resultado desta função é um ponto de bezier que será armazenado na matriz grid[i][j].

No final das iterações u e v, a matriz grid é composta por todos os pontos de bezier que constituem um patch.

3.3 Engine

3.3.1 Parsing dos ficheiros XML

Nesta fase do trabalho, mais uma vez foi necessário adaptar as nossas funções de parsing dos ficheiros XML, nomeadamente o parsing dos grupos. Assim, foi necessário adaptar a função *parseGroup()* de forma a ela ter em conta os parametros *time* e *align* e guardá-los. Nesta fase era fundamental ter a capacidade de ler e armazenar rotações e translações com o tempo. Para além disso, foi também necessário guardarmos os pontos de controlo do ficheiro XML.

3.3.2 VBO'S

Para a implementação dos VBO'S foram bastantes úteis as aulas práticas (e respetivo guião). Para tal, foi adaptada a nossa função *drawPoints()* de forma a ser gerado um buffer para armazenar os dados dos vértices. Assim, utilizamos as seguintes instruções:

```
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, pontos.size() * sizeof(Point),
             &pontos[0], GL_STATIC_DRAW);
glVertexPointer(3, GL_FLOAT, sizeof(Point), 0);
glDrawArrays(GL_TRIANGLES, 0, pontos.size());
```

3.3.3 Curvas de Catmull-Rom

As curvas de Catmull-Rom podem ser formadas por pelo menos 4 pontos(Po,P1,P2,P3), os quais manipulamos, em conjunto com uma iteração do valor t, através da equação abaixo apresentada:

$$\mathrm{p(t)} = (\mathrm{t}^3 t^2 t1) \cdot \begin{bmatrix} -0,5 & 1,5 & -1,5 & 0,5 \\ 1 & -2,5 & 2 & -0,5 \\ -0,5 & 0 & 0,5 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} P0 \\ P1 \\ P2 \\ P3 \end{bmatrix}$$

Desta forma, foi calculada a posição do objeto P(t) com base na matriz de Catmull-Rom, M, em 4 pontos pertencentes à curva e na matriz T, tendo em conta um instante t.
Finalmente, de forma a obter a posição do objeto foi criada a função *renderCatmullRomCurve()* que aplica as seguintes fórmulas:

$$A = P \cdot M \quad P(t) = A \cdot T$$

De seguida, para obter a matriz de rotação, precisou-se de calcular primeiro o seguinte:

$$X = \|P'(t)\|$$

$$Z = \frac{X \cdot Y_{i-1}}{\|X \cdot Y_{i-1}\|}$$

$$Y_{i-1} = \frac{Z \cdot X}{\|Z \cdot X\|}$$

Finalmente, depois de obter X, Y e Z bastou recorrer à *buildRotMatrix()* e a *glMultMatrixf()* para multiplicar a matriz de rotação pela atual. O cálculo de P'(t) sera também realizado em função da *renderCatmullRomCurve()*.

t global Outro parâmetro importante é o globalT. Este valor é calculado tendo em conta:

- **elapsedTime** - tempo que passou desde que o GLUT foi inicializado;
- **time** - intervalo de tempo que o objeto deve demorar a percorrer a curva;
- **actualTime** - tempo que passou desde que a volta foi iniciada;
- **turnCount** - número de voltas que o objeto já deu desde que o programa começou a correr.

$$actualTime = elapsedTime - time \times turnCount \quad globalT = actualTime \div time$$

Desenho da trajetória Finalmente, de forma a desenhar a trajetória foi criada a *renderCatmullRomCurve()*, função esta que recebe um conjunto de pontos (pertencentes à curva) e passa-os à *getGlobalCatmullRomPoint()* para calcular as diferentes posições dos restantes pontos que pertencem à curva.

Animações com Rotações Na função *animatedRotate()*, de forma a criar a rotação permanente de um objeto, utilizou-se a seguinte fórmula para calcular o ângulo da rotação em função do tempo que passou desde que o GLUT foi inicializado (elapsedTime):

$$\alpha = 360 \div (time * 100)$$

3.3.4 Desenho das primitivas

Para desenhar as primitivas com a nova estrutura de dados foi criada a função *drawGroups()*. Esta função itera sobre cada grupo presente na estrutura principal *world* e para cada um faz push da matriz das transformações através da função do glut *glPushMatrix*.

Posteriormente, percorremos o array das transformações e executamos por ordem que aparece no ficheiro, utilizando as seguintes funções: *glTranslatef*, *glScalef* e *glRotatef*.

No caso do *translate* e *rotate*, verificamos se haverá animações associadas aos mesmos e caso haja são chamadas as funções auxiliares *animatedTranslate()* e *animatedRotate()* , respetivamente. Depois desenhamos a respetiva figura deste grupo com os pontos armazenados com ajuda das funções *drawPoints*. Antes de fazer pop da matriz voltamos a chamar recursivamente a função *drawGroups()* para executar as transformações dos subgrupo. Assim garantimos que os subgrupos herdam as transformações do grupo pai.

3.3.5 Resultados

É também importante notarmos que a curva pode váriar consoante o valor de tecelagem atribuído.

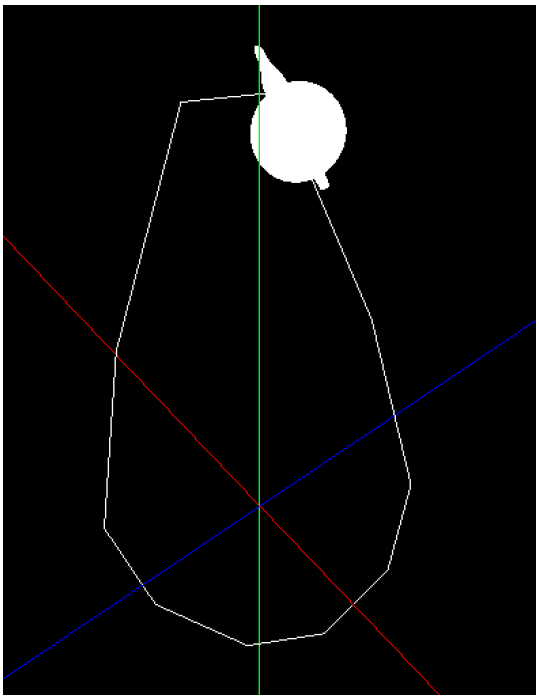

Figura 6: Resultado test_3_1 com tecelagem t += 0.1.

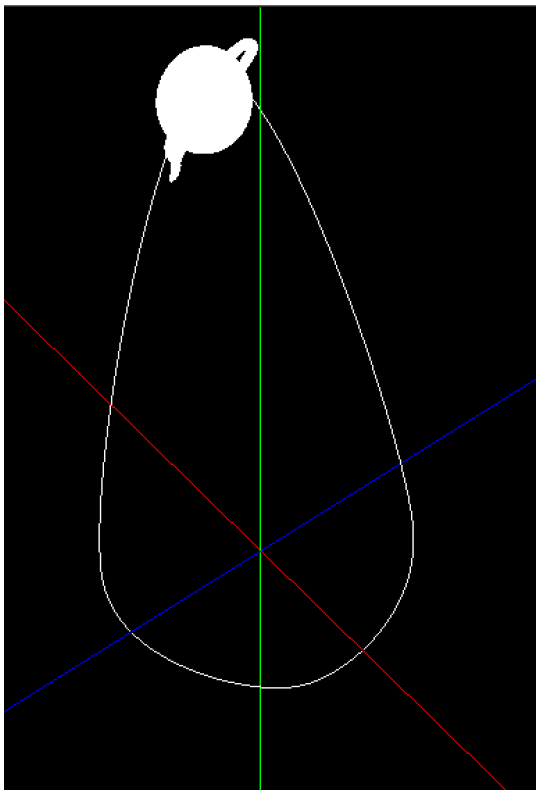

Figura 7: Resultado test_3_1 com tecelagem t += 0.001.

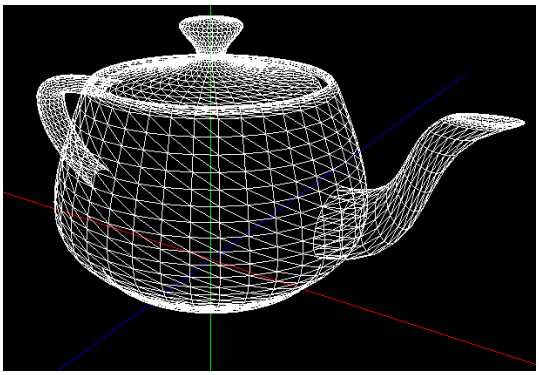

Figura 8: Resultado test_3_2 com tecelagem 10.

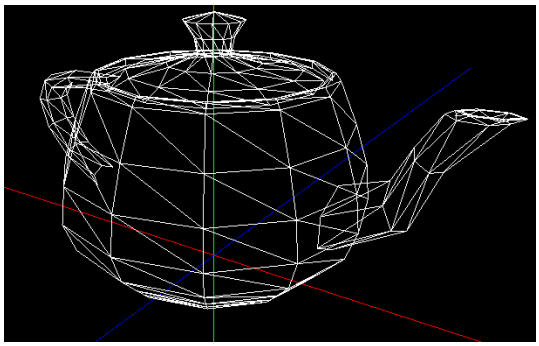

Figura 9: Resultado test_3_2 com tecelagem 3.

3.4 Sistema Solar

Nesta fase, o modelo do sistema solar, descrito no ficheiro XML, deverá incluir o Sol, os oito planetas, a lua e um cometa. De forma a tornar o modelo o mais realista possível, será também representado o anel de Saturno. O cometa, por sua vez, irá usar o modelo de um *teapot* desenvolvido com a primitiva que processa o*Bezier Patches*.

Tal como já referido na segunda fase, para a representação do sistema solar será preciso recorrer a transformações geométricas (*translate, scale e rotate*) de forma a representar as primitivas o mais próciximo da realidade. No entanto, devido a existir imensa discrepância entre as dimensões reais dos diferentes elementos do sistema, já desde a segunda fase do trabalho, foi necessário a utilização de diversas escalas e aproximações de modo a tornar a visualização mais adequada.

Para implementar a animação no sistema solar começamos por determinar os pontos das curvas de Catmull Rom que irão representar as órbitas dos planetas. A nossa ideia foi escolher a distância do sol ao planeta como raio para a órbita do planeta em questão. Assim, para auxiliar decidimos criar um script em python que nos permitisse gerar os 16 pontos que formam a órbita de cada planeta.

Neste script os pontos são gerados num círculo ao redor do eixo x, todos com coordenadas y = 0. O raio do círculo é dado pela variável 'd'. O primeiro ponto é gerado na posição x = d e os pontos subsequentes são gerados em posições ao longo do círculo com base em ângulos uniformemente espaçados.

No caso do cometa, os pontos seguem uma trajetória um pouco mais complexa definida por uma equação elíptica. O primeiro ponto é gerado na posição (h + a, k), e os pontos subsequentes seguem uma trajetória elíptica calculada com base nos ângulos uniformemente espaçados.

Uma vez as curvas das órbitas calculadas, falta estabelecer o tempo de translação de cada planeta. Numa primeira fase, analisamos os valores reais e como não podemos aplicar uma só escala para representar a realidade, pois a discrepância é enorme entre os planetas, escolhemos qual seria o valor mínimo para percorrer uma órbita e fomos a calcular uma diferença crescente para os próximos planetas. Como valor base, escolhemos Mercúrio=10s.

```
Mercurio : 10
Venus : 10 + 1 * 1,5 = 11.5
Terra : 11.5 + 2 * 2 = 15.5
Marte : 23 + 4 * 3 = 35
Jupiter : 35 + 5 * 5 = 60
Saturno : 60 + 6 * 5.5 = 93
Urano : 93 + 7 * 6 = 135
Neptuno : 135 + 8 * 6.5 = 187
```

Quanto às rotações dos planetas sobre si próprios, o método que utilizamos para calcular os tempos é semelhante ao das translações. Note-se que em caso de rotação contrária o parâmetro time é negativo. Como valor de referencia, utilizamos o tempo de Saturno completar uma rotação sobre si mesmo. Atribuíndo o valor de 5 segundos.

```
Jupiter: 5
Saturno: 5 + 1 * 1.5 = 6.5
Neptuno: 5 + 2 * 2 = 9
Urano: 5 + 2 * 2.5 = -10
Terra: 6 + 3 * 3 = 15
Marte: 6 + 3 * 3.5 = 16.5
Sol: 16.5 + 5 * 5.5 = 44
Mercurio: 44 + 6 * 6 = 80
Venus: 80 + 7 * 6.5 = -125,5
```

Relativamente ao cometa, decidimos estabelecer uma trajetória aleatória e foi também utilizado o modelo correspondente ao teapot.

Para uma melhor visualização dos planetas decidimos representá-los com cores diferentes, que seriam já fornecidas no XML. Para tal, foi criada a função parseColor() que ao ler o XML recolhe os valores da cor de cada grupo e guarda-os no atributo color da struct *Transforms* do *Group*. Para além disso, foi criado um pequeno método *transform()* que usa a *glColor3d()* que com os argumentos *red, green e blue* da cor para renderizar as primitivas com a respetiva cor.

3.4.1 Resultado

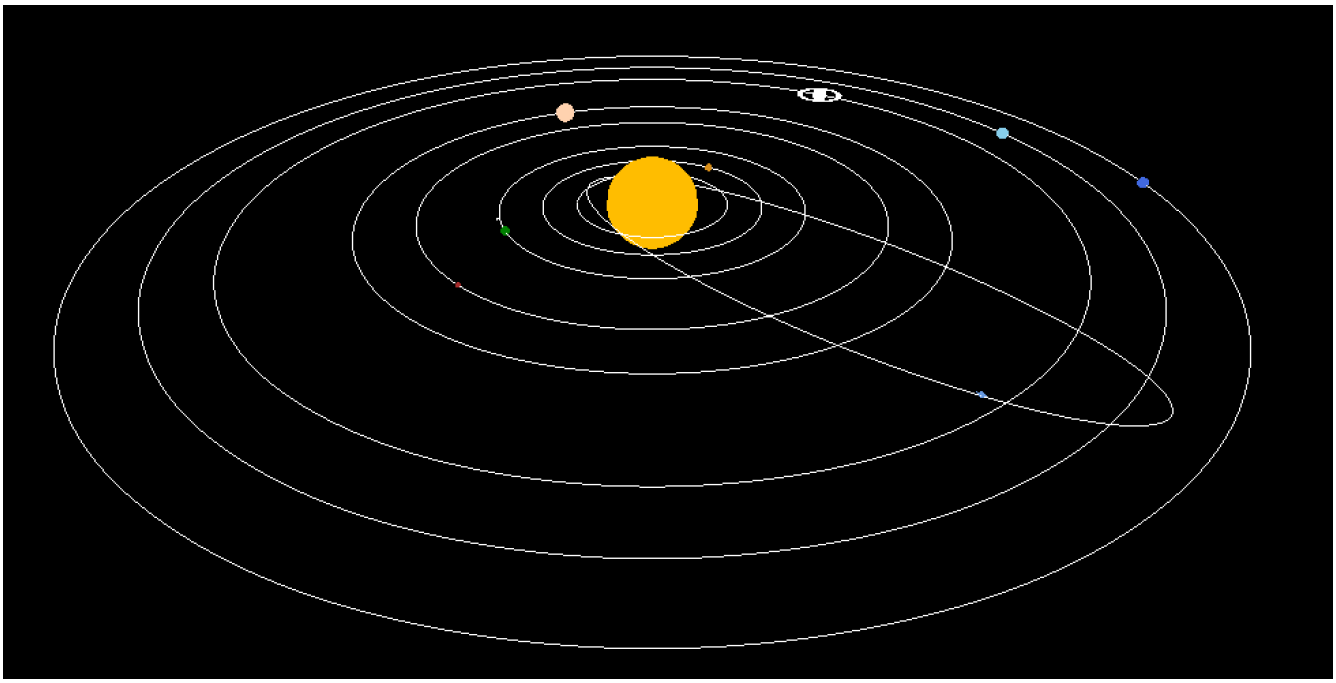

Figura 10: Sistema solar.

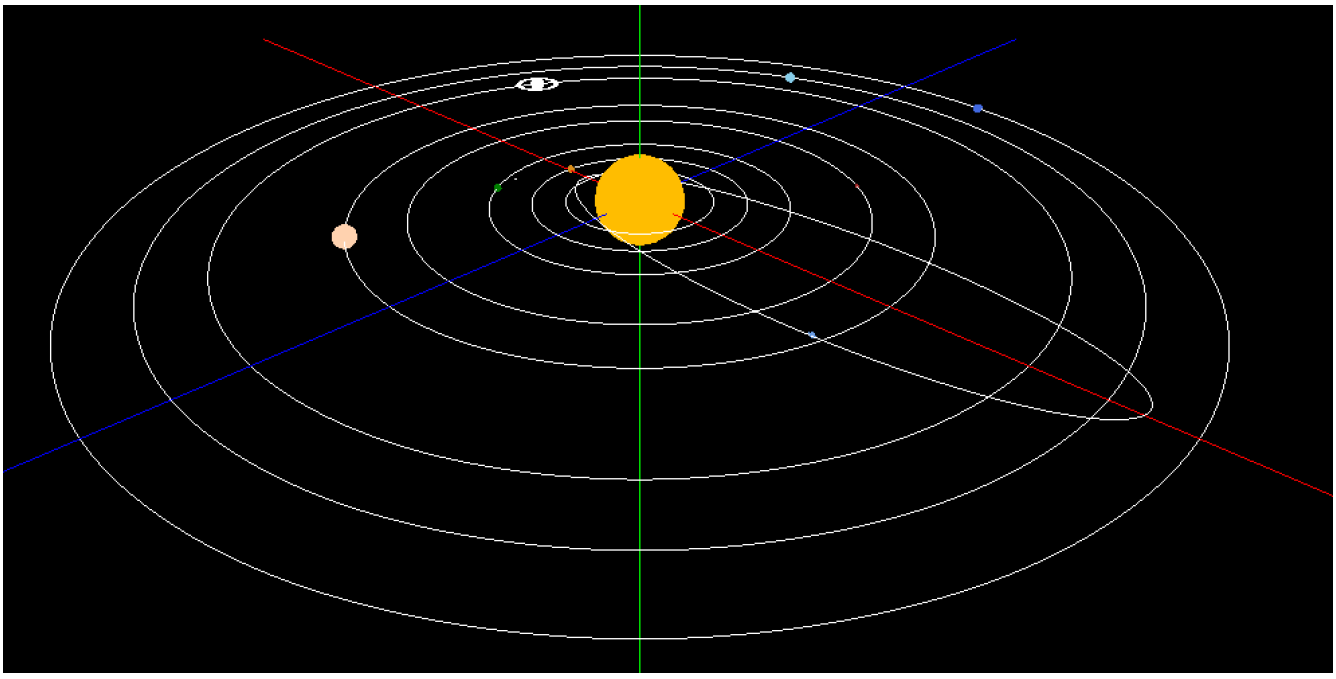

Figura 11: Sistema solar com os eixos.

4 Conclusão

De uma perspetiva geral, consideramos que a realização desta fase foi bem sucedida, visto que conseguimos cumprir com todos os requisitos estabelecidos.

No entanto, em relação à construção do ficheiro XML para a apresentação do sistema solar, enfrentamos diversas dificuldades na herança de rotações e translações entre os diferentes planetas e satélites. A junção de vários grupos e *times* também revelou-se um desafio, sendo necessário revermos o *parser* diversas vezes.

No lado positivo, destacamos o correto funcionamento do programa e a adequação das estruturas implementadas à estrutura do XML, permitindo uma visualização mais clara daquilo que é armazenado. No entanto, também enfrentamos algumas dificuldades, como a familiarização com as funções predefinidas, a implementação de funções recursivas e a implementação dos VBO's, mas todas essas dificuldades foram superadas.

Em suma, consideramos que houve um balanço positivo do trabalho realizado, pois apesar das dificuldades sentidas, conseguimos superá-las e cumprir todos os requisitos. A facilidade alcançada ao lidar com transformações geométricas, curvas e *patches*, juntamente com o *parsing* de XML, certamente facilitará a concretização da fase final do projeto.