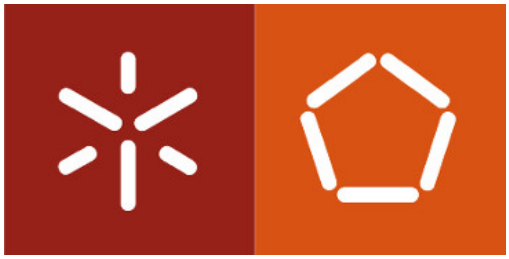

Universidade do Minho

Mestrado integrado em Engenharia Informática

Trabalho Prático - 2ª 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

4 de abril 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 segunda fase, temos o objectivo de desenvolver o *Engine* de modo a que este consiga processar translações. Com esta nova funcionalidade iremos também criar um novo ficheiro de XML para demonstração do seu efeito, o qual é inspirado no nosso sistema solar.

2 Engine

O *Engine* tem como uma das suas funções receber os ficheiros de configuração escritos em XML. Nesta fase foi preciso fazer algumas alterações de maneira a que seja possível renderizar tanto o conteúdo destes ficheiros como também as respetivas transformações associadas a estes, apresentando-as no fim em forma de *Sistema Solar*.

Na fase anterior, foram criadas as aplicações requisitadas sem ser necessário o armazenamento de informação na forma de classes. No entanto, para esta fase decidiu-se criar uma classe para cada transformação geométrica (translação, rotação e escala) que podem ser consultadas acedendo ao ficheiro **Transformação.h** na pasta *headers*.

2.1 Desenvolvimento

2.1.1 Transformações

Rotação

A rotação de uma figura geométrica implica a fixação de, pelo menos, um ponto num eixo à escolha e a esta dá-se em torno desse ponto, não havendo alteração na figura em si. Um exemplo de uma rotação no eixo Z, de um objeto com coordenadas (x,y,z) é:

```
x' = x * cos(angle) + y *sin(angle)
y' = x * sin(angle) + y *cos(angle)
z' = z
```

Com isto em mente, criou-se uma classe (*Rotacao*) que guarda toda a informação referente a uma rotação e que é composta pelas variáveis **angle, x_eixo, y_eixo, z_eixo** que identificam em qual dos eixos vai ser efetuada a rotação.

Translação

A translação desloca a figura segundo uma direção, sentido e comprimento, mais uma vez, sem alteração das suas dimensões. Se as coordenadas forem(x,y,z)

e o vetor de translação for (tx, ty, tz), iremos obter:

```
x' = x + tx
y' = y + ty
z' = z + tz
```

Sendo assim, criou-se uma classe (*Translacao*) que é composta pelas variáveis **x_eixo, y_eixo, z_eixo** que identificam em qual dos eixos vai ser efetuada a translação.

Escala

A aplicação de uma escala numa figura altera o tamanho da mesma, havendo uma multiplicação das coordenadas desta por fatores (sx,sy,sz). Uma figura que tenha como coordenadas (x,y,z), após a aplicação de uma escala passa a ter as seguintes coordenadas:

```
x' = x * sx
y' = y * sy
z' = z * sz
```

Assim, criou-se a classe *Escala* que é composta pelas variáveis **x_eixo, y_eixo, z_eixo** que identificam as dimensões dos diferentes eixos depois de realizar a escala e que permite guardar a infomração de um redimensionamento.

2.1.2 Outras classes

Finalmente, ainda se criou a classe *Cor* para permitir qualquer alteração de cor e que é composta pelas variáveis **red, green e blue** que identificam as cores segundo o modelo *RGB* e a classe *Vert* referente aos diferentes vértices necessários para construir um triângulo e composta pelas variáveis **x,y,z** que representam as suas coordenadas.

Para além destas é importante mencionar que ainda foram criadas as classes *Transformacao* que guarda um objeto do tipo **Translação, Rotação, Escala e Cor** (classes geradas anteriormente) e a classe **Transformacoes**.

A classe *Transformacoes* guarda toda a informação relativa a um subgrupo.

É constituída por uma *string tipo* que indica o tipo do ficheiro a ler (sphere.3d, por exemplo), a transformação que irá ser aplicada (**Transformacao t**) e os vértices para o desenho da figura (**vector<Vert>vert**).

O código desta classe encontra-se a seguir:

```
   class Transformacoes{
    std::string tipo;
    Transformacao t;
    std::vector<Vert> verts;

public:
    Transformacoes();
    Transformacoes(std::string tipo, Transformacao t, std::vector<Vert> vert);
    std::string getTipo(){ return tipo; }
    Transformacao getTransformacao(){ return t; }
    std::vector<Vert> getVerts(){ return verts; }
    void setTipo(std::string t){ tipo = t; }
    void setTrans(Transformacao trans){ t = trans;}
    void setVerts(std::vector<Vert> v){ verts = v;}
};
```

2.2 RenderScene

Para o desenho dos vértices em si, realizado na função *RenderScene*, uma vez que serão efetuadas transformações geométricas, implica uma alteração na matriz de transformação e por isso deve ser guardado o estado inicial desta, e depois de todas as transformações serem aplicadas, este estado deve ser reposto. Assim, usou-se as funções **glPushMatrix()** e **gelPopMatrix()** antes de aplicarmos a transformação e depois desta ser aplicada respetivamente. Para percorrermos as transformações implementamos um ciclo que percorre o vetor **transformacoes** que contém todas as transformações, e por cada iteração, fazemos um **getTransformacao()** para obtermos uma transformação.

A seguir, utilizando essa transformação, adquirimos os dados respetivos das

rotação, translação, escala e cor e utiliizamos esses valores como parâmetros nas funções *GLUT* para o processo de desenho. Para esse desenho usamos as funções **glRotatef(), glTranslatef(), glScalef() e glColor3f()** usadas nas aulas. Depois de realizadas todas as transformações, percorremos o vetor **vertexes** para procedermos ao desenho dos triângulos usando a função **glBegin(GL_TRIANGLES).**

2.3 Leitura do ficheiro XML

Nesta segunda fase do projeto, alterou-se a função *Parse* de modo a que este possa cumprir os novos requisitos pedidos. Para tal, usamos os métodos **FirstChildElement** e o **NextSiblingElement** para conseguirmos obter o primeiro *group/subggroup* do nível imediatamente abaixo e obter o próximo *group/subgroup* do mesmo nível.

Com o auxílio destes métodos, percorremos todas as transformações existentes no ficheiro e para cada uma fazemos a verificação dos seus tipos e guardamos a informação nas respetivas classes desenvolvidas mencionadas previamente.

Como o nodo mais recente (filho) tem de herdar todas as informações e caraterísticas do nodo mais antigo (filho), o passo seguinte foi fazer com que que isso fosse possível. Para alterarmos as rotações e translações somamos os nossos valores aos antigos, mas para a escala multiplicamos os novos valores pelos valores já presentes e para a cor apenas é herdada a cor do nível anterior, como podemos verificar na função *FazTransformacao()*.

Depois de tudo isto, é feito a verificação de quais os modelos que estão a ser transformados e guardamos a respetiva informação no **vetor<transformacoes> transformacoes** de modo a que essa transformação possa ser desenhada com sucesso, como explicamos anteriormente na secção do *RenderScene*.

Finalmente, verificamos se ainda falta realizar o *parse* dos níveis inferiores (filhos) ou de níveis iguais (irmãos), aplicando novamento todo este processo.

2.4 Câmara Orientada aos planetas

De forma a tornar o projeto um pouco mais atrativo e didático, foi adicionada uma *feature* extra, utilizando os números (de 0 a 9) é possível, tornar um planeta a origem e focar a câmara neste.Como exemplo temos as figuras 7 e 8 onde é utilizado o número 3 para a Terra e 6 para Saturno.

Para isto são guardadas as translações e rotações de cada planeta quando feito o *parse* do XML em dois arrays, sendo estes depois acedidos na função:

```
void keyboard(unsigned char key, int a, int b)
```

Assim quando selecionado um planeta (número), é mudada a origem para esse planeta, é mudada a posição da câmara e o seu vetor direção.

3 Generator

Nesta fase, não nos foi pedida a realização de qualquer alterações no Generator. Porém, com o intuito de criarmos modelos do Sistema Solar mais complexos e fidedignos, decidimos adicionar a opção de gerar anéis. Estes anéis irão ter 2 proveitos nos nossos modelos. O primeiro, e mais óbvio, será adicionarmos os anéis de Saturno e o segundo será adicionar orbitas "protótipo"aos planetas.

3.1 pRing

Podemos considerar o Ring ou o Anel como um plano definido entre 2 circunferências. Esse plano vai estar dividido em várias *slices* que se assemelham a um trapézio definido pela união de dois triângulos. Para criarmos o primeiro triângulo, começamos por seleccionar 2 pontos de circunferência exterior e 1 da interior (*anti-clockwise*) e para o segundo fazemos o oposto (mas mantemos a ordem *anti-clockwise*). Fazemos isto, através do seguinte excerto de código:

```
fprintf(f, "%f %f %f \n", sin(i)*radius_int, 0.0, cos(i)*radius_int);
fprintf(f, "%f %f %f \n", sin(i)*radius_out, 0.0, cos(i)*radius_out);
fprintf(f, "%f %f %f \n", sin(i+ang_slice)*radius_int, 0.0, cos(i+ang_slice)*radius_int);

fprintf(f, "%f %f %f \n", sin(i)*radius_out, 0.0, cos(i)*radius_out);
fprintf(f, "%f %f %f \n", sin(i+ang_slice)*radius_out, 0.0, cos(i+ang_slice)*radius_out);
fprintf(f, "%f %f %f \n", sin(i+ang_slice)*radius_int, 0.0, cos(i+ang_slice)*radius_int);
```

Figura 1: Excerto de código que define a face superior do anel

O plano inferior é feito da mesma forma, mas desta vez invertemos a ordem dos triângulos para que o plano fique virado no sentido oposto.

```
fprintf(f, "%f %f %f \n", sin(i)*radius_out, 0.0, cos(i)*radius_out);
fprintf(f, "%f %f %f \n", sin(i)*radius_int, 0.0, cos(i)*radius_int);
fprintf(f, "%f %f %f \n", sin(i+ang_slice)*radius_int, 0.0, cos(i+ang_slice)*radius_int);

fprintf(f, "%f %f %f \n", sin(i+ang_slice)*radius_out, 0.0, cos(i+ang_slice)*radius_out);
fprintf(f, "%f %f %f \n", sin(i)*radius_out, 0.0, cos(i)*radius_out);
fprintf(f, "%f %f %f \n", sin(i+ang_slice)*radius_int, 0.0, cos(i+ang_slice)*radius_int);
```

Figura 2: Excerto de código que define a face inferior do anel

Num plano mais geral, utilizamos o seguinte excerto de código para realizar a criação do modelo por inteiro. Este excerto já representa o mecanismo que nos permite adaptar os valores das *slices* aos ângulos das circunferências.

```
float ang_slice = (360*M_PI)/(slices*180);

for(float i = 0; i < 2*M_PI; i += ang_slice){

    fprintf(f, "%f %f %f \n", sin(i)*radius_int, 0.0, cos(i)*radius_int);
    fprintf(f, "%f %f %f \n", sin(i)*radius_out, 0.0, cos(i)*radius_out);
    fprintf(f, "%f %f %f \n", sin(i+ang_slice)*radius_int, 0.0, cos(i+ang_slice)*radius_int);

    fprintf(f, "%f %f %f \n", sin(i)*radius_out, 0.0, cos(i)*radius_out);
    fprintf(f, "%f %f %f \n", sin(i+ang_slice)*radius_out, 0.0, cos(i+ang_slice)*radius_out);
    fprintf(f, "%f %f %f \n", sin(i+ang_slice)*radius_int, 0.0, cos(i+ang_slice)*radius_int);

    fprintf(f, "%f %f %f \n", sin(i)*radius_out, 0.0, cos(i)*radius_out);
    fprintf(f, "%f %f %f \n", sin(i)*radius_int, 0.0, cos(i)*radius_int);
    fprintf(f, "%f %f %f \n", sin(i+ang_slice)*radius_int, 0.0, cos(i+ang_slice)*radius_int);

    fprintf(f, "%f %f %f \n", sin(i+ang_slice)*radius_out, 0.0, cos(i+ang_slice)*radius_out);
    fprintf(f, "%f %f %f \n", sin(i)*radius_out, 0.0, cos(i)*radius_out);
    fprintf(f, "%f %f %f \n", sin(i+ang_slice)*radius_int, 0.0, cos(i+ang_slice)*radius_int);

}
```

Figura 3: Excerto de código que define o anel

4 Demonstração

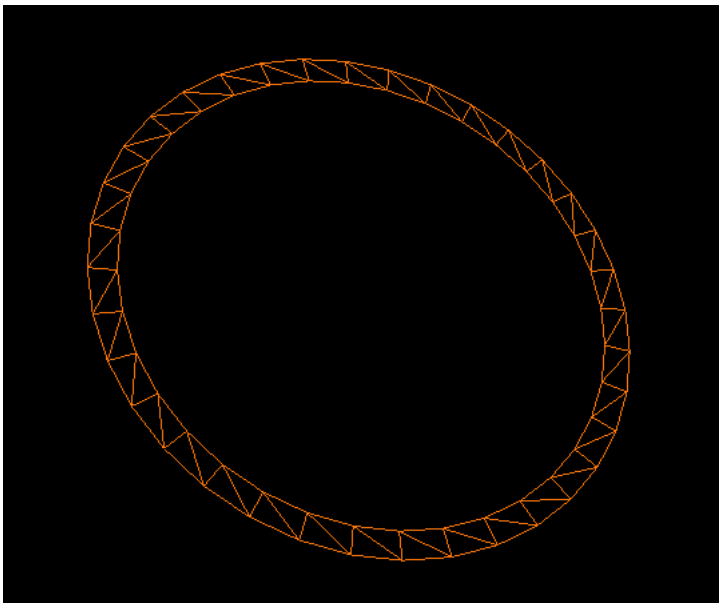

Figura 4: Demonstração de um modelo Anel

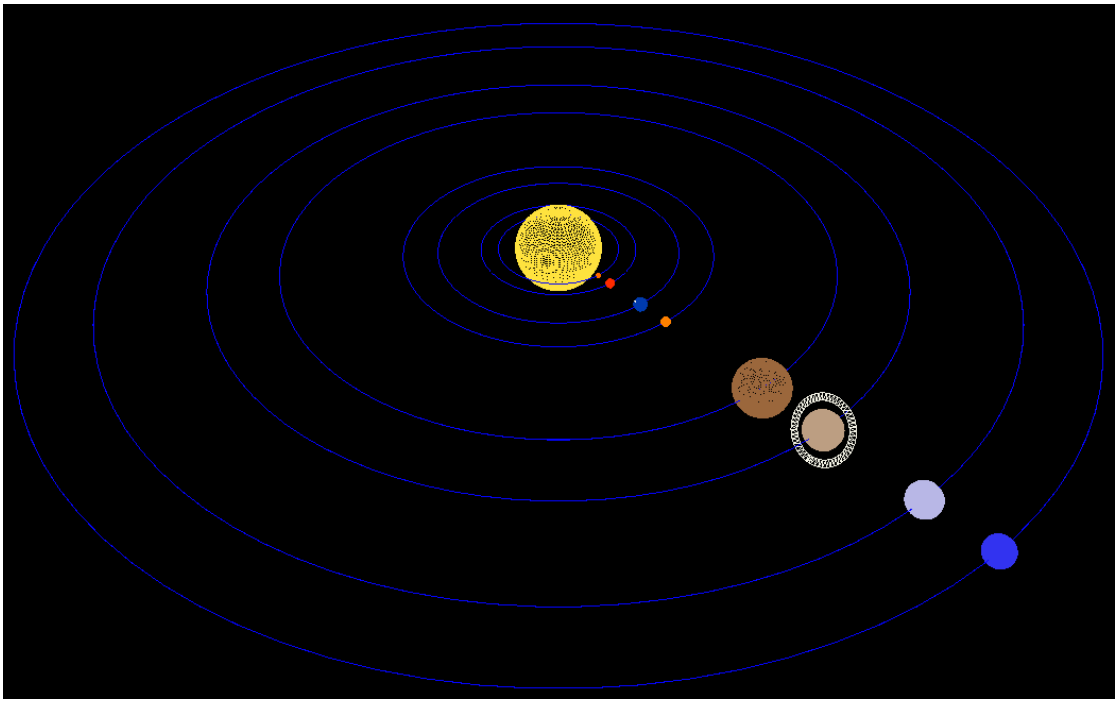

Figura 5: Demonstração do modelo SolarSystem4

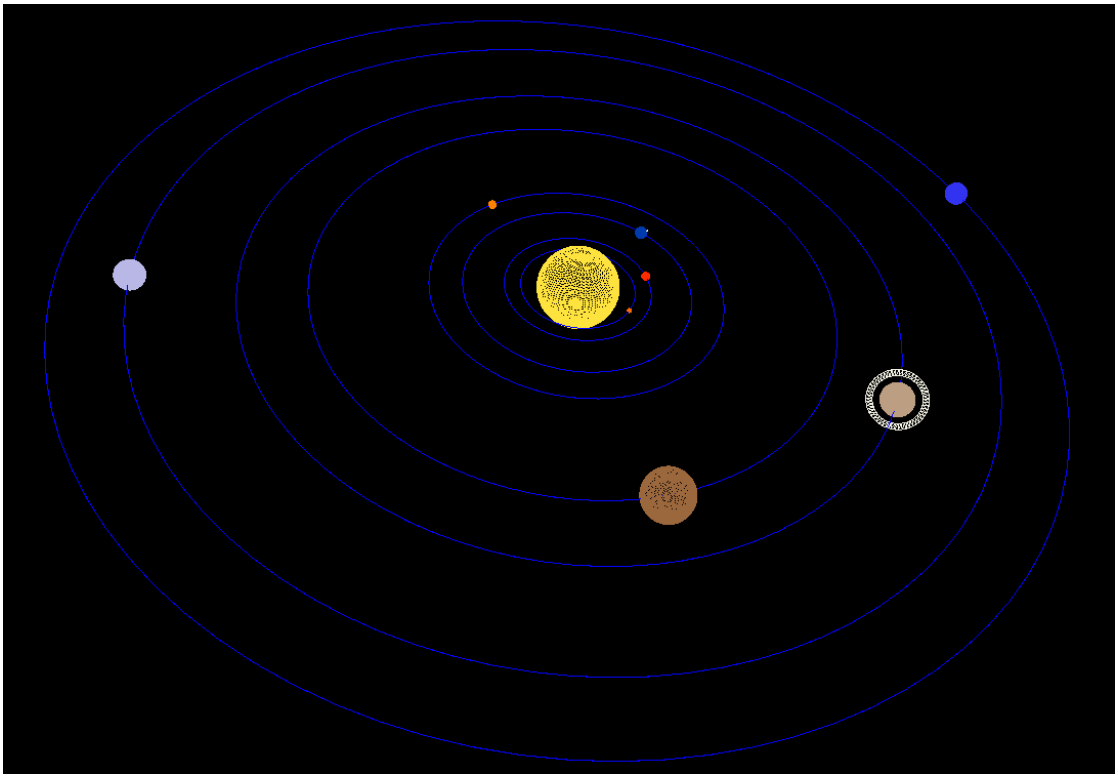

Figura 6: Demonstração do modelo SolarSystem5

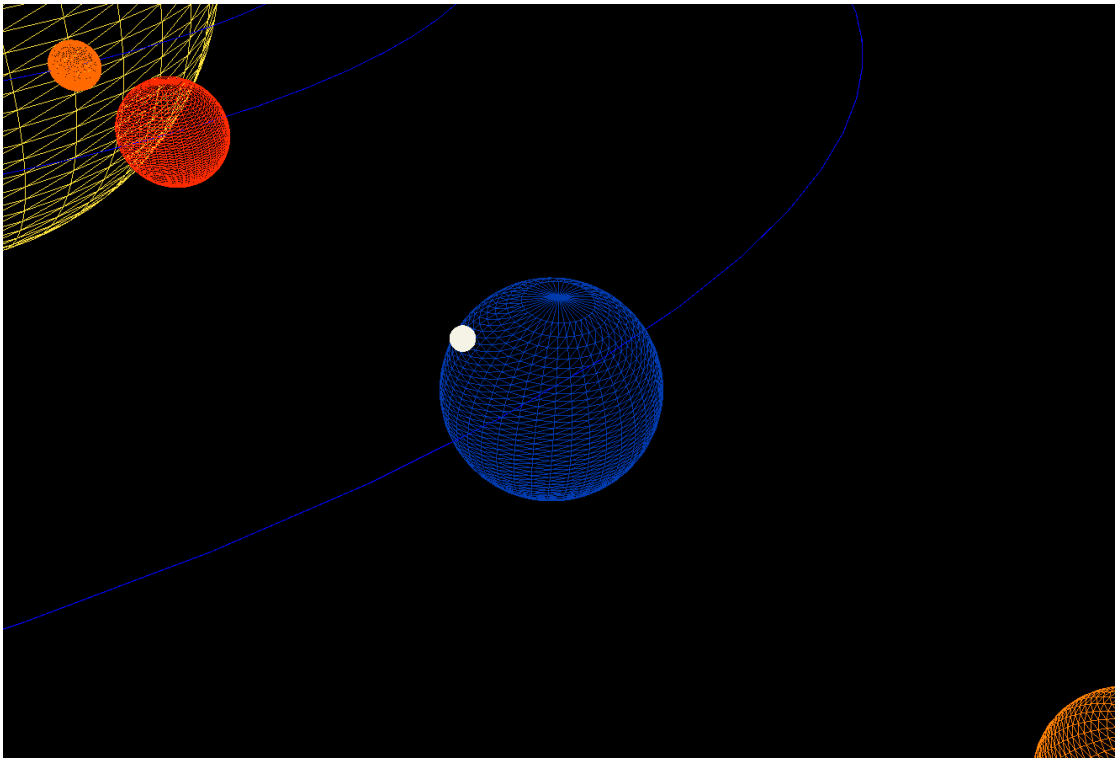

Figura 7: Demonstração do modelo da Terra

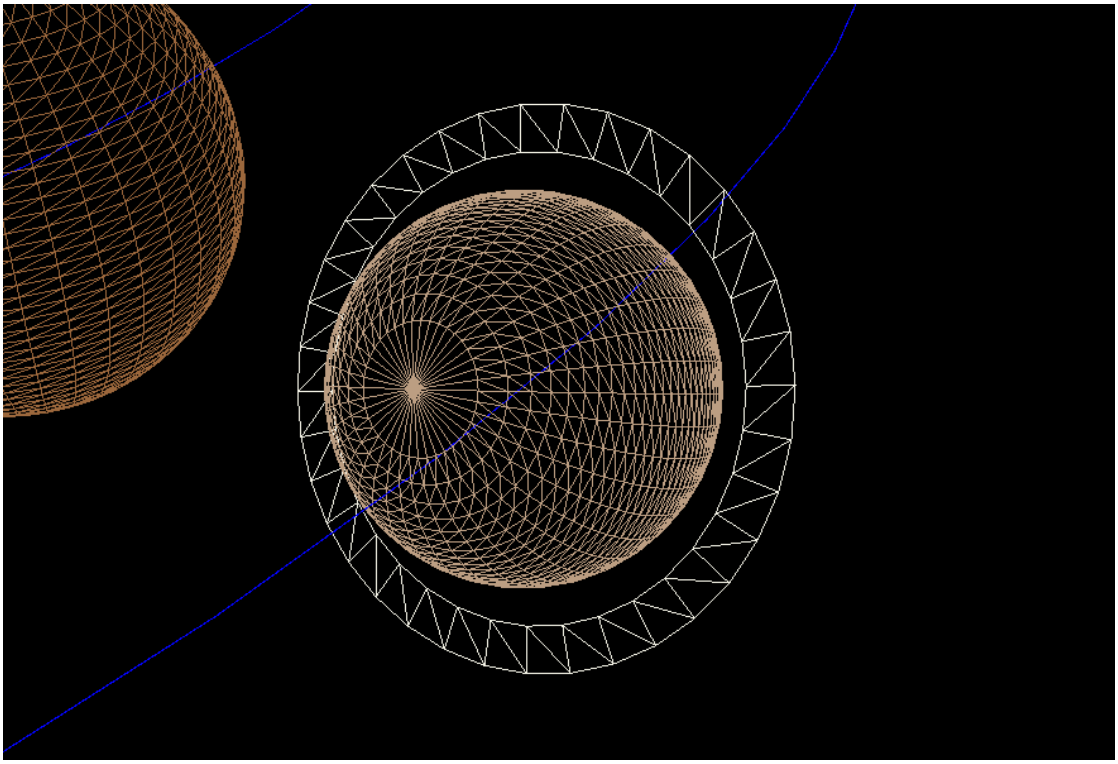

Figura 8: Demonstração do modelo de Saturno

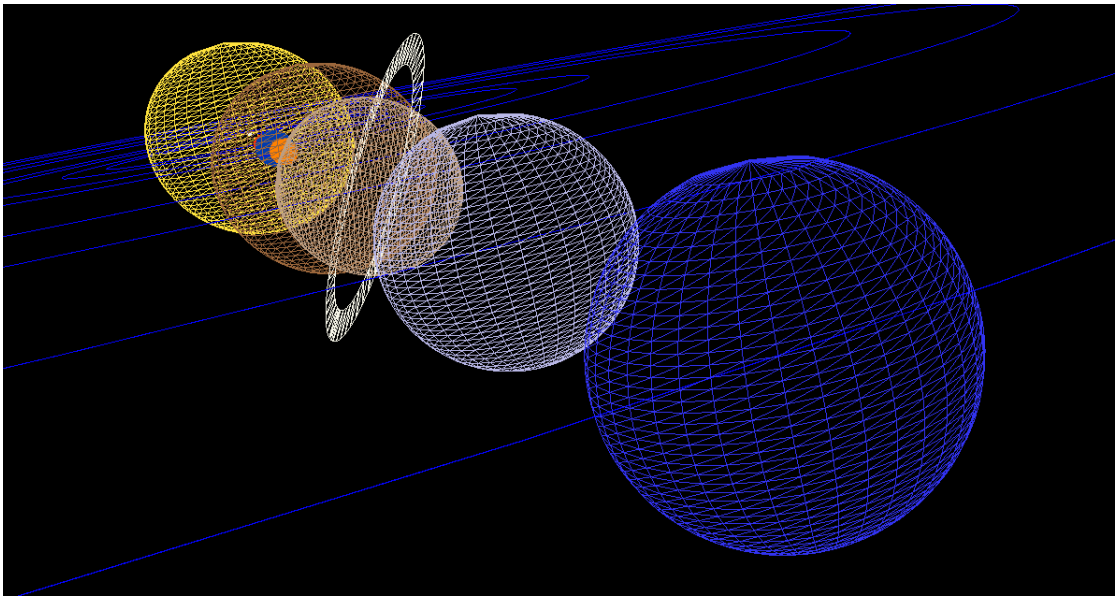

Figura 9: Demonstração da Sequência dos Planetas

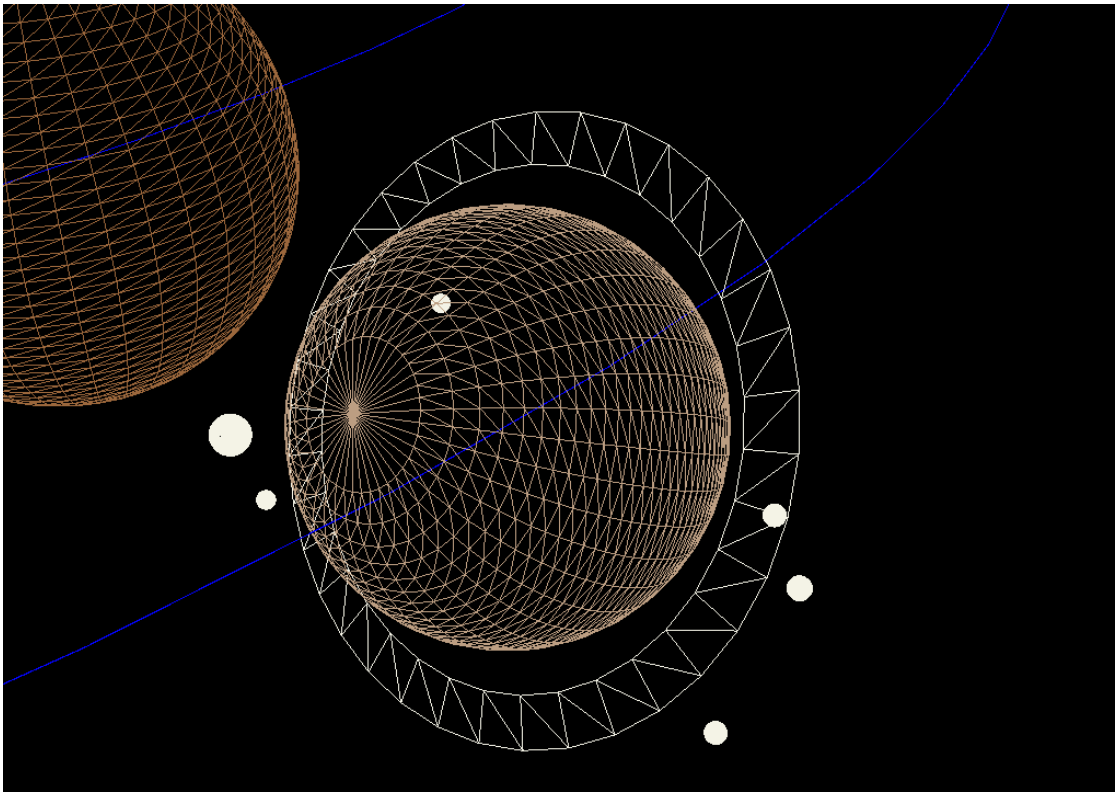

Figura 10: Demonstração do modelo atualizado de Saturno

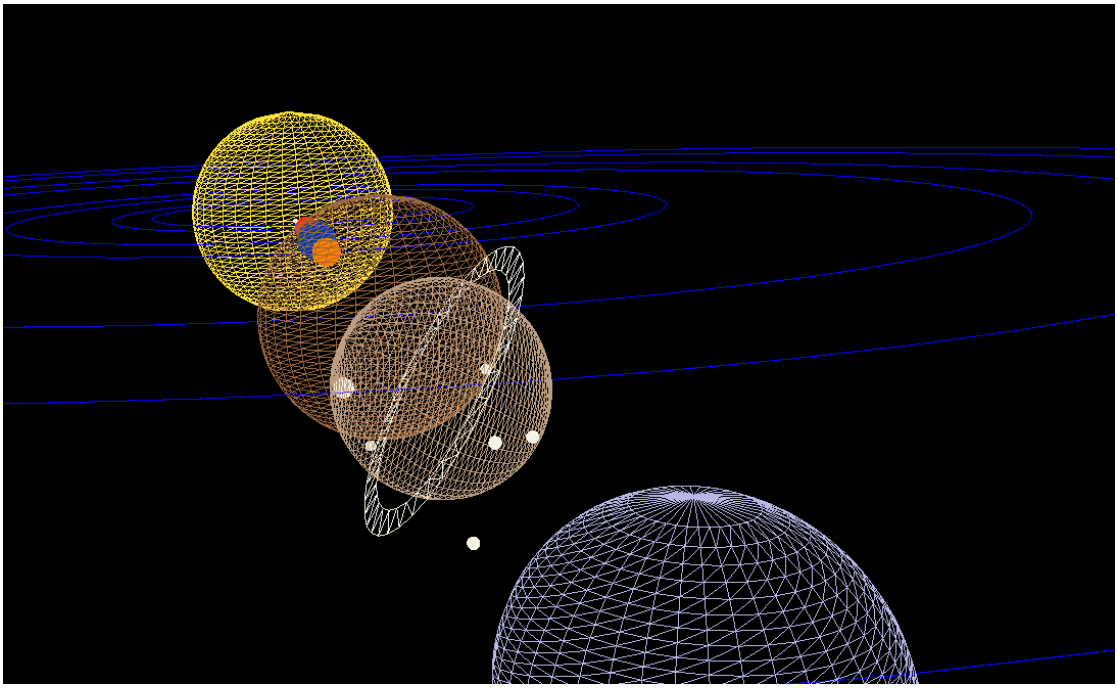

Figura 11: Demonstração da Sequência atualizada dos Planetas

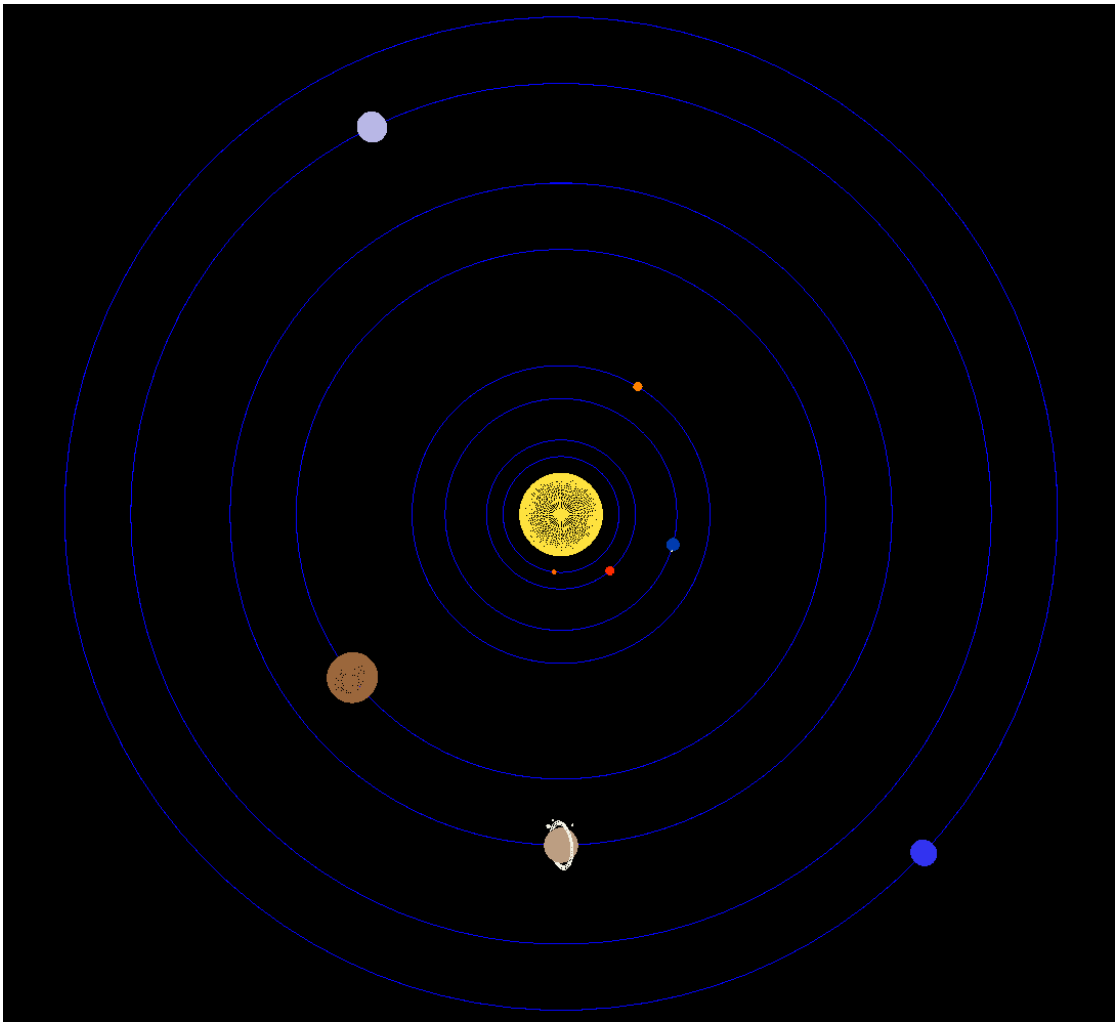

Figura 12: Demonstração do modelo SolarSystem5 atualizado

5 Conclusão

Nesta segunda fase do projecto, foi-nos pedido para realizar uma atualização ao *Engine* por forma a torná-lo capaz de realizar um conjunto de transformações geométricas. Depois, desenvolver também a nossa função de *parsing* para ser capaz de ler ficheiros XML com hierarquias às quais serão aplicadas as transformações.

Para sermos capazes de aplicar e analisar os valores que o nosso programa fornece. Criamos um modelo do Sistema Solar e para o tornar mais complexo actualizamos o *Generator* para capacitar a criação de anéis.

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.