

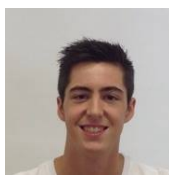
Universidade do Minho

Computação Gráfica

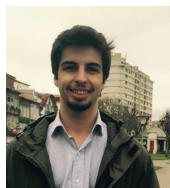
MIEI - 3º ANO - 2º SEMESTRE

UNIVERSIDADE DO MINHO

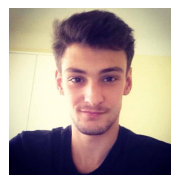
TRABALHO PRÁTICO - PARTE 2



Guilherme Guerreiro
A73860



Dinis Peixoto
A75353



Ricardo Pereira
A74185



Marcelo Lima
A75210

22 de Maio de 2017

Conteúdo

| | | |
|----------|--|-----------|
| 1 | Introdução | 2 |
| 1.1 | Contextualização | 2 |
| 1.2 | Resumo | 2 |
| 2 | Arquitetura do código | 3 |
| 2.1 | Aplicações | 3 |
| 2.1.1 | Gerador | 3 |
| 2.1.2 | Motor | 4 |
| 2.2 | Classes | 5 |
| 2.2.1 | Vértice/Ponto | 6 |
| 2.2.2 | Forma | 6 |
| 2.2.3 | Translação | 7 |
| 2.2.4 | Rotação | 7 |
| 2.2.5 | Escala | 8 |
| 2.2.6 | Cor | 9 |
| 2.2.7 | Grupo | 9 |
| 2.3 | Ficheiros auxiliares | 10 |
| 2.3.1 | Figures | 10 |
| 2.3.2 | Parser | 11 |
| 2.3.3 | tinyxml2 | 12 |
| 3 | Generator | 13 |
| 3.1 | Primitiva geométrica: <i>Torus</i> | 13 |
| 3.1.1 | Algoritmo | 13 |
| 4 | Engine | 16 |
| 4.1 | Processo de leitura | 16 |
| 4.2 | Estruturas de dados | 17 |
| 4.3 | Processo de renderização | 18 |
| 5 | Análise de Resultados - Sistema Solar | 19 |
| 5.1 | Visualização | 19 |
| 6 | Conclusão/Trabalho futuro | 23 |
| 7 | Anexos | 24 |
| 7.1 | Ficheiro de configuração - Sistema Solar | 24 |

1. *Introdução*

1.1 Contextualização

Foi-nos proposto, no âmbito da UC Computação Gráfica, a criação de um mini mecanismo 3D baseado num cenário gráfico sendo que para isso teríamos de utilizar várias ferramentas apresentadas nas aulas práticas entre as quais C++ e OpenGL.

Este trabalho foi dividido em quatro partes, sendo esta a segunda fase que tem como objetivo a criação de cenários hierárquicos usando transformações geométricas tendo como finalidade a criação de 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, como é óbvio, 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 de Saturno, além da implementação da funcionalidade Cor, que irá acompanhar as restantes transformações gráficas e ainda de uma interação na 3ª pessoa com a aplicação permitindo o utilizador navegar livremente pelo cenário criado.

2. *Arquitetura do código*

Tendo em mente a continuação do trabalho desenvolvido na fase anterior, mantemos as duas aplicações principais previamente desenvolvidas **gerador** e **engine**, sendo 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 indiscutivelmente necessário alterar a forma como o engine processa esses mesmos ficheiros.

2.1.1 Gerador

generator.cpp - Tal como explicado na fase anterior, esta é a aplicação onde estão definidas as estruturas 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 Gerador novas funcionalidades que lhe permitissem gerar esta mesma primitiva. Tudo o resto se manteve idêntico ao previamente desenvolvido na fase anterior.

```

#----- HELP -----#
|
| Usage: ./generator {COMMAND} ... {FILE}
|           [-h]
|
| COMMANDS:
| - plane [SIZE]
|     Creates a square in the XZ plane, centred in the origin.
|
| - box [SIZE X] [SIZE Y] [SIZE Z] [DIVISIONS]
|     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 [INNER RADIUS] [OUTER RADIUS] [SIDES] [RINGS]
|     Creates a torus with the inner and outer radius, sides
|     and rings given.
|
| 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 - Tal como anteriormente, esta é a aplicação que possui as funcionalidades principais. Permite a apresentação de uma janela exibindo os modelos pretendidos e ainda a interação com estes através de diversos comandos. Com alteração na estrutura do ficheiro XML, foi necessário alterar o método que está por trás do parsing, o qual será explicado mais adiante. Consequentemente, como passarão a existir grupos de primitivas com informações associadas, obviamente que é necessário armazenar a informação de maneira diferente, sendo esta renderizada pelo GLUT também de uma forma distinta da fase anterior.

```
#----- HELP -----#
|
| Usage: ./engine {XML FILE}
|         [-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
|
| - r : Reset the camera to the initial position
|
| FORMAT:
| - p: Change the figure format into points
|
| - l: Change the figure format into lines
|
| - o: Fill up the figure
|
#-----#
```

Figura 2.2: Menu de ajuda do Engine.

2.2 Classes

Para além das classes anteriormente criadas **Vertex** e **Shape**, o grupo decidiu criar 5 novas classes. Surge assim uma classe para cada transformação geométrica (**Translation**, **Rotation** e **Scale**) que armazenam a informação das mesmas, a classe **Colour**, e por último, a classe **Group**, que irá armazenar um conjunto de Formas, associando-as às respectivas transformações geométricas.

Todas as classes apresentadas de seguida possuem:

- Variáveis de instância;
- Construtores;
- Getters;

- Setters.

2.2.1 Vértice/Ponto

Vertex.h - Classe que guarda um ponto necessário para a constituição de um triângulo, através da definição das suas coordenadas (x,y,z).

```
#ifndef __VERTEX_H__
#define __VERTEX_H__
#include <string>
using namespace std;

class Vertex{

    float x;
    float y;
    float z;

public:
    Vertex();
    Vertex(float,float,float);
    float getX();
    float getY();
    float getZ();
    string print();
    virtual ~Vertex(void);
};

#endif
```

Figura 2.3: Apresentação do ficheiro Vertex.h.

2.2.2 Forma

Shape.h - Classe que guarda todo o conjunto de pontos necessários à representação de um determinado modelo, contendo, desta maneira, um *vector<Vertex*>*, ou seja, um **conjunto de vértices** (*Vertex*).

```

#ifndef __SHAPE_H__
#define __SHAPE_H__
#include <string>
#include <vector>
#include "Vertex.h"

using namespace std;

class Shape{

    string name;
    vector<Vertex*> vertex_list;

public:
    Shape();
    Shape(string,vector<Vertex*>);
    string getName();
    vector<Vertex*> getVertexList();
    virtual ~Shape();
};

#endif

```

Figura 2.4: Apresentação do ficheiro Shape.h.

2.2.3 Translação

Translation.h - Classe que armazena toda a informação necessária à execução de uma translação, sendo portanto obrigatória a existência das variáveis de instância x , y e z , representando o **vector aplicado na translação**.

```

#ifndef __TRANSLATION_H__
#define __TRANSLATION_H__

class Translation{

    float x;
    float y;
    float z;

public:
    Translation();
    Translation(float,float,float);
    float getX();
    float getY();
    float getZ();
    void setX(float);
    void setY(float);
    void setZ(float);
    Translation* clone() const;
    virtual ~Translation();
};

#endif

```

Figura 2.5: Apresentação do ficheiro Translation.h.

2.2.4 Rotação

Rotation.h - Classe que armazena toda a informação pertinente à execução de uma rotação, sendo, assim, necessária a existência das mesmas variáveis que na

situação anterior (x , y e z), representando o **vector de aplicação** e ainda uma variável **ângulo** correspondendo ao ângulo de rotação sobre o vector.

```
#ifndef __ROTATION_H__
#define __ROTATION_H__

class Rotation{

    float angle;
    float x, y, z;

public:
    Rotation();
    Rotation(float, float, float, float);
    float getAngle();
    float getX();
    float getY();
    float getZ();
    void setAngle(float);
    void setX(float);
    void setY(float);
    void setZ(float);
    Rotation* clone() const;
    virtual ~Rotation();
};

#endif
```

Figura 2.6: Apresentação do ficheiro Rotation.h.

2.2.5 Escala

Scale.h - Classe que armazena toda a informação necessária à execução de um redimensionamento, como tal, são necessárias três variáveis representativas das **dimensões** (em relação às originais) sobre cada um dos diferentes eixos.

```
#ifndef __SCALE_H__
#define __SCALE_H__

class Scale{

    float x;
    float y;
    float z;

public:
    Scale();
    Scale(float, float, float);
    float getX();
    float getY();
    float getZ();
    void setX(float);
    void setY(float);
    void setZ(float);
    Scale* clone() const;
    virtual ~Scale();
};

#endif
```

Figura 2.7: Apresentação do ficheiro Scale.h.

2.2.6 Cor

Colour.h - Classe que armazena informação relevante para a alteração de cor durante a renderização das figuras pretendidas. Uma vez que a cor é processada segundo o **modelo RGB** são necessárias três variáveis de instância, uma para cada cor (*Red*, *Green* e *Blue*).

```
#ifndef __COLOUR_H__
#define __COLOUR_H__

class Colour{

    float r;
    float g;
    float b;

public:
    Colour();
    Colour(float, float, float);
    float getR();
    float getG();
    float getB();
    void setR(float);
    void setG(float);
    void setB(float);
    Colour* clone();
    virtual ~Colour();

};

#endif
```

Figura 2.8: Apresentação do ficheiro Colour.h.

2.2.7 Grupo

Group.h - Classe cuja função é armazenar toda a informação correspondente a um determinado grupo. Esta será utilizada aquando leitura dos ficheiros input, em *XML*, na medida que, a cada grupo lido e interpretado, corresponderá um objeto *Group* com informação relativa ao seu **identificador** (*id*), às **formas/modelos** incluídos (*group_shapes*), às transformações relativas ao mesmo, desde **translações** (*translation*), **rotações** (*rotation*), **redimensionamentos** (*scale*) e ainda às mudanças de **cor** (*colour*). Por fim apresentará também a lista dos grupos contidos neste, denominados por **grupos-filho** (*group_childs*).

```

#ifndef __GROUP_H__
#define __GROUP_H__

#include <string>
#include <vector>
#include "Shape.h"
#include "Translation.h"
#include "Rotation.h"
#include "Scale.h"
#include "Colour.h"

using namespace std;

class Group{
    int id;
    vector<Shape*> group_shapes;
    vector<Group*> group_childs;
    Translation* translation;
    Rotation* rotation;
    Scale* scale;
    Colour* colour;

public:
    Group();
    Group(int);
    Group(vector<Shape*>, vector<Group*>, Translation*, Rotation*, Scale*, Colour*);
    int getID();
    vector<Shape*> getShapes();
    vector<Group*> getChilds();
    Translation* getTranslation();
    Rotation* getRotation();
    Scale* getScale();
    Colour* getColour();
    void setShapes(vector<Shape*>);
    void setChilds(vector<Group*>);
    void setTranslation(Translation*);
    void setRotation(Rotation*);
    void setScale(Scale*);
    void setColour(Colour*);
    void addChild(Group*);
    virtual ~Group();
};

#endif

```

Figura 2.9: Apresentação do ficheiro Group.h.

2.3 Ficheiros auxiliares

2.3.1 Figures

É neste ficheiro, **Figures.h**, que se encontram todos os métodos de criação de formas, isto é, todos os métodos utilizados pelo *generator* durante a criação de uma dada figura, das disponíveis pelo mesmo (plano, paralelepípedo, cone, esfera e ainda, cilindro).

```

#ifndef __FIGURES_H__
#define __FIGURES_H__
#include <string>
using namespace std;

#define _USE_MATH_DEFINES
#include <math.h>
#include "Vertex.h"
#include <vector>
using namespace std;

vector<Vertex*> createPlane(float size);
vector<Vertex*> createBox(float x, float y, float z, int div);
vector<Vertex*> createCone(float radius, float height, int slice, int stack);
vector<Vertex*> createSphere(float radius, int slice, int stack);
vector<Vertex*> createCylinder(float radius, float height, int slice, int stack);

#endif

```

Figura 2.10: Apresentação do ficheiro Figures.h.

2.3.2 Parser

O ficheiro **Parser.h** contém todos os métodos imprescindíveis ao parsing dos ficheiros que servem de input à aplicação *engine*. Através de um algoritmo, que será explicado adiante, a conjunção destes métodos com a ferramenta *tinyxml* é capaz de ler e interpretar qualquer ficheiro *XML* de modo a armazenar a informação relevante, presente no mesmo, em estruturas apropriadas, contendo referências aos ficheiros modelo e às transformações aplicadas nestes.

```

#ifndef __PARSER_H__
#define __PARSER_H__

#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <sstream>
#include "tinyxml2.h"
#include "Shape.h"
#include "Group.h"
#include "Vertex.h"

using namespace std;
using namespace tinyxml2;

void updateTranslation(XMLElement*, Group*);
void updateRotation(XMLElement*, Group* );
void updateScale(XMLElement*, Group* );
void exploreElement(XMLElement*, Group*);
vector<Shape*> exploreModels(XMLElement* );
vector<Vertex*> readFile(string);
Group* hereditaryChild(Group*);
Group* parseXML(char*);

#endif

```

Figura 2.11: Apresentação do ficheiro Parser.h.

2.3.3 `tinyxml2`

`tinyxml2.h` - Ferramenta utilizada para auxiliar no parsing dos ficheiros *XML* de modo a explorar o seu conteúdo.

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 6 primitivas geométricas que o gerador está apto a gerar.

3.1 Primitiva geométrica: *Torus*

Um *Torus* é um sólido geométrico que apresenta o formato aproximado de uma câmara de pneu. 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), **R** (raio exterior), **nrings** (número de divisões radiais) e **nsides** (número de lados por cada secção radial).

3.1.1 Algoritmo

Para a construção do *Torus* é preciso considerar que a sua constituição baseia-se nos raios interior e exterior. Para tal, é preciso definir que eixos vão ficar responsáveis para definir as circunferências que vamos percorrer para poder desenhar o *Torus*. Com isto, os eixos *Y* e *X* definem uma circunferência com o raio exterior **R**, e os eixos *X*, *Y* e *Z* definem uma circunferência com o raio interior **r**.

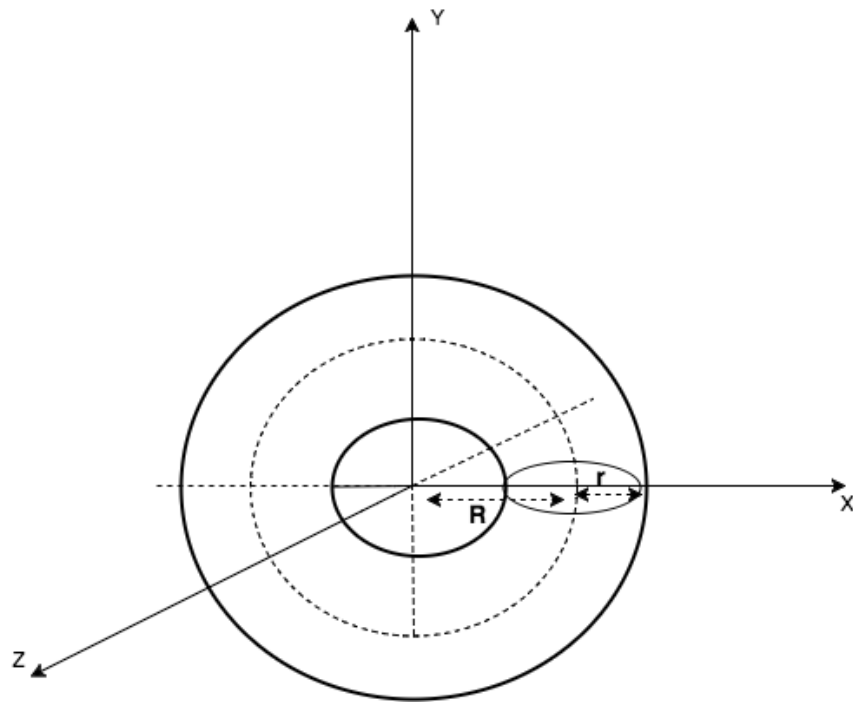


Figura 3.1: Ilustração do formato do *Torus* (vista por cima).

Para uma melhor visualização do método de construção do *Torus*, fizemos uma rotação nos eixos, obtendo a seguinte ilustração:

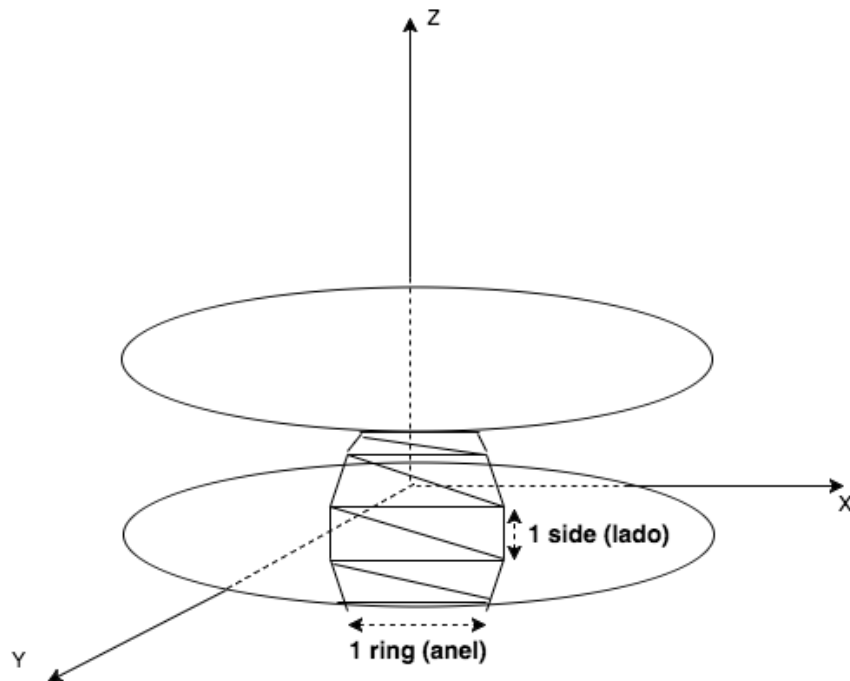


Figura 3.2: Ilustração da construção do *Torus* (vista em frente)

Para podermos iterar através das circunferências definidas, temos que recorrer aos parâmetros rings e sides (que dividem as respectivas circunferências em várias partes), onde cada porção é definida pelo uma amplitude dada por :

$$\begin{aligned}\text{dimSide} &= (2*M_PI)/\text{sides} \\ \text{dimRing} &= (2*M_PI)/\text{rings}\end{aligned}$$

Desta forma, com auxilio das funções *cos* e *sin*, podemos obter facilmente os pontos que formem as circunferências. O desenho inicia-se ($i=0$) por definir os pontos:

$$\begin{aligned}x0 & (\cos(i*\text{dimRing})) \\ y0 & (\sin(i*\text{dimRing})) \\ x1 & (\cos((i+1)*\text{dimRing})) \\ y1 & (\sin((i+1)*\text{dimRing}))\end{aligned}$$

Os pontos $x0$ e $y0$ representam os pontos de referência de amplitude e $x1$ e $y1$ os próximos pontos (próximo anel ao incrementar a amplitude), em relação à circunferência do raio externo. Desta forma, passamos a desenhar entre estes dois limitadores um anel, como representado na figura acima. Para tal, do mesmo modo que percorremos a circunferência externa, percorremos a circunferência interna, isto é, adicionamos dimSide em cada interação, para dar a volta à circunferência interna. Para definir os pontos reais, basta obter os seguinte valores:

$$\begin{aligned}r &= \text{radiusIn} * \cos(j*\text{dimSide}) + \text{radiusOut} \\ z &= \text{radiusIn} * \sin(j*\text{dimSide})\end{aligned}$$

O valor de r é o afastamento em relação ao centro, isto é, factor que afasta os pontos do centro, adicionando o raio externo, para que todos os pontos estejam para lá desse valor, e o raio interno multiplicado por $\cos(j*\text{dimSide})$, para definir o ponto da circunferência externa.

O mesmo acontece com o valor z , para as coordenadas do eixo Z , que apenas precisam do valor do raio interno multiplicado por $\sin(j*\text{dimSide})$, porque o desvio do raio externo só se aplica aos eixos X e Y .

Por fim, basta multiplicar o valor r (factor) pelos pontos $x0, y0, x1$ e $y1$, para obtermos as coordenadas dos pontos no *Torus*. Tendo os pontos, já definidos, basta formar os triângulos, seleccionando três pontos no sentido contrário ao relógio. Com isto, as iterações baseiam-se em cada anel (conjunto de 2 limitadores), iterar (adicionar dimSide), e aplicar o mesmo processo de desenho.

Terminando a circunferência interna, itera-se a amplitude da circunferência externa (adicionar dimRing), e repetir o mesmo processo até completar a circunferência externa toda.

4. *Engine*

O motor (ou *engine*) é responsável por receber ficheiros de configuração escritos em XML. Na primeira fase, o funcionamento deste era simplesmente reconhecer e apresentar o conteúdo dos ficheiros modelo presentes neste ficheiro de configuração. Na segunda fase foram feitas algumas alterações, sendo agora possível renderizar tanto o conteúdo dos ficheiros modelo como as respectivas transformações geométricas associadas a estes, apresentado-as no fim como um cenário ao utilizador.

4.1 Processo de leitura

Todo o processo de leitura efetuado pelo engine encontra-se, tal como anteriormente foi referido, no ficheiro auxiliar **Parser.h**. Este processo é iniciado quando fornecido um ficheiro XML como input à própria aplicação engine.

Posteriormente, este é explorado recursivamente através da função *exploreElement(XMLElement*, Group*)*, que recebe, a cada chamada, o elemento do XML que se encontra a explorar e o grupo onde toda a informação recolhida será armazenada. Explicaremos então, mais detalhadamente todo o processo recursivo efetuado nesta mesma função.

Na primeira chamada desta é-lhe atribuído como input o primeiro filho do elemento *scene* no ficheiro XML, que idealmente será um grupo, além deste será também atribuído o grupo inicial, representando todo o conteúdo do elemento *scene*.

Prosseguindo com o percurso da *exploreElement* será testado se o elemento a interpretar corresponde a um dos parâmetros relacionados com transformações, em caso positivo, será encaminhado para a função respetiva (*updateTranslation*, *updateRotation*, *updateScale* ou *updateColour*), cujo objetivo será guardar a informação contida nestes elementos na estrutura do grupo que estamos de momento a explorar.

Caso o elemento atual não corresponda a uma transformação resta apenas verificar se é uma lista de modelos ou um grupo-filho. No primeiro caso é chamada a função *exploreModels* que irá ler todos os ficheiros modelos, fazendo para cada, através da função *readFile*, uma forma *Shape* cujo conteúdo serão os pontos que constituem o ficheiro modelo lido. Esta *Shape* é posteriormente adicionada à lista de formas do grupo atual, mantendo assim a informação deste sempre atualizada.

No segundo caso, ao encontrarmos um grupo-filho, devemos criar um novo objeto *Group*, adicionando-o à lista de grupos-filho do grupo atual, de modo a criar a hierarquia pretendida, chamando de seguida a função *exploreElements* recursivamente e fornecendo-lhe como input o grupo-filho recém-criado.

Se por alguma exceção o elemento a explorar não corresponder a nenhum destes, o algoritmo prossegue para o elemento-irmão ignorando o respectivo elemento.

4.2 Estruturas de dados

Através do algoritmo descrito na seção anterior facilmente concluímos qual será a estrutura de dados necessária para armazenar toda a informação recolhida durante o *parsing*. Desta forma, optamos por identificar cada grupo através de um **ID** sendo, para isso, necessária a existência de uma variável global na aplicação capaz de contabilizar o total de grupos até ao momento criados.

Foi também necessário criar uma lista (*vector*) de formas/modelos - **group_shapes** - sendo que cada um destas conterá os pontos disponíveis para construir as figuras desejadas.

Tendo em conta a importância da hierarquia no ficheiro de configuração, foi indispensável a criação de uma lista de grupos-filho, capazes de herdar as transformações aplicadas ao grupo-pai, daí a existência do **group_childs**.

Por fim, e para dar como concluída a estrutura de dados de um grupo, restava apenas associar os quatro objetos representantes de cada uma das quatro transformações disponíveis: translação (**translation**), rotação (**rotation**), redimensionamento (**scale**) e, por último, a mudança de cor (**colour**).

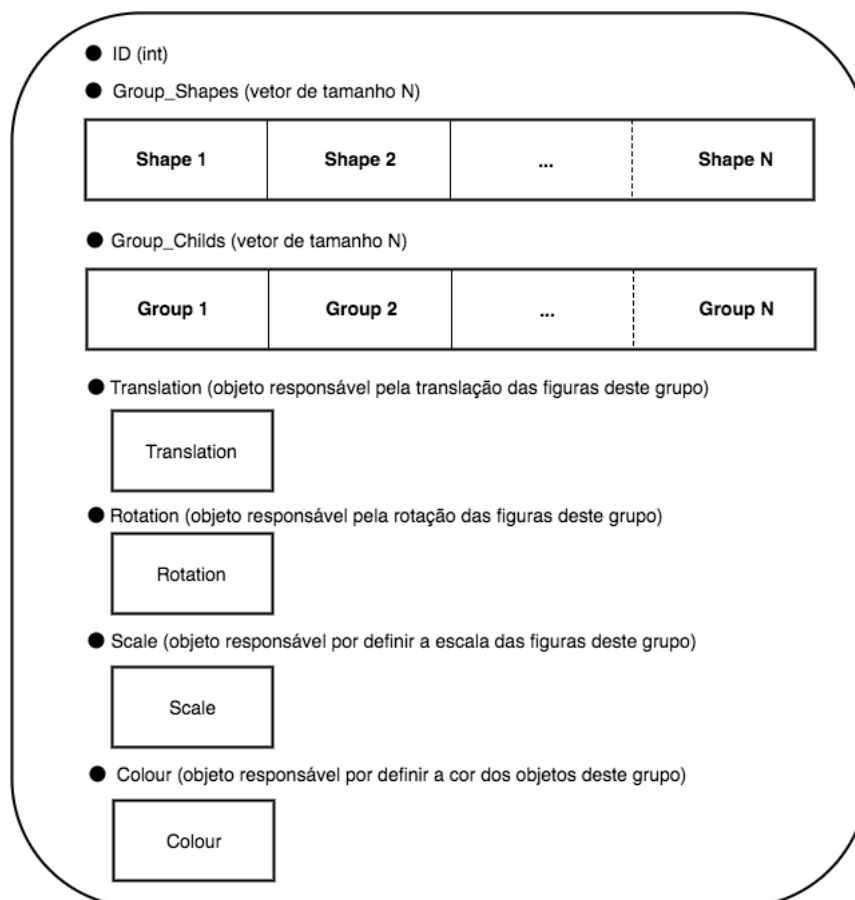


Figura 4.1: Representação da estruturação de dados de cada grupo.

4.3 Processo de renderização

Tal como vimos na secção anterior, o primeiro grupo criado aquando o parsing do ficheiro de configuração corresponde ao elemento *scene* do XML e portanto, onde se encontram todos os restantes grupos cuja informação será renderizada. Assim, este grupo funciona como variável global de toda a aplicação uma vez que através deste conseguimos alcançar todos os restantes grupos, que são nada mais que os seus grupos-filho.

A função responsável pela renderização do conteúdo tem o nome *renderScene*, esta é idêntica à apresentada na 1ª fase com uma simples alteração. Agora, ao invés de percorrer todas as formas armazenadas globalmente, esta chama a função recursiva *renderGroup*, que é a chave de todo este processo de renderização.

A função *renderGroup* recebe um único argumento do tipo *Group** e facilmente percebemos que na sua primeira chamada este argumento corresponde à variável global *scene*. Uma vez que serão efetuadas transformações geométricas, ou seja, uma vez que a matriz de transformação será alterada, deve-se primeiramente guardar o estado inicial desta e, logo após as alterações pretendidas este estado deve ser reposto, daí a utilização dos métodos *glPushMatriz()* e *glPopMatrix()*, no início e no fim da função, respectivamente.

De seguida, e antes de começar a *desenhar*, é necessário aplicar as transformações previamente recolhidas para o grupo em questão, por isso são verificadas quais as transformações contidas e em caso positivo são realizadas, segundo os parâmetros dados. É de notar que, no caso da mudança de cor, que o grupo optou por fazer como extra para uma aproximação mais realista dos planetas do Sistema Solar, as cores seguem o modelo RGB com, para cada uma das três cores, valores de 0 a 255, estes valores são convertidos para valores de 0 a 1 de modo a ser possível aplicar a função *glColor3f*.

Efetuada as transformações resta percorrer todas as formas/modelos incluídos no grupo em questão e, para cada um destes, desenhar sequencialmente os pontos de modo a conseguir triângulos capazes de demonstrar a figura inicialmente pretendida.

Por fim a função é chamada recursivamente para cada um dos grupos-filho do atual, renderizando assim toda a informação contida no ficheiro de configuração.

5. *Análise de Resultados - Sistema Solar*

O resultado final correspondeu ao esperado pelo grupo, todos os planetas foram representados à escala de modo a criar uma percepção semelhante à realidade. As suas cores foram também alteradas e foram ainda incluídos alguns satélites naturais sempre com o intuito de aproximar o resultado final ao Sistema Solar real.

Houve ainda algumas preocupações quanto à disposição dos planetas, estes não foram colados em linha reta, criando a sensação do movimento translacional a que estão sujeitos.

5.1 Visualização

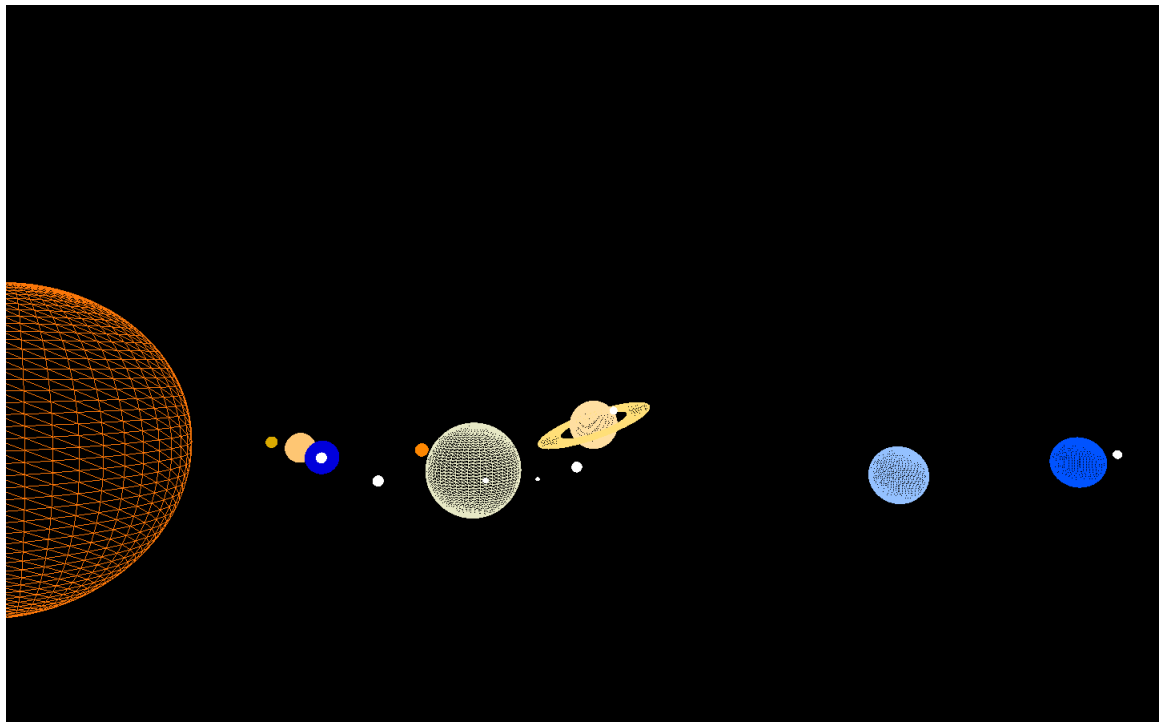


Figura 5.1: Visualização do Sistema Solar completo.

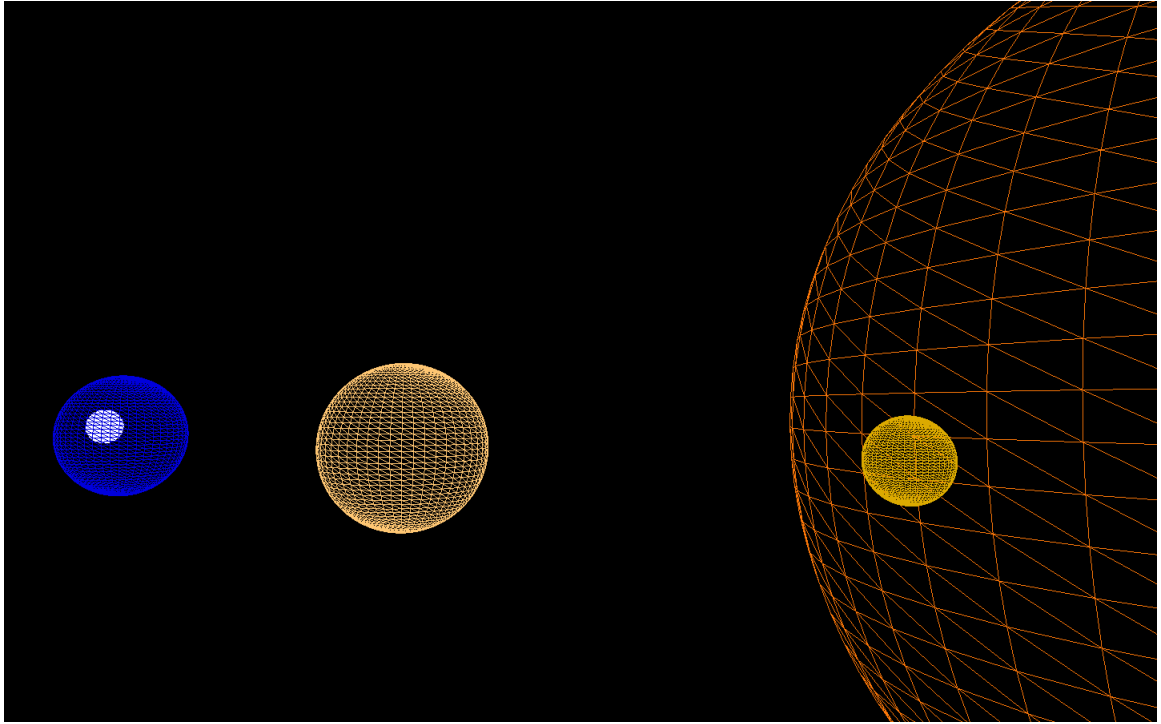


Figura 5.2: Visualização dos três primeiros planetas.

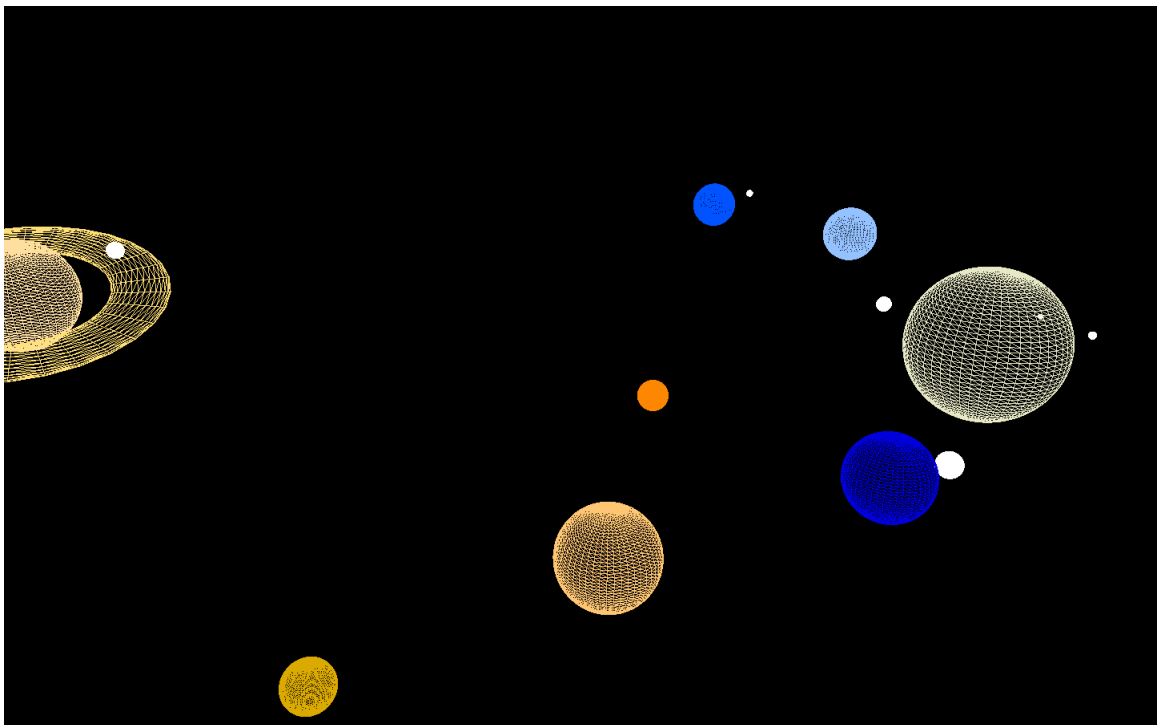


Figura 5.3: Visualização do Sistema Solar visto do Sol.

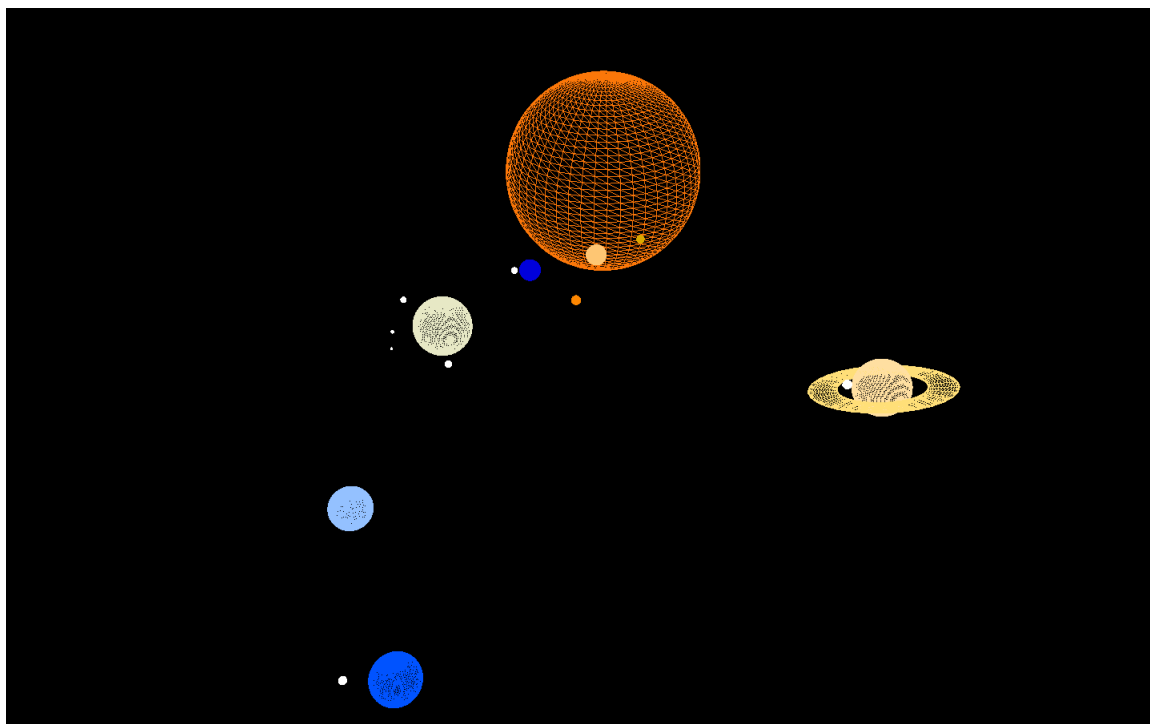


Figura 5.4: Visualização do Sistema Solar visto de Plutão.

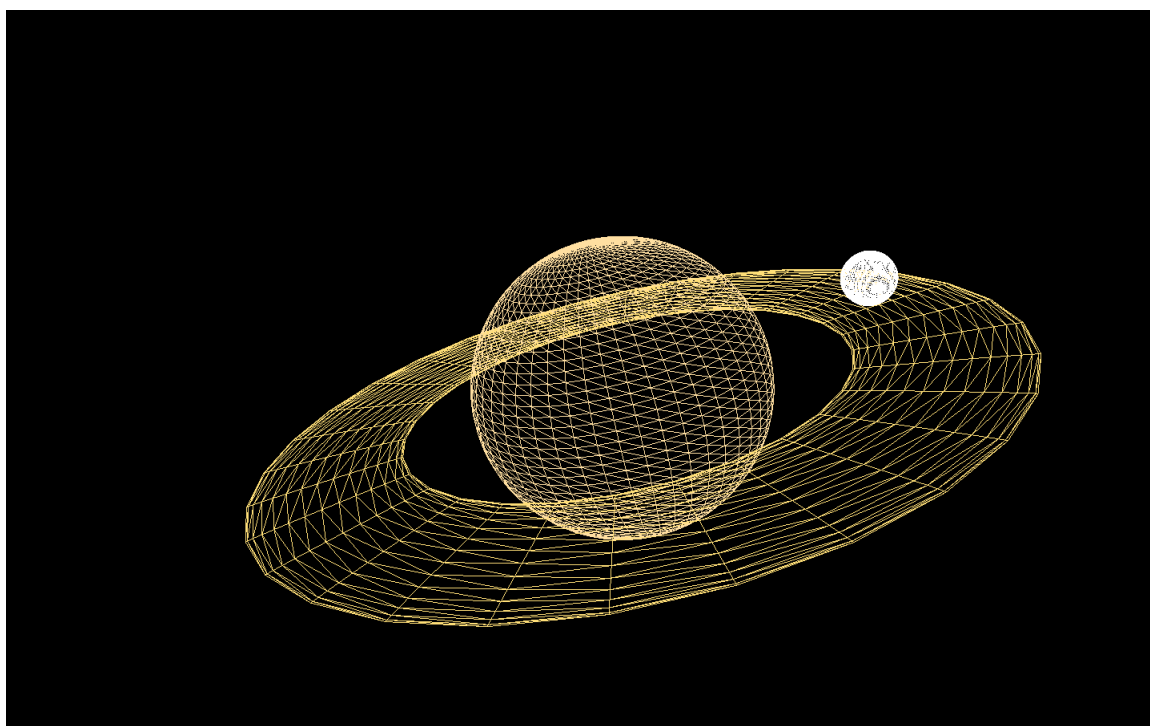


Figura 5.5: Visualização de Saturno por linhas (Comando L).

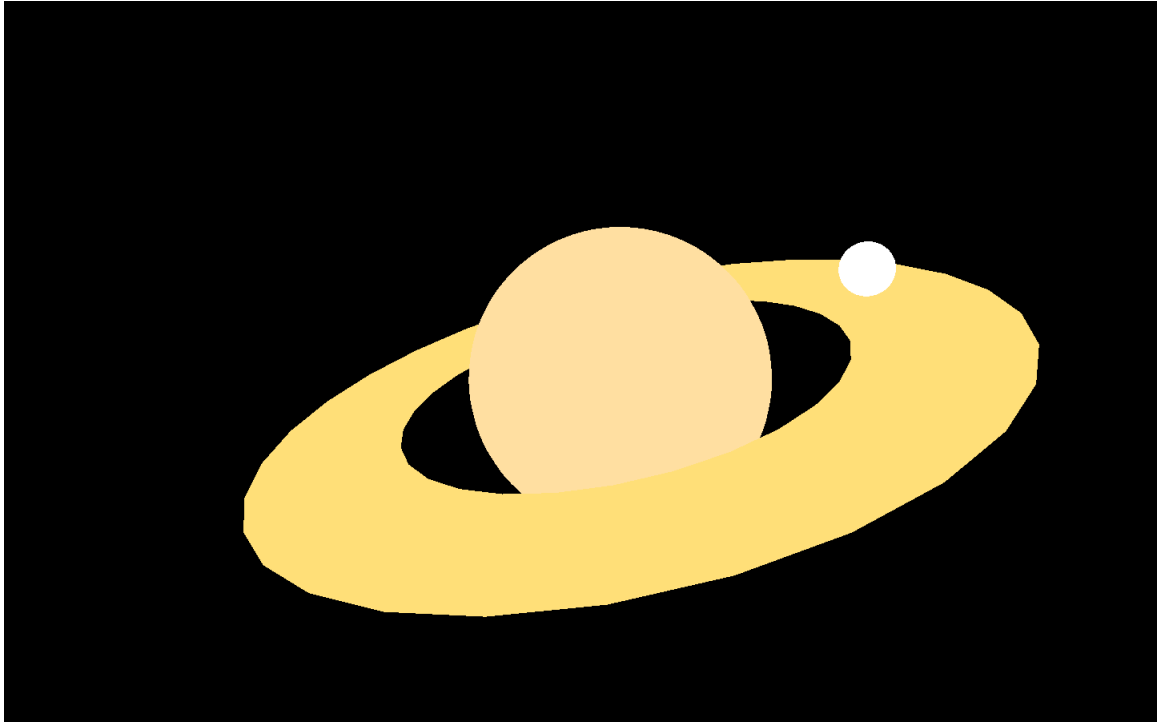


Figura 5.6: Visualização de Saturno a preenchido (Comando O).

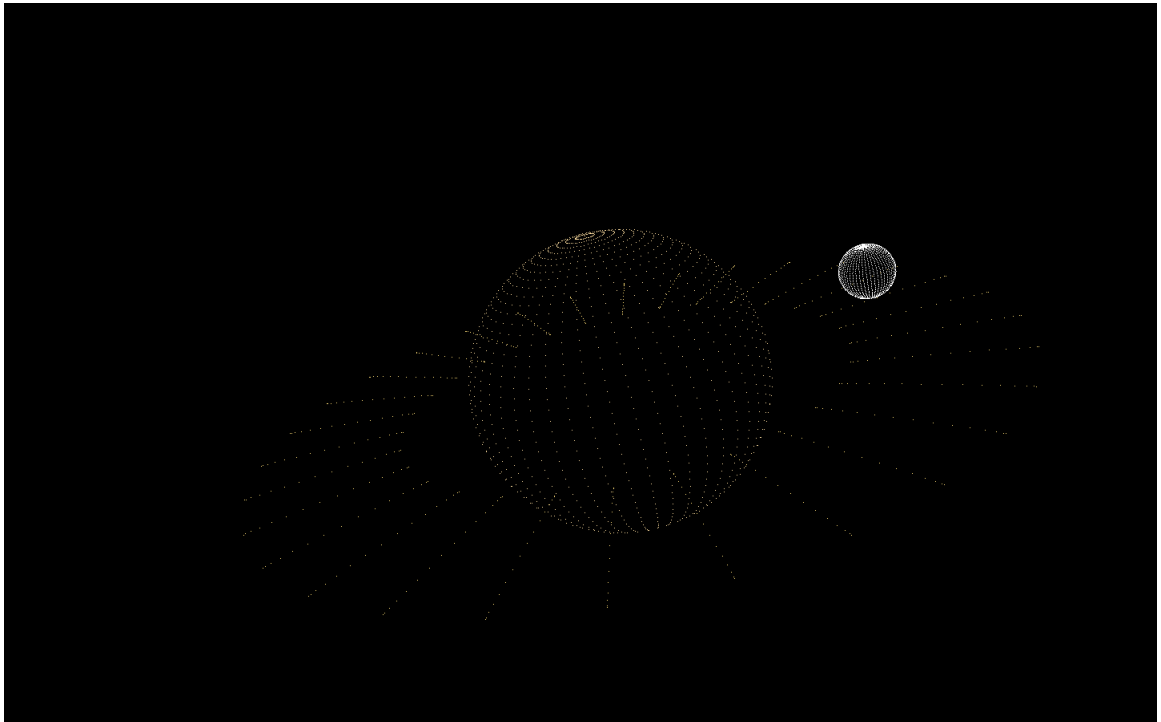


Figura 5.7: Visualização de Saturno por pontos (Comando P).

6. *Conclusão/Trabalho futuro*

A elaboração desta segunda fase do trabalho foi um pouco menos demorada e trabalhosa em relação à primeira fase. Isto deve-se, não só ao facto de os requisitos necessários serem relativamente menores, mas também às bases que fomos desenvolvendo durante a execução da primeira fase e que, de certa forma, nos ajudaram a ultrapassar algumas dificuldades que nos foram surgindo, visto já não estarmos tão inexperientes na matéria.

Pensamos que o resultado final desta fase corresponde às expectativas, na medida em que o modelo do Sistema Solar desenvolvido se enquadra perfeitamente naquilo que realmente era esperado.

No entanto, esperamos que nas restantes fases do projeto consigamos melhorar cada vez mais o modelo em questão, de forma a torna-lo o mais realista e agradável possível.

7. *Anexos*

7.1 Ficheiro de configuração - Sistema Solar

```
<scene>
  <group>

    <!--SOL-->
    <group>
      <colour R="251" G="119" B="9" />
      <scale X="10" Y="10" Z="10" />
      <models>
        <model file="sphere.3d" />
      </models>
    </group>

    <!--MERCURIO-->
    <group>
      <translate X="29.8775" Y="0" Z="0" />
      <colour R="219" G="170" B="0" />
      <scale X="0.3525" Y="0.3525" Z="0.3525" />
      <models>
        <model file="sphere.3d" />
      </models>
    </group>

    <!--VENUS-->
    <group>
      <translate X="33.9233" Y="0" Z="9.0897" />
      <colour R="254" G="198" B="115" />
      <scale X="0.8790" Y="0.8790" Z="0.8790" />
      <models>
        <model file="sphere.3d" />
      </models>
    </group>

    <!--TERRA-->
    <group>
      <translate X="36.8061" Y="0" Z="21.25" />
```

```

    <colour R="0" G="0" B="220" />
    <scale X="0.8865" Y="0.8865" Z="0.8865" />
    <models>
      <model file="sphere.3d" />
    </models>
  </group>
  <group>
    <translate X="0" Y="0.25" Z="3"/>
    <colour R="255" G="255" B="255" />
    <scale X="0.30" Y="0.30" Z="0.30" />
    <models>
      <model file="sphere.3d"/>
    </models>
  </group>
</group>

<!--MARTE-->
<group>
  <translate X="47.9523" Y="0" Z="15.5806" />
  <colour R="254" G="135" B="1" />
  <scale X="0.3720" Y="0.3720" Z="0.3720" />
  <models>
    <model file="sphere.3d" />
  </models>
</group>

<!--JUPITER-->
<group>
  <translate X="51.12987" Y="0" Z="37.1480" />
  <colour R="231" G="232" B="197" />
  <scale X="2.25" Y="2.25" Z="2.25" />
  <models>
    <model file="sphere.3d" />
  </models>

  <!--SATELITE N.-->
  <group>
    <translate X="0" Y="0" Z="3.375" />
    <colour R="255" G="255" B="255" />
    <scale X="0.05" Y="0.05" Z="0.05" />
    <models>
      <model file="sphere.3d" />
    </models>
  </group>

  <!--SATELITE N.-->
  <group>
    <translate X="2.025" Y="0" Z="3.5074" />

```

```

        <colour R="255" G="255" B="255" />
        <scale X="0.035" Y="0.035" Z="0.035" />
        <models>
            <model file="sphere.3d" />
        </models>
    </group>

    <!--SATELITE N.-->
    <group>
        <translate X="-4.025" Y="0" Z="2.4646" />
        <colour R="255" G="255" B="255" />
        <scale X="0.1" Y="0.1" Z="0.1" />
        <models>
            <model file="sphere.3d" />
        </models>
    </group>

    <!--SATELITE N.-->
    <group>
        <translate X="4.525" Y="0" Z="0" />
        <colour R="255" G="255" B="255" />
        <scale X="0.11" Y="0.11" Z="0.11" />
        <models>
            <model file="sphere.3d" />
        </models>
    </group>
</group>

<!--SATRUNO-->
<group>
    <rotate angle="30" X="0" Y="1" Z="1"/>
    <translate X="79.5005" Y="0" Z="-21.3021" />
    <colour R="255" G="223" B="161" />
    <scale X="1.995" Y="1.995" Z="1.995" />
    <models>
        <model file="sphere.3d" />
    </models>

    <!--SATELITE N.-->
    <group>
        <translate X="0" Y="1.2" Z="3.375" />
        <colour R="255" G="255" B="255" />
        <scale X="0.15" Y="0.15" Z="0.15" />
        <models>
            <model file="sphere.3d" />
        </models>
    </group>

```

```

        <!--ANEL-->
        <group>
            <rotate angle="90" X="1" Y="0" Z="0"/>
            <colour R="255" G="223" B="120" />
            <scale Z="0.01"/>
            <models>
                <model file="torus.3d" />
            </models>
        </group>
    </group>

    <!--URANO-->
    <group>
        <translate X="88.0747" Y="0" Z="50.85" />
        <colour R="148" G="193" B="255" />
        <scale X="1.290" Y="1.290" Z="1.290" />
        <models>
            <model file="sphere.3d" />
        </models>
    </group>

    <!--NEPTUNO-->
    <group>
        <translate X="112.35529" Y="0" Z="46.53908" />
        <colour R="0" G="83" B="255" />
        <scale X="1.275" Y="1.275" Z="1.275" />
        <models>
            <model file="sphere.3d" />
        </models>

    <!--SATELITE N.-->
    <group>
        <translate X="0" Y="1" Z="3.75" />
        <colour R="255" G="255" B="255" />
        <scale X="0.15" Y="0.15" Z="0.15" />
        <models>
            <model file="sphere.3d" />
        </models>
    </group>
</group>
</group>
</scene>

```