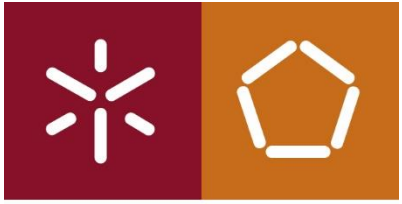

Universidade do Minho
Escola de Engenharia

Computação Gráfica

Relatório do projeto prático

Fase II

Grupo de Trabalho

Ana Esmeralda Fernandes

A74321

Bárbara Nadine Oliveira

A75614

João Paulo Ribeiro Alves

A73542

Miguel Dias Miranda

A74726

Mestrado Integrado em Engenharia Informática

março de 17

Conteúdo

1. Introdução

Este relatório visa apresentar as decisões tomadas na realização da segunda fase do trabalho prático da Unidade Curricular de Computação Gráfica. Procuramos assim justificar todas as ponderações feitas na formulação do problema, assim como a abordagem tomada para gerar, a partir de um ficheiro de input em *xml* com elementos pré-definidos, uma maquete de um sistema solar de forma simples, fácil de interagir e fazendo um bom uso das funções de glPushMatrix e glPopMatrix da matriz de visualização do modelo (modelview Matrix) do *glut*.

Toda a modulação do problema foi feita com recurso à ferramenta de programação Visual Studio e à linguagem C++, abordada nas aulas práticas da UC. Numa primeira fase será descrito o processo de leitura e tratamento dos dados contidos no ficheiro *xml* e posteriormente é explicado o modo como desenvolvemos as estruturas de dados que suportam o programa e o modo como cada elemento é renderizado na função *renderscene*.

2. Projetos Desenvolvidos

2.1. Aplicação sistemaSolar

O projeto responsável por interpretar as informações contidas num ficheiro xml, foi designada como sistemaSolar. Apesar do seu nome ser bastante especifico, devido ao contexto das figuras que serão desenhadas, a aplicação será capaz de traçar qualquer tipo de cenário desde que o mesmo esteja representado de forma correta, segundo os parâmetros e hierarquias que compõe o ficheiro xml de input (este aspeto será descrito na seguinte secção do relatório).

Deste modo, a valorização do desenvolvimento desta fase não se encontra no rigor com que, neste caso, o sistema solar é desenhado, mas sim no modo como foi abordada a questão de interpretar o ficheiro de input e como foram guardadas as informações eventualmente contidas nele.

O módulo designado de sistemaSolar.cpp será irá conter a função main, assim como as funções responsáveis por lidar com a interpretação do xml de entrada e as funções da biblioteca *glut*, usadas de forma semelhante à abordagem tida nas aulas da disciplina.

Inicialmente a aplicação começa por fazer a análise e leitura do ficheiro xml cujo nome é passado como argumento, criando e compondo uma estrutura de dados global (esta será descrita seguidamente) que irá conter as informações sobre os grupos do ficheiro xml. Concluída esta fase inicial, feita apenas uma vez, a estrutura de dados será usada pela função renderscene da biblioteca glut, para ser iterada e ser extraída a informação de modo a desenhar de forma correta o cenário descrito no ficheiro de input, com recurso ao uso das funções glPushMatrix e glPopMatrix e sem "desperdício" de transformações já realizadas, como exemplo tivemos em consideração um ficheiro de input que contém uma maquete do sistema solar.

2.1.1. Descrição das estruturas de dados utilizadas

Para dar resposta à necessidade de guardar diferentes tipos de informações, lidas a partir do ficheiro xml e posteriormente usadas para desenhar corretamente a maquete do sistema solar, foram construídas as seguintes estruturas de dados, que podem ser descritas brevemente da seguinte forma:

- Classes Escala, Translacao, Rotacao, Orbita e Cor: classes implementadas para guardar informações sobre escalas, translações, rotações, órbita e cores, respetivamente. Estas classes serão usadas para inicialmente guardar informações lidas do ficheiro xml e, posteriormente, usadas para fornecer os valores de cada uma das transformações que representam às funções *glut* que as executam, nomeadamente, *glScalef, glTranslatef, glRotatef e glColorf*.

- Classes Vertice e Modelo: classes já utilizadas no projeto da Fase I. A classe Modelo representa uma figura, neste caso um astro do sistema solar, que é composto pelo conjunto de vértices que formam os seus triângulos. Por estarmos a desenvolver uma animação a três dimensões, um Vertice será um ponto no espaço, representando pelas três coordenadas cartesianas x,y e z.

Para esta segunda fase do projeto, a classe Modelo foi reajustada para se adequar à necessidade do problema. Sendo inicialmente usada para gerar modelos (como esferas ou cubos) passa a ser usada apenas para guardar os vértices que compõe um determinado modelo a desenhar.

- Classe Parsing: Módulo responsável por agrupar as funções responsáveis por lidar com ficheiros, quer seja para interpretar o ficheiro xml de input quer para ler as informações contidas nos ficheiros com os vértices que representam os modelos a representar.
  Será a partir deste módulo que se gera a estrutura de dados (variável do tipo *Parsing**) que será a variável global do módulo principal com a função main. Esta variável contem em si uma variável privada do tipo *Grupo** que será um apontador para o grupo inicial (raiz da árvore) a desenhar.
- Classe Grupo: Classe desenvolvida pela necessidade de guardar informações sobre os grupos a desenhar. Deste modo, a classe Grupo terá como variáveis um conjunto de informações relativas a transformações (uma escala, uma translação, uma rotação, uma órbita e uma cor), aos modelos a desenhar naquele grupo e um apontador para o Grupo irmão e Grupo filho, caso existam.
  Será posteriormente desenvolvida a forma como as estruturas de dados desta classe permitem hierarquizar as cenas tal como inicialmente estavam relacionadas no ficheiro de input.
  Deste modo cada Grupo acaba por ser uma árvore binária com os ramos irmão e filho, uma vez que todos os grupos são obtidos a partir de um Grupo pai ou de um Grupo irmão esta estrutura é suficiente para conseguirmos representar toda a informação pretendida.

Todas estas classes foram desenvolvidas segundo uma estrutura modelar, estando separadas entre o ficheiro com o código em extensão .cpp e o ficheiro com os cabeçalhos com a extensão .h.

A seguinte imagem procura descrever o diagrama de classes da aplicação usada.

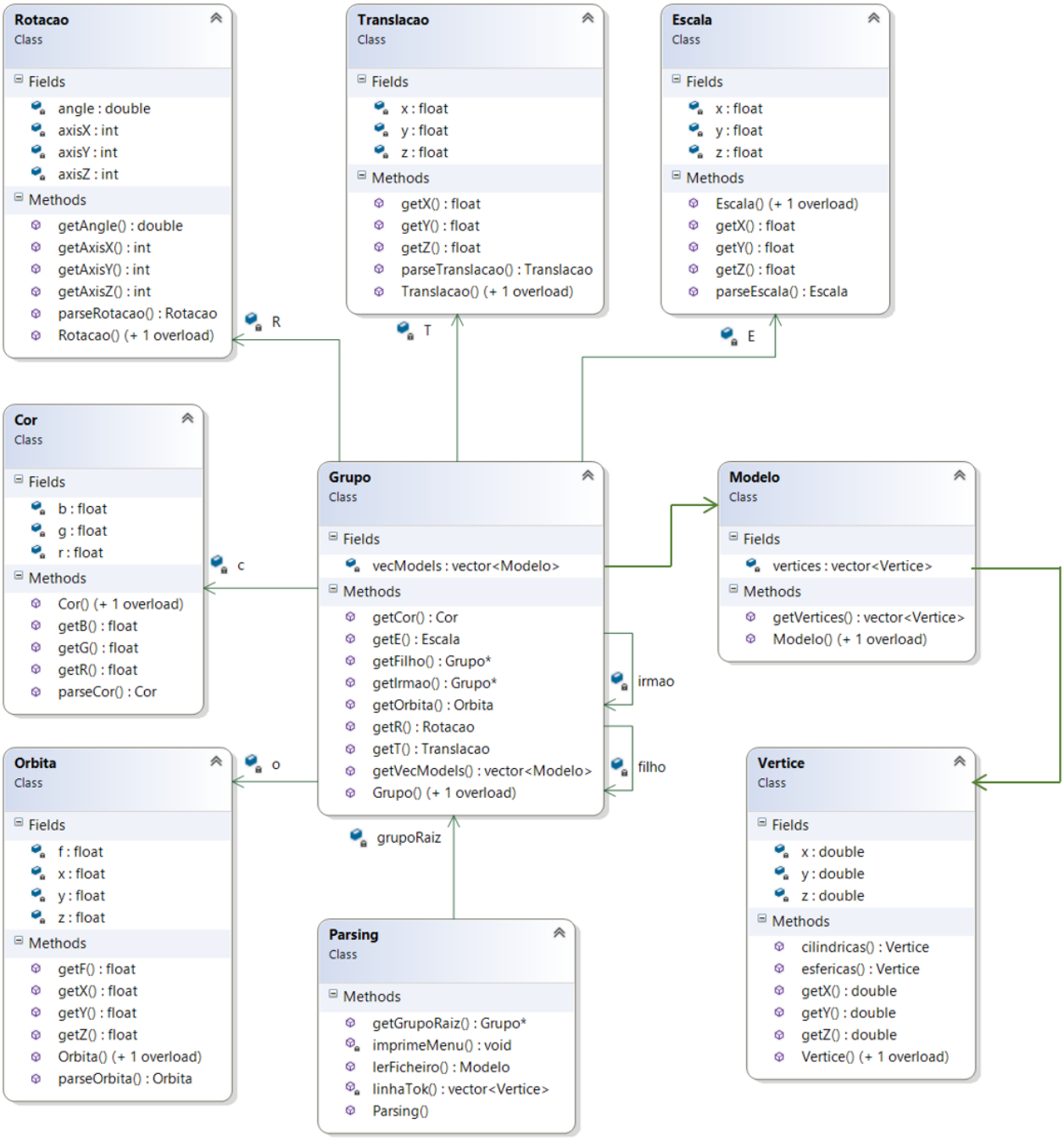

Figura 1 – Diagrama de Classes do Projeto sistemaSolar

2.1.2. Descrição do processo de leitura *xml*

Para realizar a análise do input em formato *xml* foi reutilizada a biblioteca *TinyXML* já utilizada e familiarizada na Fase I. Para conseguir lidar com a estrutura própria deste ficheiro e a hierarquia dos grupos que possam existir nele, foram tomadas as seguintes considerações para a abordagem ao problema da leitura do ficheiro:

1. Cada elemento grupo, será filho do elemento scene do ficheiro xml e poderá ou não conter dentro de si outros elementos grupo.
   Um grupo será composto por elementos que representam uma translação, uma escala, uma rotação, um conjunto de modelos, uma órbita e uma cor.
   Estes campos do xml são opcionais e, no caso de não serem especificados, será feita a respetiva transformação genérica/neutra. Por exemplo, se não for especificado nenhum tipo de rotação, é feita uma rotação para a posição (0,0,0) ou, no caso de uma escala, se esta não for indicada é feita uma escala com as proporções x=y=z=1.
   Para determinar se um determinado elemento em xml dentro do campo grupo existe, é testado se o apontador da biblioteca *tinyXML* para esse elemento se encontra ou não a *null*. Por uma questão de coerência contextual, este teste ao apontador é feito nas funções de parsing do XML, que existem nas classes da respetiva transformação. Por exemplo, a função de parsing de um elemento xml do tipo *escala* será implementada no respetivo módulo Escala.cpp.
2. Para guardar as informações que existem em cada elemento grupo do xml, foi criada a classe designada também por Grupo, descrita em pormenor no capitulo anterior referente às estruturas de dados utilizadas.
   De uma forma geral, esta classe terá então uma variável para cada tipo de transformação, sendo cada uma delas usada para guardar a informação associada com a correspondente transformação existente no xml, enunciadas no ponto (1).
   Para o caso do ficheiro xml que nos propomos a analisar, cada elemento grupo terá que conter a referência obrigatória a pelo menos um modelo/ficheiro onde estarão os vértices da figura a desenhar com as transformações desse grupo. Para o contexto especifico do nosso sistema solar, temos apenas um modelo por grupo apesar da aplicação conseguir lidar com vários modelos dentro do mesmo grupo.
3. A interpretação do conteúdo do ficheiro xml é realizada de forma recursiva, pelo construtor da classe `Grupo::Grupo(TiXmlElement* grupo)`.
   Ao realizar o tratamento do elemento xml recebido como argumento, são preenchidas as variáveis referentes às transformações da classe Grupo.
   Posteriormente, se existir no elemento xml atual um grupo filho, a variável da classe Grupo que aponta para o filho (Grupo* this->filho) fica com o Grupo construído pela chamada recursiva do construtor `Grupo::Grupo(TiXmlElement* grupo)`, que recebe como argumento o apontador do xml filho. Se não existir um elemento grupo filho no xml, esta variável this->filho ficará com o valor nullptr.
   Caso exista no elemento xml atual um grupo irmão, a variável da classe Grupo que aponta para o irmão (Grupo* this->irmao) fica com o Grupo construído pela chamada recursiva do construtor `Grupo::Grupo(TiXmlElement* grupo)` com o apontador do xml irmão. Se não existir um elemento grupo irmão no xml, a variável this->irmao ficará com o valor nullptr atribuído.

O seguinte excerto de código implementa a ideia descrita:

```
Grupo::Grupo(TiXmlElement* rootGrupo) {

	// Construção e atribuição das variáveis da classe Grupo
	// referentes às transformações existentes no campo grupo do xml.

	TiXmlElement* aux = rootGrupo->FirstChildElement("group");
	if (aux) {
		this->filho = new Grupo(aux);
		}
	else {
		this->filho = nullptr;
	}
	aux = rootGrupo->NextSiblingElement("group");
	if (aux) {
		this->irmao = new Grupo(aux);
	}
	else {
		this->irmao = nullptr;
	}
}
```

4. Será a classe Parsing a ter na sua variável Grupo* grupoRaiz o apontador para o primeiro grupo a desenhar. Será para esta variável que se devolve o valor retornado pelo construtor da classe Grupo descrito no ponto (3) anterior.
   Terminadas todas as invocações recursivas deste construtor da classe Grupo, são então vistos todos os elementos grupo do ficheiro xml de input, sejam estes filhos ou irmãos entre si.
   Na variável grupoRaiz fica então o apontador para a o primeiro grupo a desenhar e, nas variáveis grupoRaiz->filho e grupoRaiz->irmao os apontadores para os filhos ou irmãos deste grupo, caso existam.
5. A leitura do ficheiro xml é realizada apenas uma vez, logo na fase inicial de compilação do projeto, ficando assim na classe principal sistemaSolar a variável, já referida, do tipo Parsing* com toda a informação sobre os grupos a desenhar, tal como descrito no anterior ponto (4).

2.1.3. Descrição do ciclo de rendering

2.1.3.1. Formulações iniciais

Para iniciar esta secção vamos enunciar e explicitar alguns dos conceitos que vão ser mencionados nesta descrição:

- Matriz de visualização do modelo (*modelview Matrix*):

  A matriz de visualização do modelo é a concatenação da matriz do modelo com a matriz de visualização. A matriz do modelo define a posição da frame em que as primitivas vão ser desenhadas, enquanto que a matriz de visualização define a localização da câmara.

  Quando utilizamos transformações como glRotate, glTranslate ou glScale, a matriz de visualização será alterada de modo a permitir que as transformações desejadas sejam executadas. Relembramos que em OpenGl quando executamos transformações, nós posicionamos o mundo em relação à câmara que permite a visualização, e não o inverso, como representa a imagem seguinte em que para aumentarmos o "zoom" da imagem é esta que é aproximada da câmara.

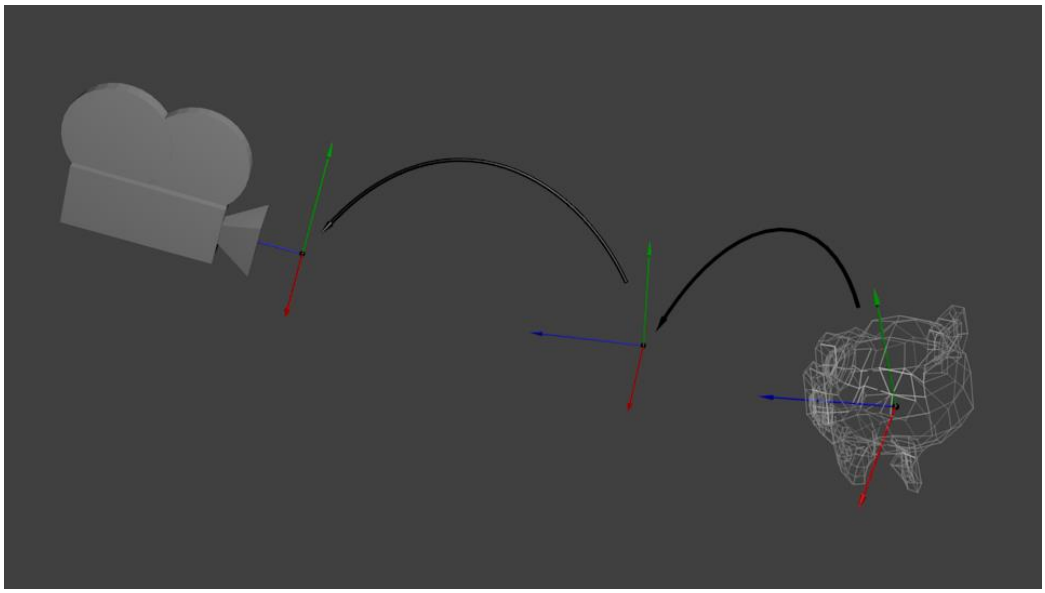

Figura 2 - - Transformações de visualização em OpenGL

- *glPushMatrix();* & *glPopMatrix();*

  Para conseguir preservar um estado da matriz de visualização de um determinado modelo, é utilizada a função *glPushMatrix()* da biblioteca glut, que irá guardar a matriz atual numa stack.

  Quando for pretendido recuperar a última matriz guardada na referida stack, isto é, quando desejarmos que o estado da matriz de visualização do modelo volte a ser o que era aquando do último *glPushMatrix()* executado, basta para isso realizar a chamada da função *glPopMatrix()*.

Na imagem seguinte pretendemos ilustrar de modo simples e genérico este funcionamento das funções de glPushMatrix e glPopMatrix:

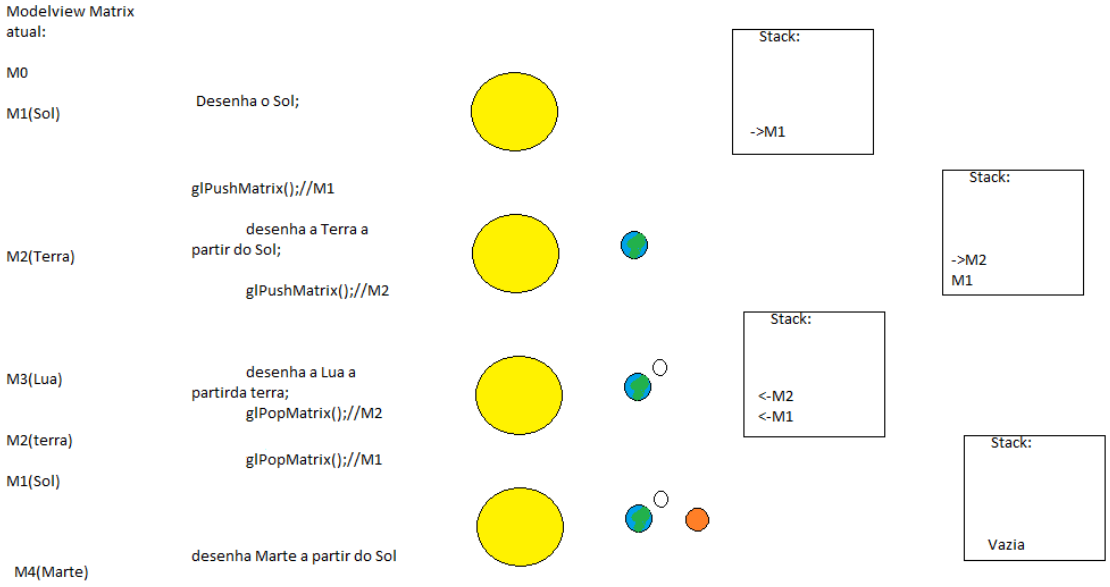

Figura 3- Ilustração do funcionamento da stack da matriz de visualização do modelo

2.1.3.2. Descrição da função renderscene

Tendo em conta os conceitos mencionados será descrito do processo utilizado para a execução do ciclo de rendering.

Durante a execução inicial do programa, após a leitura e tratamento do ficheiro xml de input terem sido realizadas, ficará guardada a informação nele contida em memória, numa variável global que será um apontador para uma estrutura do tipo Parsing. Será então na função renderscene, com a invocação da função *desenhaGrupo(),* que se realiza a iteração desta variável global e obtém a informação necessária, de modo a conseguirmos desenhar o cenário pretendido.

A implementação da referida função *desenhaGrupo* irá percorrer a estrutura de dados de forma idêntica ao modo com que esta foi preenchida, para ser possível obter as transformações a ser efetuadas à matriz de visualização atual do modelo (modelview Matrix) de cada um dos grupos "filhos". Este processo de iteração tem em consideração as transformações já realizadas pelo respetivo Grupo pai e permite assim ser possível reverter a modelViewMatrix para o ponto anterior, depois de todos os "filhos" de um dado grupo terem sido desenhados.

2.1.3.3. Descrição da função desenhaGrupo

Esta função *desenhaGrupo* recebe como argumento o apontador para um elemento da classe grupo, que representa o nodo inicial da estrutura de dados e iterar. O seu funcionamento pode ser descrito pelos seguintes passos:

- Antes de realizar qualquer alteração à matriz de visualização do modelo é efetuado um *glPushMatrix()* de modo a guardar o estado da matriz atual para que possam ser recuperadas posteriormente.
- Num segundo passo, é realizada a extração da informação necessária no nodo que recebe como argumento, e realiza as transformações correspondentes. Estas transformações são uma rotação, uma translação e uma escala que vão ser aplicadas à matriz atual, utilizando as funções glut correspondente. É define qual a cor a utilizar.
- Realizadas as transformações, utilizando a informação contida no nodo, na variável *vector<Modelo>*, será feito um ciclo para cada um dos modelos encontrados em com recurso à primitiva *glBegin(GL_TRIANGLES);* vão ser desenhado todos os vértices correspondentes a cada um dos modelos. Como já referido, no nosso contexto cada Grupo só terá um modelo, embora seja possível ter vários.
- Após ter sido realizado o desenho dos triângulos, a função tenta identificar se o grupo atual tem algum nodo/grupo filho. Para que não se percam as transformações já realizadas, caso se verifique que o nodo atual tenha descendência, a função desenhaGrupo é chamada recursivamente para o filho, sem que o *glPopMatriz()* seja efetuado.
  Caso o grupo atual não tenha nenhum grupo filho, é efetuado o *glPopMatrix(),* uma vez que vamos querer voltar ao estado da matriz de visualização do modelo anterior, sem as alterações que foram efetuadas pela atual iteração da função desenhaGrupo. Este processo é efetuado porque as transformações dos grupos seguintes (irmãos) já não serão sobrepostas às transformações atuais.
- Posteriormente, e de forma análoga ao tópico anterior, a função tenta identificar se o grupo atual tem algum irmão, e será invocada recursivamente para este, caso exista.

A natureza recursiva da função e o modo como se encontra definida garante que as transformações à matriz de visualização do modelo para os filhos de um dado grupo são sempre obtidas a partir de transformações já efetuadas para o seu respetivo grupo pai, pelo razão de não existir nenhum *glPopMatrix()* entre as transformações realizadas para o pai, e a invocação recursiva da função *desenhaGroup* para o filho, invocação esta que irá efetuar novas transformações a partir das anteriores.

Por motivos idênticos, todos os irmãos de um dado grupo serão desenhados a partir das mesmas transformações, ou seja, a partir da matriz de visualização do modelo que foi obtida a partir do grupo pai de todos eles, isto porque, quando a função é invocada para um grupo a partir do seu irmão, é feito um *glPopMatrix()* que faz a matriz reverter para o estado anterior. Este processo anula as transformações executadas para o grupo atual e assim a matriz de visualização do modelo irá ser a mesma que foi obtida a partir das transformações executadas pelo pai do grupo atual.

Na função renderScene invocamos a função desenhaGrupo passando-lhe a "raíz" da variável global que corresponde ao primeiro grupo a ser desenhado e a partir daí, a função irá processar de forma semelhante ao processo descrito acima.

O seguinte código mockup descreve de forma sucinta este processo:

```
DesenhaGrupo(grupo){
      glPushMatrix();
   ➢ obtém transformações guardadas no grupo;
   ➢ obtém informações (cor e vértices a serem desenhados)
      guardadas nos modelos do grupo;
   ➢ efetuar transformações;
   ➢ desenhar vértices dos modelos;

      se (Grupo atual tiver filho) =>
                  desenhaGroup (filho do Grupo atual);
      glPopMatrix();
      se (Grupo atual tiver irmão) =>
                  desenhaGroup (irmão do Grupo atual);
   ➢ termina a função;
}

renderScene(){
      (…)
      raiz = getRaizGrupos();
      /* grupo* raiz -> raiz da variável global, que contem
a informação obtida pelo parsing do ficheiro de input xml)
*/
      DesenhaGrupo(raiz);
      (…)
}
```

3. Funcionalidades adicionais no projeto

Para além dos pontos descritos no enunciado como requisitos básicos, foi ainda realizada a implementação de algumas funcionalidades extra que julgamos valorizar o trabalho desenvolvido.

3.1. Cor dos planetas

Para atribuir uma cor a cada um dos objetos desenhados no sistema solar foi adicionada a possibilidade de inserir o elemento Cor dentro de um elemento Grupo do ficheiro xml de entrada.

Assim, caso tenha sido atribuído o elemento Cor a um determinado grupo do xml, este irá conter como atributos os valores de R, G, e B, que correspondem respetivamente á proporção de cada cor primária. Feito o parsing do xml e convertidos estes atributos para o respetivo valor em formato numérico, quando combinados na função *glColor3f* será possível obter a cor do respetivo plano que se insere no grupo com aquele elemento Cor.

Para isto foi criada a classe Cor, que irá apenas guardar o valor das proporções de vermelho, verde e azul nas suas variáveis R,G e B respetivamente. É também nesta classe que será feito o parsing do elemento cor e, caso este esteja a null, a cor por omissão será R=B=G = 0, que identifica a cor negra.

3.2. Rotação dos objetos

Tendo já o sistema solar desenhado de forma estática, optamos por desenvolver um mecanismo para criar as rotações dos astros em torno do Sol. De forma semelhante a outras transformações, uma órbita passa a puder ser definida no elemento Orbita do ficheiro xml e deve ter como atributos X, Y, Z e F.

Para guardar estas informações associadas a um Grupo, foi desenvolvida a classe Orbita, que contem as variáveis X,Y e Z, usadas para determinar em que eixos a rotação ocorre, e a variável F, que determina o fator/velocidade de rotação do planeta em torno do sol.

Esta rotação segundo uma órbita não é mais que uma rotação implementada com recurso à função *glrotatef,* onde o primeiro parâmetro, correspondente a um angulo designado por *spinOrbit*, é multiplicado pelo fator de rotação F e incrementado a cada iteração da renderscene.

Gerimos ainda a funcionalidade de pause/play e o reset destas rotações através das teclas “P” e “R”, respetivamente. Se pretendemos que o sistema solar esteja em movimento, partindo de uma posição estática, devemos clicar na tecla “R” para iniciar a rotação. Se o sistema se encontrar em movimento e for clicado na tecla “R” o sistema solar ficará estático e alinhado segundo o eixo dos X. Se clicarmos no “P” o modelo fica estático, pausado, e para retomar o movimento, basta clicar novamente na tecla P.

Esta funcionalidade foi implementada recorrendo a duas variáveis binárias de controlo, designadas por *rodar* e *pausa*.

4. Instruções de funcionamento do projeto

4.1. Aplicação sistemaSolar

A execução da aplicação sistemaSolar depende apenas da passagem de um único parâmetro à função main, que será o nome do ficheiro xml utilizado para gerar a cena.

Face ao formato do ficheiro xml, destaca-se que deve existir um campo ***diretoria***, com o atributo ***dir***, que irá conter a diretoria para onde se encontram os ficheiros (referentes aos vértices de um determinado modelo a desenhar) que serão usados. Estes ficheiros, identificados pelo seu nome, devem estar contidos no elemento ***models*** e o seu nome especificado no atributo ***file.*** A seguinte imagem exemplifica a forma do ficheiro xml de input utilizado e disponibilizado para o teste da aplicação sistemaSolar.

Na possibilidade de encontrar vários nomes de ficheiros no elemento *models*, estes serão todos considerados e desenhados tendo por base a mesma "origem". Esta origem será influenciada pelas transformações que possam ter sido feitas pelo grupo pai (se existir grupo pai e este tiver transformações como rotações, escalas ou translações) e consoante as transformações existentes no grupo que contem o referido conjunto de modelos.

A nível da aplicação sistemaSolar foi também tido em conta o controlo de erros, quando o nome dos ficheiros não existir ou estes não forem encontrados na diretoria especificada.

Como aspeto negativo, existe o facto do acesso ao ficheiro *.xml*, como este é passado ao main apenas pelo seu nome, é necessário alterar no código o valor de uma variável global que indica a diretoria do ficheiro *.xml* a utilizar. Esta diretoria é identificada numa variável global do tipo string, a aplicação sistemaSolar concatena o nome do ficheiro passado ao main com esta string e chega assim até ao caminho para o ficheiro *.xml* de input.

Como aspetos positivos, destacamos as diversas formas de interação e visualização dos modelos renderizados, através de diversas teclas. Estas funcionalidades serão descritas na seção seguinte e apresentam funcionalidades como zoom, várias formas de apresentação do sistema (estático ou dinâmico, com os planetas a rodar em torno do sol) e escolha do modo de desenho (linha, pontos ou a cheio).

4.2. Interação com o cenário desenhado

Para possibilitar uma melhor visualização do sistema solar desenhado foram criadas as seguintes opções de interação com a maqueta, baseadas nos conhecimentos apreendidos nas aulas práticas. Estas opções incluem funcionalidades como deslocar o ponto de visão sobre os eixos, rodar o ponto de visão da camara e realizar zoom.

Alem destes aspetos, destacamos com maior relevo a funcionalidade de colocar os planetas a rodar segundo a sua órbita em torno do sol, sendo este movimento possível de colocar em pausa, retomar ou repor, ficando o modelo apresentado estático.

Sugestões de interação com o sistema solar gerado:

1. Alteração do ponto visão da camara

Tecla Up - Deslocar para cima
Tecla Down - Deslocar para Baixo
Tecla Right - Deslocar para esquerda
Tecla Left - Deslocar para direita

2. Opções de visualização/desenho do modelo

Tecla 1 - Só linhas Tecla 2 - Preenchido Tecla 3 - Só vértices

3. Movimentação do modelo

Tecla W - Mover no sentido positivo do eixo Z
Tecla S - Mover no sentido negativo do eixo Z
Tecla A - Mover no sentido positivo do eixo X
Tecla D - Mover no sentido negativo do eixo X

4. Animações e Zoom

Tecla R - Ativar ou desativar a animação de rotação segundo as órbitas
Tecla P - Pausa/Play da animação de rotação
Tecla + - Aumentar o zoom da camara
Tecla - - Diminuir o zoom da camara
Tecla O - Restaurar todas as definições de visualização iniciais

Devido á escala dos planetas face ao sol, e pelo facto da distancia de visão inicial ser grande, possibilitando a vista de todos os astros renderizados, recomenda-se como forma de interação o uso das teclas + e – para aumentar ou reduzir o zoom e das teclas A ou D para deslocar sobre o eixo dos X, sendo assim possível "percorrer" todos os objetos representados.

5. Conclusão

Durante a realização deste trabalho fomos encontrando algumas dificuldades, mais concretamente na forma com interpretar o conteúdo do ficheiro *xml* de entrada e no modo como, utilizando as funções glPushMatrix e glPopMatrix da biblioteca glut poderíamos reutilizar as transformações já realizadas até um determinado ponto do modelo desenhado.

No nosso entender, todas as eventuais dificuldades foram totalmente ultrapassadas e compreendidas em grupo, simplificando assim a criação das estruturas de dados utilizadas e o modo como lidamos com a questão da hierarquia entre os grupos do ficheiro xml. Deste modo cumprimos todas as exigências pretendidas para esta segunda fase do trabalho, construindo uma maquete do sistema solar completa e enriquecida ainda com elementos como luas, cores dos planetas, rotações dos mesmos segundo uma órbita em torno do sol e, em tom de relaxamento do contexto do trabalho, um pequeno ovni.

No final desta implementação, consideramos ainda que fizemos uma boa reutilização das estruturas e classes já utilizadas na fase I e que este projeto terá sido a mais cativante de realizar, pela forma como conseguimos interagir e moldar o sistema final que pretendemos expor.

6. Referências

Lighthouse3d. (2017). GLUT Tutorial. [online] Available at:
http://www.lighthouse3d.com/tutorials/glut-tutorial/
[Accessed 27 Feb. 2017].

Opengl-tutorial.org. (2017). *Tutorial 3 : Matrices*. [online] Available at:
http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/
[Accessed 29 Mar. 2017].

7. Anexos

7.1. Imagens do sistema solar produzido

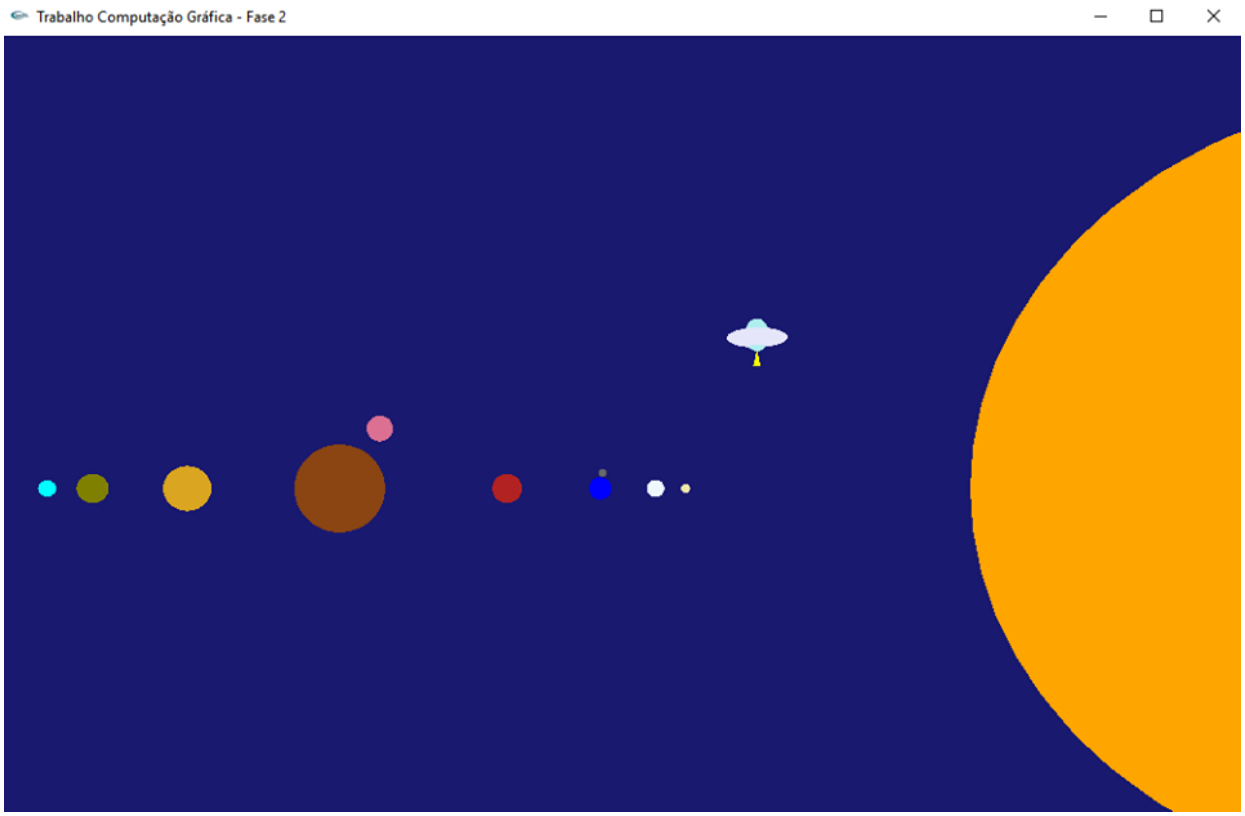

Figura 4 -Sistema solar completo

Figura 5 - Sistema solar completo

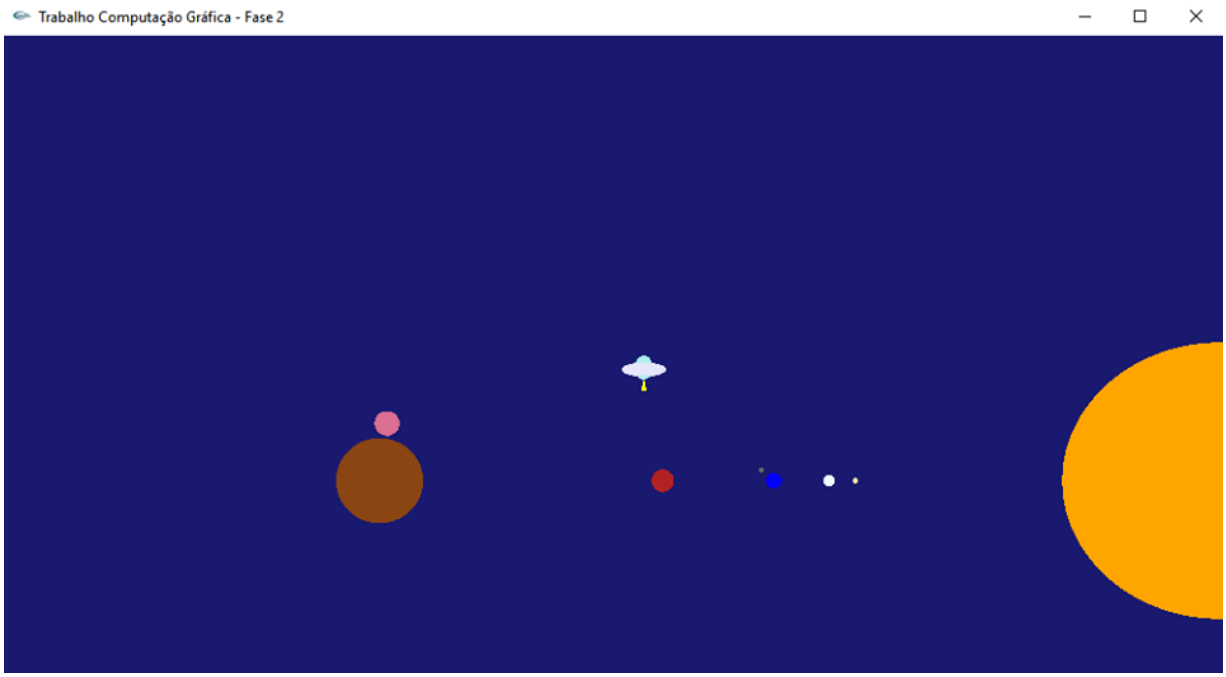

Figura 6 - Sistema solar parcial

Figura 7 - Sistema solar completo

Figura 8 – sistema Solar completo com os planetas em rotação.

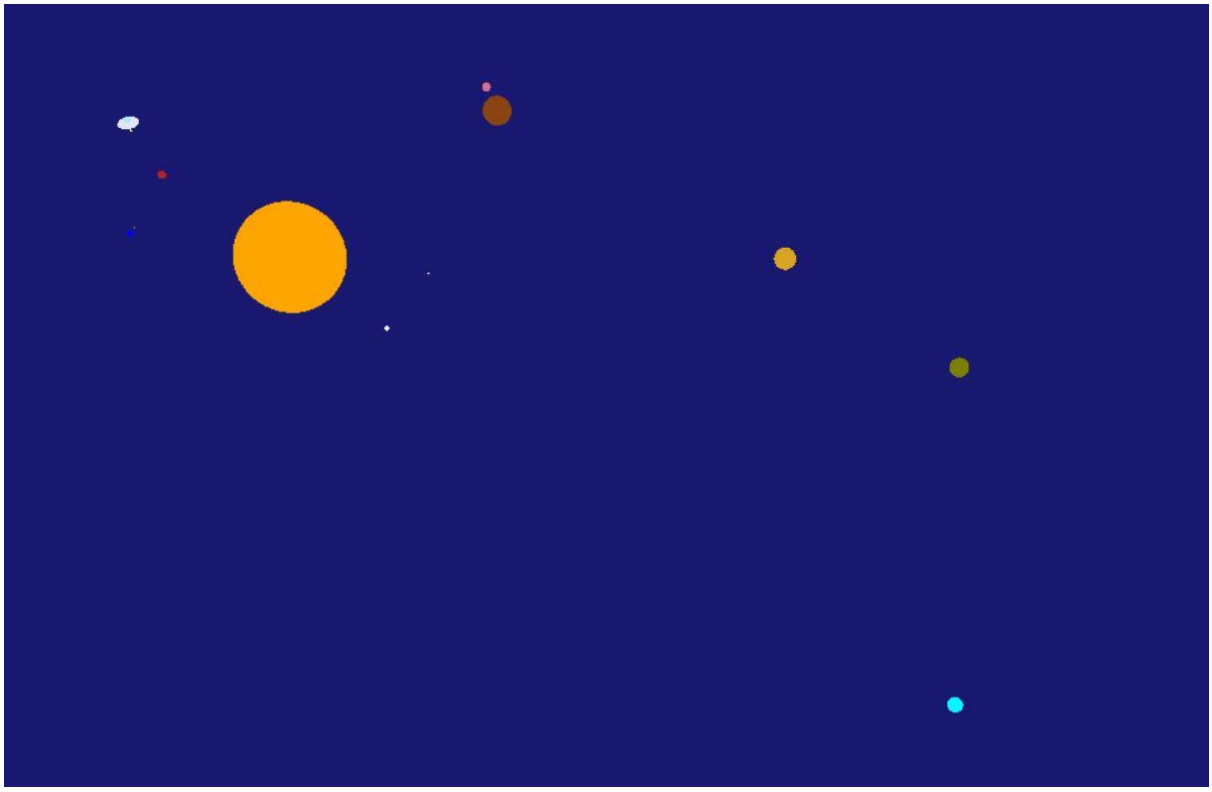

Figura 9 – Sistema solar completo após alguns segundos de rotação dos planetas segundos as suas órbitas

7.2. Código do ficheiro XMl de entrada

```
<?xml version="1.0" encoding="UTF-8"?>
<scene>
	<diretoria dir="C:\Users\migue\OneDrive\Trabalho CG\Fase II\sistemaSolar\Ficheiros\" />
	<group>
			<!-- SOL -->
			<translate X=0 Y=0 Z=0 />
			<cor R=1.0 G=0.647 B=0 />
			<scale X=23 Y=23 Z=23 />
			<rotate angle=0 axisX=0 axisY=0 axisZ=0 />
			<models>
				<model file="esfera.3d" />
			</models>
			<!-- Mercurio -->
			<group>
				<translate X=1.3 Y=0 Z=0 />
				<cor R=0.933 G=0.910 B=0.667 />
				<orbita X=0 Y=1 Z=0 F=12 />
				<scale X=0.02 Y=0.02 Z=0.02 />
				<rotate angle=0 axisX=0 axisY=0 axisZ=0 />
				<models>
					<model file="esfera.3d" />
				</models>
			</group>
			<!-- Venus -->
			<group>
				<translate X=1.4 Y=0 Z=0 />
				<cor R=0.941 G=0.973 B=1 />
				<orbita X=0 Y=1 Z=0 F=8 />
				<scale X=0.04 Y=0.04 Z=0.04 />
				<rotate angle=0 axisX=0 axisY=0 axisZ=0 />
				<models>
					<model file="esfera.3d" />
				</models>
			</group>
			<!-- Terra -->
			<group>
				<translate X=1.6 Y=0 Z=0 />
				<cor R=0.0 G=0.0 B=1 />
				<orbita X=0 Y=1 Z=0 F=1 />
```

```
        <scale X=0.055 Y=0.055 Z=0.055 />
        <rotate angle=0 axisX=0 axisY=0 axisZ=0 />
        <models>
                <model file="esfera.3d" />
        </models>
        <!-- Lua -->
        <group>
                <translate X=0.7 Y=0.7 Z=0.7 />
                <orbita X=0 Y=1 Z=0 F=1 />
                <cor R=0.412 G=0.412 B=0.412 />
                <scale X=0.35 Y=0.35 Z=0.35 />
                <rotate angle=0 axisX=0 axisY=0 axisZ=0 />
                <models>
                        <model file="esfera.3d" />
                </models>
        </group>
</group>
<!-- OVNI -->
<group>
        <translate X=2 Y=0.5 Z=1 />
        <cor R=0.902 G=0.902 B=0.980 />
        <orbita X=0 Y=1 Z=0 F=1 />
        <scale X=0.2 Y=0.06 Z=0.2 />
        <rotate angle=0 axisX=0 axisY=0 axisZ=0 />
        <models>
                <model file="esfera.3d" />
        </models>
        <group>
                <translate X=0 Y=0.15 Z=0 />
                <cor R=0.686 G=0.933 B=0.933 />
                <scale X=0.4 Y=1.8 Z=0.4 />
                <orbita X=0 Y=1 Z=0 F=1 />
                <rotate angle=0 axisX=0 axisY=0 axisZ=0 />
                <models>
                        <model file="esfera.3d" />
                </models>
                <group>
                        <translate X=0 Y=-0.7 Z=0 />
                        <cor R=1.0 G=1 B=0 />
                        <scale X=0.3 Y=0.1 Z=0.3 />
                        <orbita X=0 Y=1 Z=0 F=1 />
                        <rotate angle=0 axisX=0 axisY=0 axisZ=0 />
                        <models>
                                <model file="cone.3d" />
                        </models>
                </group>
        </group>
</group>
<!-- Marte -->
<group>
                <translate X=2 Y=0 Z=0 />
                <cor R=0.698 G=0.133 B=0.133 />
                <orbita X=0 Y=1 Z=0 F=0.9 />
                <scale X=0.08 Y=0.08 Z=0.08 />
                <rotate angle=0 axisX=0 axisY=0 axisZ=0 />
                <models>
                        <model file="esfera.3d" />
                </models>
</group>
<!-- Jupiter -->
<group>
        <translate X=3 Y=0 Z=0 />
        <cor R=0.545 G=0.271 B=0.075 />
        <scale X=0.3 Y=0.3 Z=0.3 />
        <orbita X=0 Y=1 Z=0 F=0.6 />
        <rotate angle=0 axisX=0 axisY=0 axisZ=0 />
        <models>
                <model file="esfera.3d" />
        </models>
        <!-- Lua Júpiter -->
        <group>
                <translate X=0 Y=0.7 Z=0.5 />
                <cor R=0.859 G=0.439 B=0.576 />
                <scale X=0.3 Y=0.3 Z=0.3 />
                <orbita X=0 Y=1 Z=0 F=0.6/>
                <rotate angle=0 axisX=0 axisY=0 axisZ=0 />
                <models>
                        <model file="esfera.3d" />
                </models>
        </group>
</group>
<!-- Saturno -->
<group>
                <translate X=4.5 Y=0 Z=0 />
                <cor R=0.855 G=0.647 B=0.125 />
                <scale X=0.2 Y=0.2 Z=0.2 />
                <orbita X=0 Y=1 Z=0 F=0.4 />
```

```
                                <rotate angle=0 axisX=0 axisY=0 axisZ=0 />
                                <models>
                                        <model file="esfera.3d" />
                                </models>
                        </group>
                        <!-- Urano -->
                        <group>
                                <translate X=6 Y=0 Z=0 />
                                <cor R=0.502 G=0.502 B=0 />
                                <scale X=0.16 Y=0.16 Z=0.16 />
                                <orbita X=0 Y=1 Z=0 F=0.35 />
                                <rotate angle=0 axisX=0 axisY=0 axisZ=0 />
                                <models>
                                        <model file="esfera.3d" />
                                </models>
                        </group>
                        <!-- Neptuno -->
                        <group>
                                <translate X=7 Y=0 Z=0 />
                                <cor R=0 G=1 B=1 />
                                <scale X=0.1 Y=0.1 Z=0.1 />
                                <orbita X=0 Y=1 Z=0 F=0.25 />
                                <rotate angle=0 axisX=0 axisY=0 axisZ=0 />
                                <models>
                                        <model file="esfera.3d" />
                                </models>
                        </group>
                </group>
</scene>
```