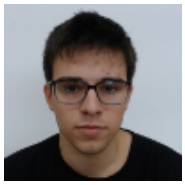
RELATÓRIO PRÁTICO FASE 2

GRUPO 30

Ana Almeida, A83916

André Figueiredo, A84807



Luís Ferreira, A86265

Rafael Lourenço, A86266

Mestrado Integrado em Engenharia Informática 2019/2020

Computação Gráfica

Universidade do Minho

29 de Março de 2020

Conteúdo

Capítulo 1

Introdução

1.1 Contextualização

Foi-nos proposto, no âmbito da UC Computação Gráfica, a criação de um motor gráfico genérico para representar objetos a 3 dimensões, com o auxílio de duas ferramentas - *OpenGL* e *C++*.
O projeto está dividido em várias fases distintas, sendo que nesta segunda serão criados cenários hierárquicos usando transformações geométricas, com o objetivo de criar um modelo estático do Sistema Solar.

1.2 Resumo

Visto que esta se trata da segunda parte do projeto prático é natural que se mantenham algumas das funcionalidades criadas na primeira parte e, por outro lado, algumas delas sejam alteradas, de modo a cumprir com os requisitos necessários. Assim, a principal mudança que surge nesta fase está inteiramente relacionada com a forma como o engine, previamente criado na fase anterior, lê e processa a informação contida nos ficheiros XML que irá receber.
A estrutura destes ficheiros sofre uma grande mudança. Agora, em vez de estes conterem unicamente o nome dos ficheiros com as primitivas que se pretende exibir, estes contêm a formação de diversos grupos hierárquicos com esses mesmos ficheiros. Estes grupos têm associado a si, diversas transformações geométricas (*translate*, *rotate* e *scale*) que serão responsáveis pelo modo como cada uma das primitivas, previamente criadas na fase anterior, são exibidas.
Deste modo, vai ser necessário, não só alterar a forma como o nosso engine lê estes mesmos ficheiros, como também a forma como este processa essa mesma informação. Assim, será necessária a criação de novas classes que terão como objetivo armazenar e relacionar esta mesma informação.
Tudo isto tem como finalidade conseguirmos gerar e exibir primitivas gráficas que, no seu conjunto, representem um modelo estático do Sistema Solar. Desta forma, para além dos requisitos mínimos exigidos, decidimos implementar algumas funcionalidades extra como a inclusão da primitiva gráfica *Torus* para uma representação mais realista dos anéis de Saturno, além da implementação da funcionalidade Cor, que irá acompanhar as restantes transformações gráficas e ainda uma nova câmera para permitir ao utilizador navegar livremente pelo cenário.

Capítulo 2

Arquitetura do código

Tendo em mente a continuação do trabalho desenvolvido na fase anterior, mantivemos as duas aplicações principais previamente desenvolvidas, gerador e engine, tendo sido este último alvo de algumas modificações mais acentuadas, tendo em vista o cumprimento dos requisitos necessários.

2.1 Aplicações

Nesta secção são apresentadas as aplicações fundamentais que permitem gerar e exibir os diferentes cenários pretendidos. Uma vez que houve alteração da estrutura dos ficheiros de configuração escritos em XML, foi necessária uma transformação da forma como o Engine processa estes ficheiros.

2.1.1 Gerador

generator.cpp - Tal como explicado na fase anterior, esta é a aplicação onde estão definidos os algoritmos das diferentes formas geométricas a desenvolver de forma a gerar os respetivos vértices. Para além das primitivas gráficas desenvolvidas na fase anterior, o grupo decidiu acrescentar a primitiva *Torus*, e como tal, foi necessário acrescentar ao *Generator* novas funcionalidades que lhe permitissem gerar esta mesma primitiva. Tudo o resto manteve-se idêntico ao que foi previamente desenvolvido na fase anterior.

```
#_____________________________ HELP ____________________________________#
|                                                                        |
|   Usage: ./generator {COMMAND} ... {OUTPUT FILE}                       |
|                      [-h]                                              |
|                                                                        |
|   COMMANDS:                                                            |
| - plane [SIZE]                                                         |
|       Creates a square in the XZ plane, centred in the origin.         |
|                                                                        |
| - box [SIZE X] [SIZE Y] [SIZE Z] [DIVISION X] [DIVISION Y] [DIVISION Z] |
|       Creates a box with the dimensions and divisions specified.       |
|                                                                        |
| - sphere [RADIUS] [SLICE] [STACK]                                      |
|       Creates a sphere with the radius, number of slices and           |
|       stacks given.                                                    |
|                                                                        |
| - cone [RADIUS] [HEIGHT] [SLICE] [STACK]                               |
|       Creates a cone with the radius, height, number of slices         |
|       and stacks given.                                                |
|                                                                        |
| - torus [RADIUS] [WIDTH] [HEIGHT] [SLICE] [STACK]                      |
|       Creates a cone with the radius, width, height, number of slices  |
|       and stacks given.                                                |
|                                                                        |
|   OUTPUT FILE:                                                         |
| In the file section you can specify any file in which you wish         |
| to save the coordinates generated with the previous commands.          |
|                                                                        |
#________________________________________________________________________#
```

Figura 2.1: Menu de ajuda do Generator

2.1.2 Motor

engine.cpp - Aplicação que possui a capacidade de apresentar a janela, exibindo os modelos indicados no ficheiro *ss.xml* e interagir com o ambiente. Com a alteração do ficheiro XML e da respetiva estrutura, foi necessário alterar o modo como se faz o seu parsing, ou seja, a função *parseXML()*. Esta função é uma função recursiva que percorre todos os elementos do ficheiro XML e, para cada elemento, é novamente chamada com o próprio elemento e uma instância do *ContainerElement* pai, como argumentos.

```
#_____________________________ HELP _____________________________#
|                                                                 |
| Usage: ./engine                                                 |
|                 [-h]                                            |
|                                                                 |
|   FILE:                                                         |
| Specify a path to an XML file in which the information about    |
| the models you wish to create are specified                     |
|                                                                 |
|   MOVE:                                                         |
|   w: Move your position forward                                 |
|                                                                 |
|   s: Move your position back                                    |
|                                                                 |
|   a: Move your position to the left                             |
|                                                                 |
|   d: Move your position to the right                            |
|                                                                 |
|   ↑: Rotate your view up                                        |
|                                                                 |
|   ↓: Rotate your view down                                      |
|                                                                 |
|   ←: Rotate your view to the left                               |
|                                                                 |
|   →: Rotate your view to the right                              |
|                                                                 |
|   Pag_UP: Zoom in                                               |
|                                                                 |
|   Pag_Down: Zoom out                                            |
|                                                                 |
|   q: Go up                                                      |
|                                                                 |
|   z: Go down                                                    |
|                                                                 |
|   FORMAT:                                                       |
|   .: Change the figure format into points                       |
|                                                                 |
|   -: Change the figure format into lines                        |
|                                                                 |
|   ,: Fill up the figure                                         |
|                                                                 |
|   e: Remove/draw Axis                                           |
#_________________________________________________________________#
```

Figura 2.2: Menu de ajuda do Engine

2.2 Classes

Para esta nova fase o grupo decidiu organizar melhor o código e criar 9 novas classes (3 delas correspondentes a cada uma das transformações geométricas: *Translation*, *Rotation* e *Scale* e cujas diferenças estão ilustradas na figura 2.3, respetivamente) para tratar dos diversos elementos do XML, sendo estas:

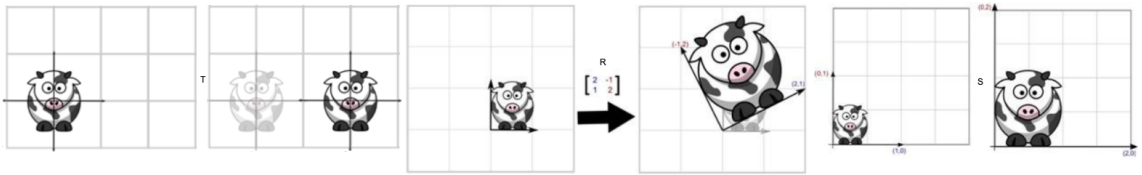


Figura 2.3: Exemplos Transformações

- Interface *BaseElement* que todos os outros implementam;
- *TranslateElement*;
- *RotateElement*;
- *ScaleElement*;
- *ModelElement*;
- Classe abstrata *ContainerElement*;
- *SceneElement*;
- *GroupElement*;
- *ModelsElement*.

2.2.1 Base Element

baseElement.h - Interface que todos os outros elementos terão de implementar, herdando o método *Apply()*, usado para renderizar os respetivos elementos.

2.2.2 Translate Element

translateElement.h - Classe que guarda os atributos x, y e z, necessários para a invocação da função OpenGL *glTranslatef(x, y, z)* realizada no método *Apply()* que é por esta classe implementado.

2.2.3 Rotate Element

rotateElement.h - Classe que guarda os atributos *angle*, *axisX*, *axisY* e *axisZ* necessários para a invocação da função OpenGL *glRotatef(angle, axisX, axisY, axisZ)* realizada no método *Apply()* que é por esta classe implementado.

2.2.4 Scale Element

translateElement.h - Classe que guarda os atributos x, y e z necessários para a invocação da função OpenGL *glScalef(x, y, z)* realizada no método *Apply()* que é por esta classe implementado.

2.2.5 Model Element

modelElement.h - Classe que guarda a informação relativa a uma figura, ou seja, guarda numa estrutura *Triangulo* de nome *figure*, os dados lidos de um ficheiro fornecido, assim como a sua cor nos atributos r, g, b. Foi ainda criado um macro chamado *deviation* de modo a alterar a cor dos vértices eliminando o fator uniforme da cor da figura. Aquando da invocação do método *Apply()* é renderizada a figura utilizando as funções OpenGL *glColor3f()*, *glVertex3f()* para cada vértice.

2.2.6 Container Element

containerElement.h - Classe abstrata extendida por todos os elementos que podem ter filhos, como é o caso da *scene*, *group* e *models*. Esta classe tem como atributo um vetor para ponteiros de *BaseElements* chamado *children*, onde serão guardados todos os *BaseElements* filhos. E ainda implementa um método de nome *addChild()* usado para guardar no vetor *children* o ponteiro para o filho recebido como argumento.

2.2.7 Scene Element

sceneElement.h - Classe "Root"que extende a classe *ContainerElement*, ou seja, é a classe que tem o vetor *children* principal, onde serão colocados todos os ponteiros dos seus filhos (*groups*). O seu único método, *Apply()*, faz a invocação do *Apply()* de todos os seus filhos. O Engine tem como variável global uma instância desta classe de modo a ser chamado o seu *Apply()* quando for para renderizar o "mundo"na função *renderScene()*.

2.2.8 Group Element

groupElement.h - Classe que extende a classe *ContainerElement*. O seu método *Apply()* é similar ao do *SceneElement*, ou seja, também faz a invocação do *Apply()* de todos os seus filhos, com uma pequena diferença: antes usa a função OpenGL *glPushMatrix()* e depois usa a *glPopMatrix()*, de maneira a que qualquer transformação efetuada na matriz seja só aplicada nos seus filhos.

2.2.9 Models Element

modelsElement.h - Classe que extende a classe *ContainerElement*. O seu método *Apply()*, assim como a classe *GroupElement*, é similar ao do *SceneElement*, ou seja, também invoca o *Apply()* de todos os seus filhos, com a ressalva dos filhos só poderem ser instâncias da classe *ModelElement* e de chamar as funções OpenGL *glBegin(GL_TRIANGLES)* e *glEnd()*, antes e depois, respetivamente, para delimitar os vértices das figuras dos filhos (*ModelElement*).

2.3 Ficheiros Auxiliares

2.3.1 Triangle

triangle.h - Módulo que guarda e trata da estrutura *Triangulo*, isto é, implementa as funções da sua inicialização e de leitura de um ficheiro (recebido como argumento) colocando o resultado numa estrutura *Triangulo* de nome *figure* (também recebida como argumento).

2.3.2 Camera

camera.h - Módulo que trata de todas as operações relacionadas com o posicionamento da câmera.
Tem os métodos capazes de alterar a posição para onde o utilizador está a olhar e a distância, entre outros. Também contém o método para desenhar os eixos de coordenadas.

As variações em cada coordenada dependem da direção para a qual se está a olhar, por exemplo, se estiver na posição (0, 0, 5) a olhar para a origem e mover-se para a direita, passará a estar na posição (1, 0, 5) e a olhar para a posição (1, 0, 0), ou seja, o ponto para o qual se está a olhar acompanha o movimento da câmera.
Contudo, se estiver na posição (5, 0, 0) e a olhar para (0, 0, 0) e andar para a direita, não irá simplesmente mover-se uma unidade em $\mathbf{x}$, mas sim deslocar-se em $\mathbf{z}$, neste caso, a câmera passará a estar em (5, 0, -1) e o ponto de foco é (0, 0, -1).
O nosso objetivo foi tentar replicar o comportamento de uma câmera de um jogo FPS.

Capítulo 3

Generator

O **Gerador**, tal como na fase anterior, é responsável por gerar ficheiros que contêm o conjunto de vértices das primitivas gráficas que se pretende gerar, conforme os parâmetros escolhidos. A única mudança que ocorreu nesta transição de fases foi a inclusão de uma nova primitiva, o *Torus*, passando assim a fazer parte do conjunto das 5 primitivas geométricas que o gerador está apto a gerar. Esta foi criada para poder representar os anéis de Saturno, como será mostrado mais à frente.

3.1 Primitiva geométrica: Torus

Um *Torus* é um sólido geométrico que apresenta o formato aproximado de uma câmara de pneu ou donut. Em geometria, pode ser definido como o lugar geométrico tridimensional formado pela rotação de uma superfície circular plana de raio interior, em torno de uma circunferência de raio exterior.
Como tal, os parâmetros para gerar um *Torus* são **R** (raio interior), **L** (largura do anel), **H** (altura do anel), **slices** (número de divisões no raio interior) e **stacks** (número de divisões por secção radial, sendo também divisões de ângulo na elipse).
Foi decidido que o *Torus* não seria representado como uma rotação de um superficie circular, mas sim, uma rotação de uma elipse[1], de forma a chegar a valores que melhor representem os aneis de Saturno[2].

3.1.1 Algoritmo

Para a construção do *Torus* é preciso considerar que a sua constituição se baseia no raio interior, largura e altura do anel. Para tal, é preciso estabelecer que eixos vão ficar responsáveis por definir a circunferência e elipse que vamos percorrer para poder desenhar o *Torus*. Com isto, os eixos X e Z definem uma circunferência com o raio interior R e os eixos X, Y e Z definem uma elipse com altura H e largura L, centrada na circunferência.
Para podermos iterar pela cirunferência, temos que recorrer ao parâmetro *slices* onde cada porção é definida por uma amplitude dada por:

$$\delta\theta = (2 * \pi)/slices$$

Por sua vez, para iterar pela elipse, temos que recorrer, desta vez, ao parâmetro *stacks*, onde a amplitude é dada por:

[1] Se a altura for igual à largura, resultará num *Torus* clássico
[2] Usaremos o *glScale*, todavia, desta forma, chegaremos mais depressa aos valores

$$\delta\alpha = (2 * \pi)/stack$$

Desta forma, com auxílio das funções *cos* e *sin*, podemos obter facilmente os pontos que formam estas superfícies. O desenho inicia-se ($i = 0$) por definir os pontos:

$$r = R + L/2$$
$$p_{0_x} = r * cos(i * \delta\theta) \qquad p_{0_z} = r * sin(i * \delta\theta)$$
$$p_{1_x} = r * cos((i+1) * \delta\theta) \qquad p_{1_z} = r * sin((i+1) * \delta\theta)$$

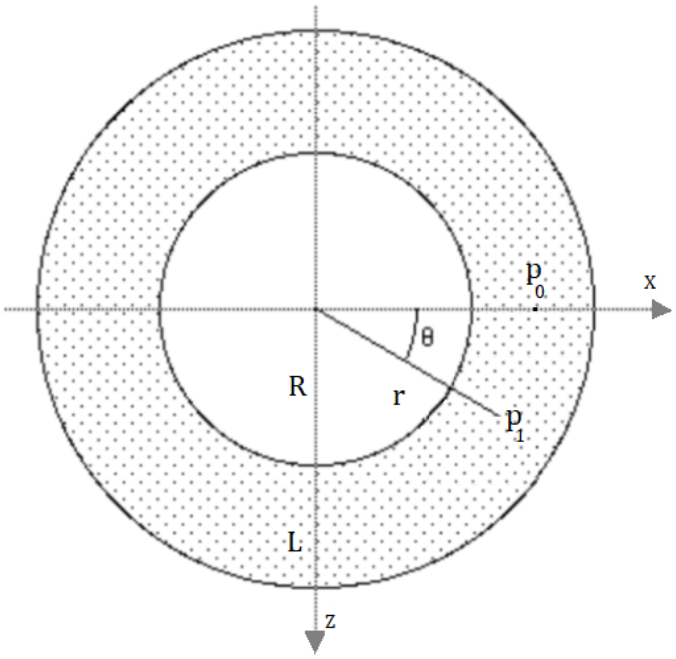


Figura 3.1: Ilustração do corte horizontal do *Torus* (vista de cima)

O ponto p_0 representa o ponto de referência da elipse, isto é onde será centrada e p_1 o próximo ponto (próxima elipse, ao incrementar a amplitude em $\delta\theta$). Desta forma, passámos a desenhar entre estes dois limitadores um anel, como representado na figura acima. Para tal, percorremos a circunferência interna, adiciondo $\delta\theta$ em cada interação, para dar a volta à essa circunferência interna.
Para definir agora os pontos em relação ao centro da elipse, temos de calcular os seguintes pontos:

$$x_0 = L * cos(i * \delta\theta) * cos(j * \delta\alpha)$$
$$x_1 = L * cos((i+1) * \delta\theta) * cos(j * \delta\alpha)$$
$$x_2 = L * cos(i * \delta\theta) * cos((j+1) * \delta\alpha)$$
$$x_3 = L * cos((i+1) * \delta\theta) * cos((j+1) * \delta\alpha)$$
$$y_0 = H * sin(j * \delta\alpha)$$
$$y_1 = H * sin((j+1) * \delta\alpha)$$
$$z_0 = L * cos(i * \delta\theta) * sin(j * \delta\alpha)$$
$$z_1 = L * cos((i+1) * \delta\theta) * sin(j * \delta\alpha)$$
$$z_2 = L * cos(i * \delta\theta) * sin((j+1) * \delta\alpha)$$
$$z_3 = L * cos((i+1) * \delta\theta) * sin((j+1) * \delta\alpha)$$

Desta forma, podemos obter os vértices para desenhar o triângulo simplesmente adicionando as 2 componentes: posição em relação à origem e posição em relação ao centro da elipse, como representado na figura abaixo.

$$
\begin{array}{lll}
p_{0_x} = p_{1_x} + x_1 & p_{0_y} = y_0 & p_{0_z} = p_{1_z} + z_1 \\
p_{1_x} = p_{0_x} + x_0 & p_{1_y} = y_0 & p_{1_z} = p_{0_z} + z_0 \\
p_{2_x} = p_{0_x} + x_2 & p_{2_y} = y_1 & p_{2_z} = p_{0_z} + z_2
\end{array}
$$

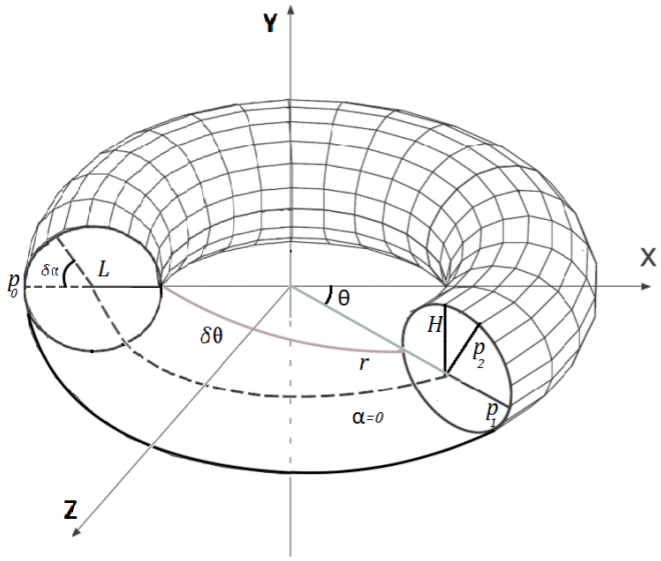


Figura 3.2: Ilustração do posicionamento dos pontos

Tendo também em mente a seleção dos pontos no sentido contrário aos ponteiros do relógio, de forma a ser visto pela parte de fora. Desta forma, as iterações baseiam-se em cada anel, iterando dentro desse anel (de forma a completar a elipse) e passar para o próximo anel (até ter a circunferência terminada), tal como pode ser visto na figura abaixo.

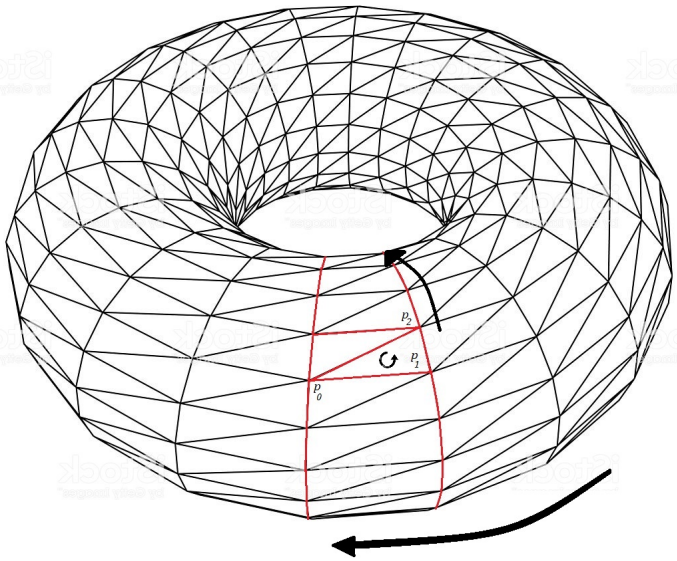


Figura 3.3: Ilustração da iteração sobre o *Torus*

Capítulo 4

Engine

O **Engine** é responsável por receber ficheiros escritos em XML. Na primeira fase, o funcionamento deste era simples, sendo apenas constituído por uma *tag scene* com várias *tags model* que tinham como atributo o nome do ficheiro respetivo. Nesta segunda fase, foram adicionadas mais algumas *tags* que vieram acrescentar alguma complexidade ao processo de leitura do ficheiro XML para, por fim, apresentar a *scene* completa ao utilizador. Foi criada uma instância global *SceneElement* para guardar toda essa informação.

4.1 Processo de leitura

Como referido anteriormente o processo de leitura é feito pela função recursiva *parseXML()* que, como vamos ver, tem duas implementações com a diferenciação nos argumentos recebidos. Depois de obtido o nome do ficheiro XML é calculado o path para este e passado como argumento a esta função, ou seja, (*parseXML(xml_filename)*). Para facilitar a compreensão do algoritmo vai ser exemplificado, de seguida, algumas chamadas da função *parseXML()* para a leitura do ficheiro *ss.xml*.
Na primeira iteração é aberto o ficheiro e obtido o primeiro elemento que será uma *scene*, colocando tanto o elemento (**elem**) como a instância *SceneElement* global (**se**) como argumentos, ou seja, *parseXML(elem,se)*, o que vai invocar a segunda implementação da função *parseXML()*.
Esta segunda iteração vai comparar o valor do elemento (**elem**→*Value()*) com a string *"scene"*, que vai ser verdadeiro, e vai provocar a iteração de todos os filhos da *scene*, que serão *groups* se o ficheiro XML estiver correto, executando novamente o *parseXML()*, desta vez com o *group* filho (**aux**) e novamente a *SceneElement* global (**se**) como argumentos, isto é, *parseXML(aux,se)*.
Mais uma iteração, mais uma comparação: desta vez o valor do elemento (**elem** → *Value()*) será uma string *"group"* (entrando noutro $if case$) onde vai ser criada uma instância de *GroupElement* (**ge**) e adicionada ao vetor de pointers da instância *ContainerElement* parent que, mais uma vez, será a *SceneElement* global (**se**) e vão ser iterados todos os filhos invocando a função *parseXML()*, desta vez passando como argumentos o elemento filho (**aux**) e a nova instância criada.
Na quarta iteração entrará no $if case$ do *scale*, que tem um tratamento idêntico a um *rotate* e *translate*: lê os seus atributos colocando-os nas respetivas variáveis, criando de seguida uma instância do seu elemento com as variaveis obtidas e invocando o método *addChild()* do parent, neste caso o *group* anterior, para adicionar ao *vector* do pai o ponteiro para a sua instância.

Esta quinta iteração, é uma invocação para outro filho do elemento *group* tratado anteriormente, desta vez o filho é um *models*. O $if case$ do *model* é similar ao do *group*, contudo, ao invés de criar uma instância *group*, cria uma *models*, invocando também o método *addChild()* do parent que, mais uma vez será, o *group* anteriormente instanciado, acrescentando também o ponteiro no seu *vector*. De seguida percorre todos os seus filhos (**aux**), que serão elementos *model*, invocando novamente a função *parseXML()* passando como argumentos o filho (**aux**) e a sua instancia (***models**).
Chegando à sexta e última iteração desta exemplificação, sendo o elemento um *model*, entra no respetivo $if case$, que vai tentar obter o nome do ficheiro do modelo, assim como a sua cor *rgb*, e inicializar uma instância *model* com os valores obtidos, o que vai resultar na leitura do ficheiro e criação de uma nova figura com os seus dados. Adicionando-se, finalmente, no *vector* do pai, que será o *models* anterior, com a invocação do *addChild()*.
E assim sucessivamente até chegar ao último filho que, no caso da **scene** é um *group*, no caso do **group** é um *models* e no caso do **models** é um *model*.
É de notar que sempre que é dito que são iterados todos os seus filhos, eles são realmente iterados, mas só é passado para o próximo filho quando o corrente já está completamente tratado.

4.2 Estruturas de dados

Através do algoritmo descrito na secção anterior e das classes apresentadas na secção 2.2, facilmente concluímos qual será a estrutura de dados necessária para armazenar toda a informação recolhida durante o parsing, que será a instância global *SceneElement* (**se**) que terá um vetor com poiters para todos os seus filhos que, por sua vez, terão para os seus filhos e assim sucessivamente até chegar ao model que terá uma estrutura *Triangulo* com a figura guardada.

4.3 Processo de renderização

No que toca ao processo de renderização, o processo é extremamente simples. Foi criado um método *Apply()* em todos os elementos e, invocando esse método na instância global *SceneElement* (**se**), todas as outras instâncias terão esse método invocado. Assim como no processo de leitura será feita uma exemplificação do processo de renderização no caso do ficheiro *ss.xml*. Como foi dito, é chamado o método *Apply()* na *SceneElement* global que vai se limitar a invocar o método *Apply()* para todos os seus filhos, no nosso caso, começa por invocar o do *group* seguinte. Este, já um pouco diferente, vai fazer o push da matriz de transformações com um *glPushMatrix()* e chama o *Apply()* para todos os seus filhos fazendo, no final, o pop da matriz com um *glPopMatrix()*. O primeiro filho é um *scale*, cujo *Apply()* é apenas um *glScalef(x,y,z)*, levando à execução do *Apply()* do segundo filho que é um *models*, que delimita a escrita dos modelos com um *glBegin(GL_TRIANGLES)* e um *glEnd()*, invocando lá dentro, mais uma vez, o *Apply()* de todos os filhos que, neste caso, será só um *model* que vai pegar na estrutura *Triagulo* figure e desenhar a figura, vértice a vértice, cor a cor. De seguida, vem a execução do *Apply()* próximo filho do *group* anterior que será também ele um *group* ... isto vai prosseguindo até todos os filhos, filhos de filhos, e por diante, serem tratados, ou seja, quando todas as figuras forem desenhadas na scene.

Capítulo 5

Análise de Resultados - Sistema Solar

O resultado final correspondeu ao esperado pelo grupo, ou seja,

- Todos os planetas foram representados à escala para percecionar a realidade;
- As suas cores foram alteradas para fazer uma distinção clara e fiel;
- Foram incluídos alguns satélites naturais;
- A sua disposição não consistiu numa linha reta, para criar a sensação do movimento translacional a que estão sujeitos.

5.1 Visualização

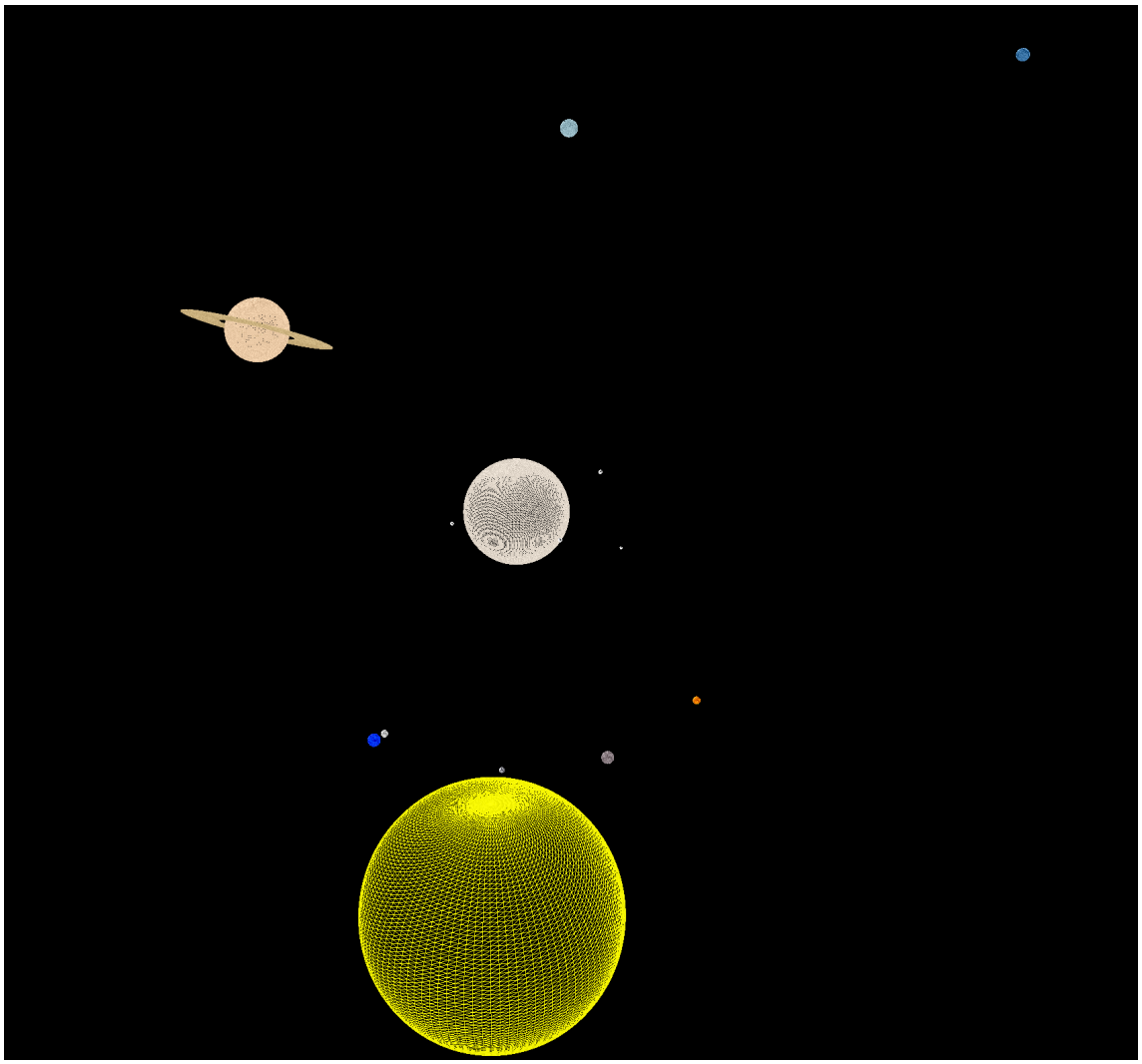


Figura 5.1: Representação do Sistema Solar completo

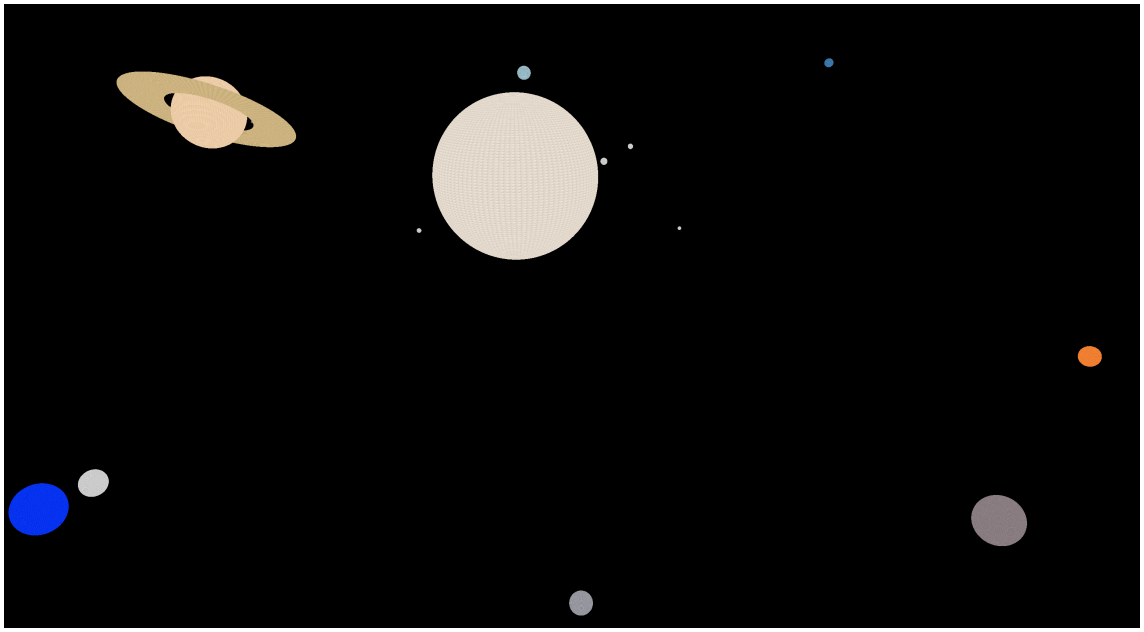


Figura 5.2: Representação do Sistema Solar visto do Sol

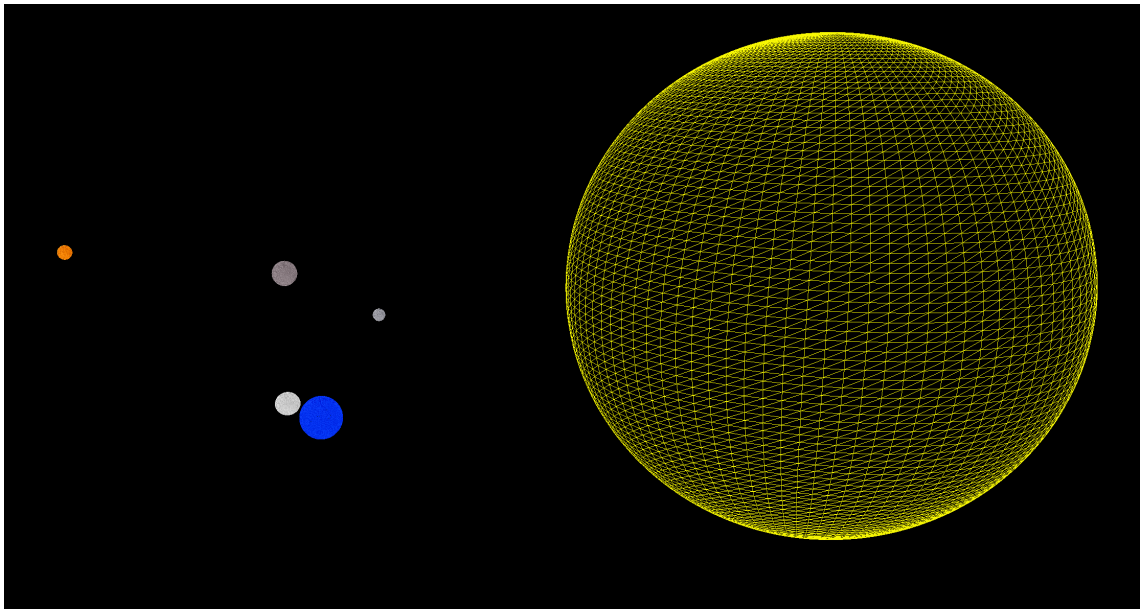


Figura 5.3: Representação dos quatro primeiros planetas do Sistema Solar com o satélite natural Lua

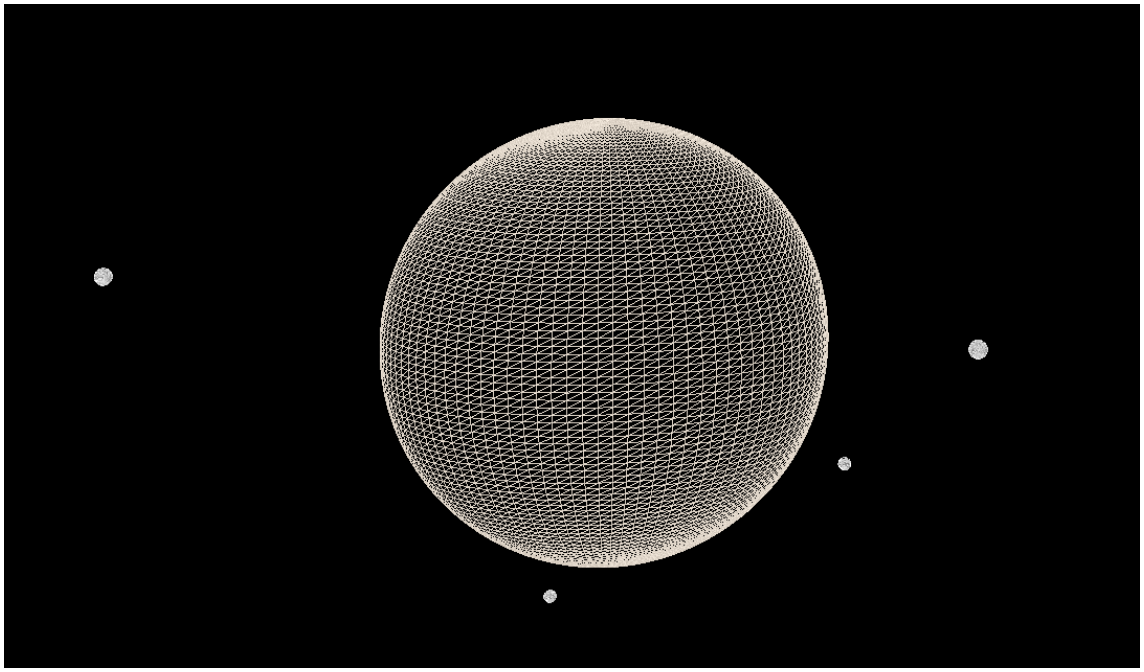


Figura 5.4: Representação do planeta Júpiter com os satélites naturais Europa, Ganímedes, IO e Calisto

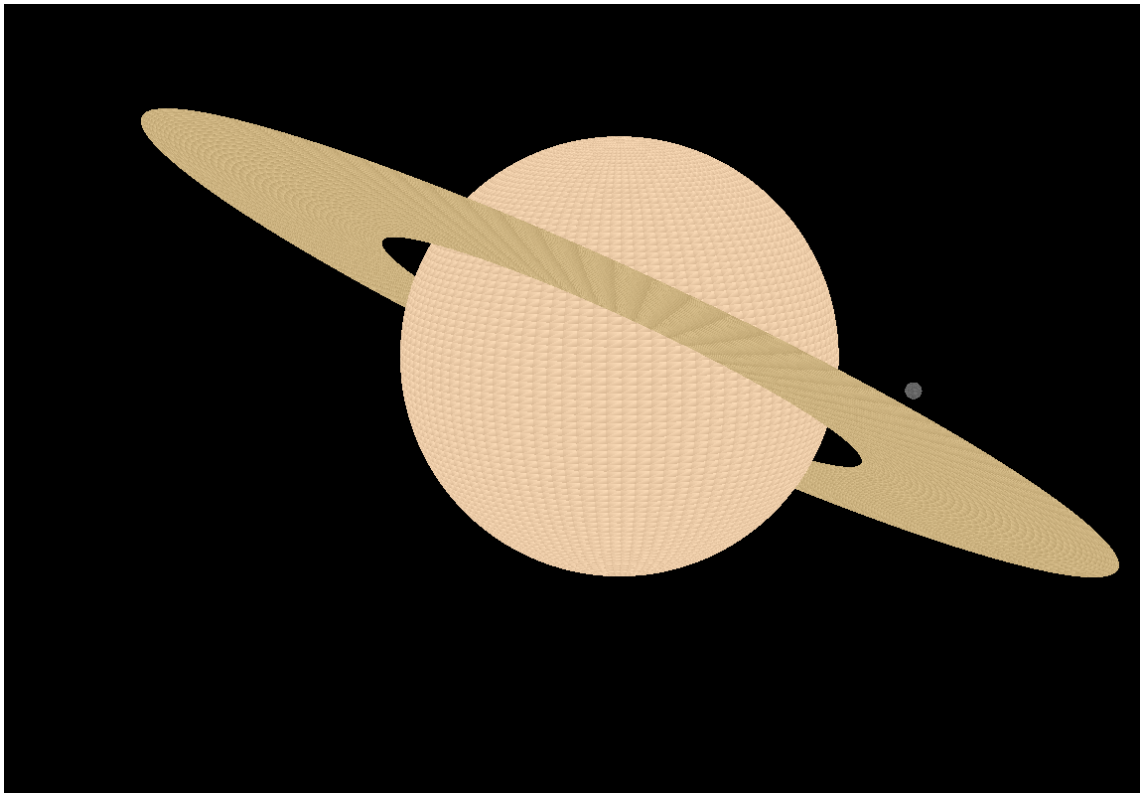


Figura 5.5: Representação do planeta Saturno com o satélite natural Titã (preenchido)

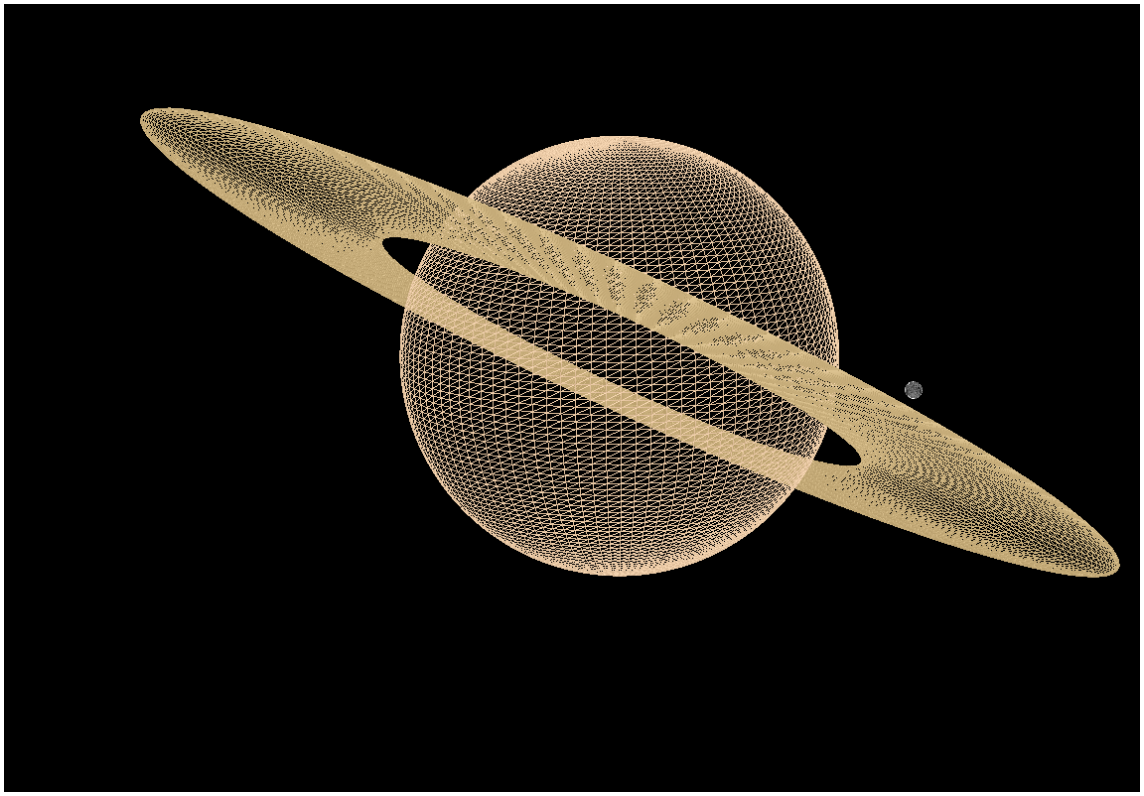


Figura 5.6: Representação do planeta Saturno com o satélite natural Titã (por linhas)

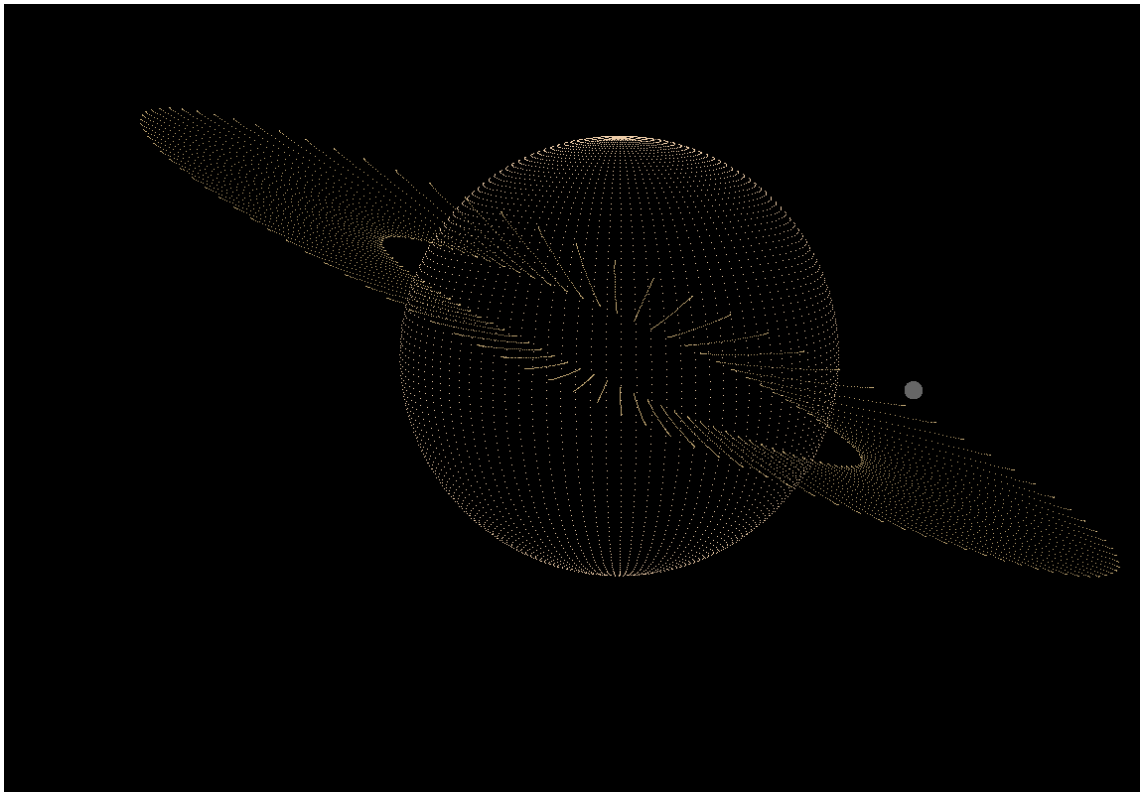


Figura 5.7: Representação do planeta Saturno com o satélite natural Titã (por pontos)

Capítulo 6

Conclusão

Ao longo desta segunda fase conseguimos trabalhar um pouco melhor nos requisitos pedidos, pois, para além de serem menores, já haviamos adquirido algum ritmo e conhecimento com a realização da primeira fase.
No nosso entender, conseguimos atingir o resultado final desta fase, uma vez que, o modelo do Sistema Solar desenvolvido se enquadra perfeitamente naquilo que era esperado, tendo até cuidado com alguns aspetos da realidade (movimento de translação dos planetas, satélites naturais, entre outros).
Contudo, nas restantes fases, esperámos conseguir melhorar cada vez mais o modelo em questão, de forma a torná-lo o mais realista e agradável à vista possível.

Capítulo 7

Anexos

Ficheiro de input para representar o sistema solar:

```
<scene>
    <!--Sun-->
    <group>
        <scale X=2 Y=2 Z=2 />
        <models>
            <model file="sphere.3d" R=0.98 G=0.98 B=0 />
        </models>
        <!-- Mercury -->
        <group>
            <translate X=-0.0378246429902437 Z=1.7996025384463816 />
            ↪ <!-- alpha: -0.021015237378258278 | radius: 1.8
            ↪ -->
            <scale X=0.0216 Y=0.0216 Z=0.0216 />
            <models>
                <model file="sphere.3d" R=0.59 G=0.59 B=0.621 />
            </models>
        </group>
        <!-- Venus -->
        <group>
            <translate X=-0.9541075765499087 Z=1.982341729462925 />
            ↪ <!-- alpha: -0.4485786692897512 | radius: 2.2 -->
            <scale X=0.0531 Y=0.0531 Z=0.0531 />
            <models>
                <model file="sphere.3d" R=0.54 G=0.49 B=0.51 />
            </models>
        </group>
        <!-- Earth -->
        <group>
            <translate X=1.1023444082205645 Z=2.243844202627458 />
            ↪ <!-- alpha: 0.4566432233109321 | radius: 2.5 -->
            <scale X=0.0555 Y=0.0555 Z=0.0555 />
            <models>
                <model file="sphere.3d" R=0 G=0.20 B=0.95 />
            </models>
            <!-- Moon -->
            <group>
```

```
            <translate X=-1.562802456801457 Y=0.3495855806969056
            ↪  Z=1.2820609824747564 /> <!-- alpha:
            ↪  5.399421988971284 | radius: 2.0514 -->
            <scale X=0.54 Y=0.54 Z=0.59 />
            <models>
                <model file="sphere.3d" R=0.8 G=0.8 B=0.8 />
            </models>
        </group>
    </group>
    <!-- Mars -->
    <group>
        <translate X=-1.8129991570166193 Z=2.8762882429716647 />
        ↪  <!-- alpha: -0.5624200571083329 | radius: 3.4 -->
        <scale X=0.0333 Y=0.0333 Z=0.0333 />
        <models>
            <model file="sphere.3d" R=0.95 G=0.5 B=0 />
        </models>
    </group>
    <!-- Jupiter -->
    <group>
        <translate X=-0.11115273553965137 Z=6.7990914885286 />
        ↪  <!-- alpha: -0.01634671852610481 | radius: 6.8 -->
        <scale X=0.6111 Y=0.6111 Z=0.6111 />
        <models>
            <model file="sphere.3d" R=0.89 G=0.85 B=0.80 />
        </models>
        <!-- Europa -->
        <group>
            <translate X=-1.9739134754677707
            ↪  Y=-0.6058151146019737 Z=-0.32106354244377094 />
            ↪  <!-- alpha: 4.551147722545702 | radius: 2.0896
            ↪  -->
            <scale X=0.0222 Y=0.0222 Z=0.0222 />
            <models>
                <model file="sphere.3d" R=0.8 G=0.8 B=0.8 />
            </models>
        </group>
        <!-- Io -->
        <group>
            <translate X=1.3009021831337488
            ↪  Y=-0.9117855918605499 Z=1.290616435814047 />
            ↪  <!-- alpha: 0.7893671463847127 | radius: 2.0468
            ↪  -->
            <scale X=0.03181 Y=0.03181 Z=0.03181 />
            <models>
                <model file="sphere.3d" R=0.8 G=0.8 B=0.8/>
            </models>
        </group>
        <!-- Ganymede -->
        <group>
```

```
            <translate X=-1.6360631702462545
            ↪  Y=0.18780201298969929 Z=1.3716172741988903 />
            ↪  <!-- alpha: 5.410089129623691 | radius: 2.1432
            ↪  -->
            <scale X=0.03704 Y=0.03704 Z=0.03704 />
            <models>
                <model file="sphere.3d" R=0.8 G=0.8 B=0.8 />
            </models>
        </group>
        <!-- Callisto -->
        <group>
            <translate X=-0.7902424269980806
            ↪  Y=0.4610331189744719 Z=-2.057125987823996 />
            ↪  <!-- alpha: 3.5083599438343995 | radius: 2.2514
            ↪  -->
            <scale X=0.03448 Y=0.03448 Z=0.03448 />
            <models>
                <model file="sphere.3d" R=0.8 G=0.8 B=0.8 />
            </models>
        </group>
    </group>
    <!-- Saturn -->
    <group>
        <translate X=3.9053847471424357 Z=13.027201156687005 />
        ↪  <!-- alpha: 0.2912613134099548 7 | radius: 13.6 -->
        <scale X=0.4888 Y=0.4888 Z=0.4888 />
        <models>
            <model file="sphere.3d" R=0.9294 G=0.8 B=0.6588 />
        </models>
        <!-- Titan -->
        <group>
            <translate X=-0.623182250994466
            ↪  Y=-0.44433399546510904 Z=2.0202819957915485 />
            ↪  <!-- alpha: -0.29920281941546634 | radius:
            ↪  2.1604 -->
            <scale X=0.04425 Y=0.04425 Z=0.04425 />
            <models>
                <model file="sphere.3d" R=0.4 G=0.4 B=0.4 />
            </models>
        </group>
        <group>
            <rotate angle="30" axisX="1" axisY="0" axisZ="0.5"
            ↪  />
            <scale X="1.2" Y="0.1" Z="1.2" />
            <models>
                <model file="torus.3d" R="0.803" G="0.702"
                ↪  B="0.503" A="0.5" />
            </models>
        </group>
    </group>
```

```
<!-- Uranus -->
<group>
    <translate X=-1.1387076500733375 Z=27.17615397527149 />
    ↪ <!-- alpha: -0.04187649015215089 | radius: 27.2
    ↪ -->
    <scale X=0.2052 Y=0.2052 Z=0.2052 />
    <models>
        <model file="sphere.3d" R=0.5882 G=0.7215 B=0.7647
        ↪ />
    </models>
    <!-- Miranda -->
    <group>
        <translate X=0.98943899321693 Y=-0.30295734745001557
        ↪ Z=0.02647147007475641 /> <!-- alpha:
        ↪ -4.739136619271513 | radius: 1.03512 -->
        <scale X=0.0037728 Y=0.0037728 Z=0.0037728 />
        <models>
            <model file="sphere.3d" R=0.8 G=0.8 B=0.8 />
        </models>
    </group>
    <!-- Ariel -->
    <group>
        <translate X=-0.06895170372620689
        ↪ Y=-0.24934606877077728 Z=-1.5061026479432975 />
        ↪ <!-- alpha: 3.187342251872174 | radius: 1.52816
        ↪ -->
        <scale X=0.0092624 Y=0.0092624 Z=0.0092624 />
        <models>
            <model file="sphere.3d" R=0.8 G=0.8 B=0.8 />
        </models>
    </group>
    <!-- Umbriel -->
    <group>
        <translate X=0.5658399564594316 Y=1.1761745075093901
        ↪ Z=-1.683758543128744 /> <!-- alpha:
        ↪ 2.8173922121456716 | radius: 2.1304 -->
        <scale X=0.0093552 Y=0.0093552 Z=0.0093552 />
        <models>
            <model file="sphere.3d" R=0.8 G=0.8 B=0.8 />
        </models>
    </group>
    <!-- Titania -->
    <group>
        <translate X=-1.423712709885508
        ↪ Y=-0.1104434509015812 Z=3.1815037580149763 />
        ↪ <!-- alpha: 5.8624150037967375 | radius:
        ↪ 3.48728 -->
        <scale X=0.0126144 Y=0.0126144 Z=0.0126144 />
        <models>
            <model file="sphere.3d" R=0.8 G=0.8 B=0.8 />
```

```
                </models>
            </group>
            <!-- Oberon -->
            <group>
                <translate X=0.15274979821996085
                ↪  Y=-0.6680450275971876 Z=4.617586071298124 />
                ↪  <!-- alpha: -6.2501173545660915 | radius:
                ↪  4.66816 -->
                <scale X=0.0121824 Y=0.0121824 Z=0.0121824 />
                <models>
                    <model file="sphere.3d" R=0.8 G=0.8 B=0.8 />
                </models>
            </group>
        </group>
        <!-- Neptune -->
        <group>
            <translate X=-14.758121400868891 Z=37.17792157608082 />
            ↪  <!-- alpha: -0.37788233116053316 | radius: 40 -->
            <scale X=0.1887 Y=0.1887 Z=0.1887 />
            <models>
                <model file="sphere.3d" R=0.2352 G=0.4627 B=0.6588
                ↪  />
            </models>
            <!-- Triton -->
            <group>
                <translate X=10.7182539421744 Y=-6.168509809381747
                ↪  Z=10.5726732268913 /> <!-- alpha:
                ↪  -5.4909495624328155 | radius: 16.27 -->
                <scale X=2.05e-05 Y=2.05e-05 Z=2.05e-05 />
                <models>
                    <model file="sphere.3d" R=0.8 G=0.8 B=0.8 />
                </models>
            </group>
        </group>
        <!-- Pluto -->
        <group>
            <translate X=24.639319031721364 Z=47.601091979628514 />
            ↪  <!-- alpha: 0.4776447342524617 | radius: 53.6 -->
            <scale X=0.01 Y=0.01 Z=0.01 />
            <models>
                <model file="sphere.3d" R=0.5607 G=0.5294 B=0.5058
                ↪  />
            </models>
        </group>
    </group>
</scene>
```