UNIVERSIDADE DO MINHO
ESCOLA DE ENGENHARIA

Computação Gráfica

TRABALHO PRÁTICO - FASE II

Adriana Henriques Esteves Teixeira Meireles A82582
Ana Marta Santos Ribeiro A82474
Carla Isabel Novais da Cruz A80564
Jéssica Andreia Fernandes Lemos A82061

MIEI - 3º Ano - 2º Semestre

Braga, 25 de março de 2019

Índice

Índice **1**

1 Introdução **2**

2 Organização do Código **2**

2.1 Generator 2

2.2 Engine 6

2.2.1 Point 6

2.2.2 Shape 6

2.2.3 Transformation 7

2.2.4 Group 7

2.2.5 Camera 8

2.2.6 Movimentação com as setas 9

2.2.7 Movimentação com o rato 9

2.2.8 Parser 9

2.2.9 Representação do sistema solar 12

3 Demonstração **15**

3.1 Usabilidade 15

3.1.1 Menu do Sistema Solar 16

3.2 Sistema Solar 17

4 Conclusão **23**

1 Introdução

A segunda fase deste projeto consiste em acrescentar novas funcionalidades e modificações ao trabalho realizado anteriormente, de modo a que seja possível efetuar transformações geométricas tais como translações, rotações e alterações da escala bem como adicionar o Torus como uma nova primitiva permitindo a criação de anéis para alguns planetas, como será comprovado mais à frente.

Desta forma foi possível implementar um Sistema Solar englobando o Sol, os planetas e os respetivos satélites naturais, na ordem científica destes.

2 Organização do Código

Tendo em conta a implementação realizada na primeira fase do projeto, optamos por seguir a abordagem estabelecida anteriormente, pelo que continuamos a ter duas aplicações, a engine e a generator.

2.1 Generator

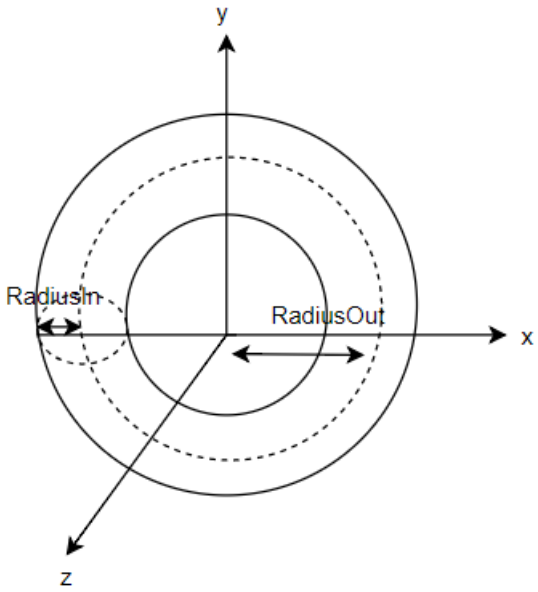
Tal como na fase anterior, este será responsável por gerar os pontos dos triângulos que permitem construir as diversas figuras geométricas. Para além das primitivas elaboradas na fase anterior, decidimos também construir a figura **Torus**. Assim, foi adicionada na classe **figures**, a seguinte primitiva:

- Torus (radiusIn, radiusOut, slices, layers)

Para a criação desta primitiva foram adicionadas funcionalidades ao Generator. Esta pode ser vista como um cilindro onde são unidas as duas bases.

Tal como aconteceu com o cilindro, foi usado um raio interno (radiusIn) para se definir a espessura do Torus. A lateral do cilindro pode ser dividida em camadas e fatias.

Com base neste raio interno, começa-se por gerar um anel e este será replicado até se completar o conjunto de fatias.



Como foi possível observar na imagem previamente apresentada, o radiusOut define a abertura do Torus, ou seja, a distância dos anéis do Torus à origem do referencial. Ainda consideramos os ângulos α e β que variam entre 0 e 2π de modo a abranger o Torus na sua totalidade, sendo o alpha utilizado para a construção das camadas e o beta para a criação dos anéis interiores. Para gerar os pontos necessários à criação deste foram realizados dois ciclos aninhados (como apresentado no excerto de código em baixo), sendo que o primeiro diz respeito às camadas e o outro corresponde à criação das fatias.

```
//...
for (int i = 0; i < layers; i++) {
    beta = i *( 2*M_PI / layers);
    nextBeta = (i + 1) * 2*M_PI / layers;

    for (int j = 0; j < slices; j++){
        alpha = j * 2 * M_PI / slices;
        nextAlpha = (j + 1) * 2 * M_PI / slices;
        //...
    }
}
```

Recorremos à função *pointsTorus* para gerar os pontos da figura. Para isto, foram utilizadas expressões matemáticas em que cada ponto é gerado da seguinte forma, em

função do beta e do alfa recebido.

```
Point pointsTorus(float radiusIn,float radiusOut,float beta,float
    alpha){

    Point result;

    result.x= cos(alpha) * (radiusIn * cos(beta) + radiusOut);
    result.y = sin(alpha) * (radiusIn*cos(beta) + radiusOut);
    result.z = radiusIn * sin(beta);

    return result;
}
```

As expressões matemáticas utilizadas anteriormente:

- x = cos α * (radiusIn * cosβ + radiusOut);
- y = sin α * (radiusIn * cosβ + radiusOut);
- z = radiusIn * sin(beta);

As expressões que calcula x e y ao serem multiplicadas por cos(alpha) e sin(alpha) possibilita a elaboração do exterior do Torus. A expressão do z não sofre um desvio do radiusOut pois o Torus é criado no plano xOy. Como foi explicado previamente, as outras expressões dizem respeito à criação das fatias.

Assim, à semelhança da esfera criada na fase anterior, são gerados 2 triângulos (4 pontos necessários) a cada iteração.

```
 for (int j = 0; j < slices; j++){
        //...
        p1 = pointsTorus(radiusIn, radiusOut, nextBeta, alpha);
        p2 = pointsTorus(radiusIn, radiusOut, beta, alpha);
        p3 = pointsTorus(radiusIn, radiusOut, nextBeta, nextAlpha);
        p4 = pointsTorus(radiusIn, radiusOut, beta, nextAlpha);
        //...
}
```

Foi assim gerado e apresentado o Torus da mesma forma que as outras figuras foram criadas na primeira fase do trabalho.

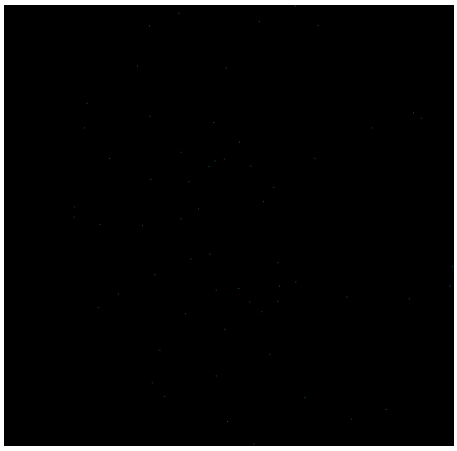


Figura 1. Modelo do Torus apresentado por pontos

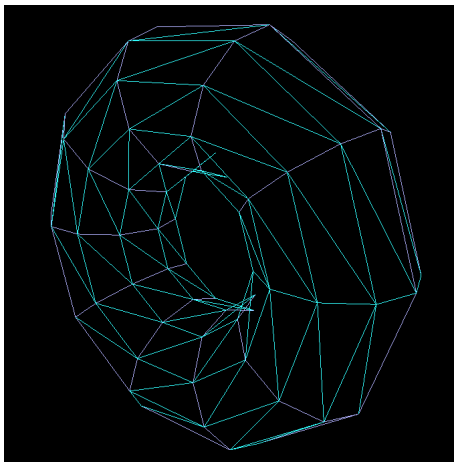


Figura 2. Modelo do Torus apresentado por linhas

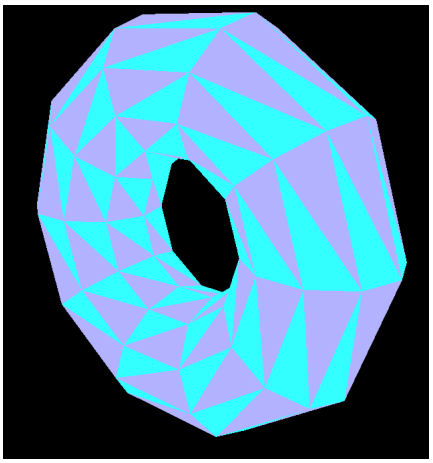


Figura 3. Modelo do Torus preenchido

2.2 Engine

De modo implementar as funcionalidades esperadas para esta fase, decidimos que deveríamos alterar a estruturação do código já elaborado. Desta forma, realizamos algumas classes que passaremos de seguida a apresentar.

2.2.1 Point

Na primeira fase do projeto optamos por criar uma estrutura Points, que continuamos a recorrer à sua utilização no Generator. Contudo consideramos que deveríamos criar uma classe para armazenar as coordenadas de um ponto.

```
class Point{
    private:
        float x;
        float y;
        float z;

    public:
        Point();
        Point(float,float,float);
        // gets
};
```

2.2.2 Shape

Uma vez que pretendíamos armazenar os pontos necessários para a construção das figuras geométricas, elaboramos esta classe.

```
class Shape {
    private:
        vector<Point*> points;

    public:
        Shape();
        Shape(vector<Point*> p);
        vector<Point*> getPoints();
};
```

2.2.3 Transformation

De modo a guardar todas as informações relativas a uma dada transformação, que poderá ser uma rotação, translação ou redimensionamento, elaboramos a classe da forma apresentada em baixo. Para tal tivemos em consideração que é necessário armazenar o vetor aplicado na transformação e, no caso da rotação, o ângulo. É importante referir que caso seja indicado no ficheiro xml uma cor para a figura, esta informação será tratada como uma transformação.

```
class Transformation {
    private:
        string type;
        float angle;
        float x;
        float y;
        float z;

    public:
        Transformation();
        Transformation(string typeT, float a, float xx, float yy,
            float zz);
        // gets
};
```

2.2.4 Group

Esta classe será responsável por armazenar toda a informação de um dado grupo, ou seja, todas as informações das transformações geométricas e das primitivas associadas pelo que é uma das classes mais importantes do projeto. Esta classe irá armazenar:

- Groups - Vetor com todos os grupos filhos
- Transformations - Vetor com todas as transformações geométricas do grupo
- Shapes - Vetor com os pontos para desenhar cada uma das formas geométricas

```
class Group {
    private:
        vector<Group*> groups;
        vector<Transformation*> transformations;
        vector<Shape*> shapes;

    public:
        Group();
        void addTransformation(Transformation* t);
        void addGroup(Group* g);
        void setShapes(vector<Shape*> pt);
        // gets
};
```

2.2.5 Camera

Nesta fase do projeto optamos por manter a funcionalidade de movimentar a câmera através das setas do teclado mas será acrescentada a possibilidade de movê-la através do rato ou recorrendo ao menu disponibilizado para focar em determinado planeta. Com o intuito de implementar tais funcionalidades criamos esta classe.

Inicialmente a câmera encontra-se direcionada para a origem do referencial e esta movimenta-se numa superfície esférica imaginária. Deste modo, esta classe contém as variáveis *positionX, positionY e positionZ* que indicam a posição x, y e z da câmera, respetivamente e a *radius* que corresponde ao raio da superfície. Existem também as variáveis *alpha e teta* que correspondem ao ângulo que define a posição da câmera no plano xOz e xOy respetivamente. De forma a implementar o menu que permite focar em determinado planeta foi necessário garantir que o movimento da câmera acompanha a posição para o qual esta está a olhar, pelo que definimos como variáveis *lookX, lookY e lookZ*, que indicam as posições para onde esta está virada. Assim e atendendo às coordenadas esféricas é possível definir a posição da câmera como:

positionX = lookX + radius*sin(alpha)*cos(teta)

positionY = lookY + radius*sin(teta)

positionZ = lookZ + radius*cos(alpha)*cos(teta)

2.2.6 Movimentação com as setas

De modo a implementar esta funcionalidade optamos por definir uma variável global *speed* que permite estabelecer a velocidade com que será possível movimentar a câmera. As setas para cima e para baixo permitem movimentá-la no sentido respetivo e para tal teremos de alterar o valor de teta, dado que como já foi referido corresponde à posição no plano xOz. Para movimentar a câmera para esquerda e direita será também através das setas e terá de ser o alfa a sofrer modificações. As teclas F1 e F2 permitem aproximar ou afastar a posição da câmera respetivamente e para tal será necessário alterar o valor do raio da superfície. É ainda possível colocar a câmera na posição inicial através da tecla F6.

2.2.7 Movimentação com o rato

Esta funcionalidade permite que o utilizador movimente a câmera, presionando o lado esquerdo do rato e mexendo-o. Assim definimos uma variável *mouseLeftIsPressed* que indica se o lado esquerdo do rato está a ser pressionado.

A função *mousePress* é responsável por armazenar as posições x e y do rato, quando o botão esquerdo é pressionado, e por somar aos ângulos da câmera a variação efetuada com o movimento deste. Isto garante que a câmera fica na posição desejada quando o botão é libertado. Foi então necessário definir como variáveis globais *mouseX e mouseY* que indicam as coordendadas x e y do rato.

Sempre que o rato é movimentado é necessário alterar os valores da posição da câmera. Para tal, antes de obtermos a sua posição final somamos aos ângulos a variação do movimento do rato. Neste processo é preciso garantir que o ângulo *teta* não ultrapassa os 90 e -90 graus.

2.2.8 Parser

Para esta fase foi necessário alterar a forma como efetuamos o parsing do *XML*, dado que nesta fase além de pretendermos obter os ficheiros .3d, que contêm os pontos que permitem a construção das figuras geométricas, teremos de ter em conta que estes contêm as informações relativas a transformações geométricas. Estas poderão estar encadeadas. Assim, apresentamos de seguida um exemplo de um ficheiro *XML*.

```
1 <scene>
```

```
2       <group>
3             <group>
4                   <!-- Sun -->
5                   <scale X="3.6" Y="3.6" Z="3.6" />
6                   <models>
7                         <model file="sphere.3d" />
8                   </models>
9             </group>
10            <group>
11                  <!-- Earth -->
12                  <translate X="-40" Z="10" />
13                  <scale X="0.5" Y="0.5" Z="0.5" />
14                  <models>
15                        <model file="sphere.3d" />
16                  </models>
17                  <group>
18                        <!-- Moon -->
19                        <scale X="0.3" Y="0.3" Z="0.3" />
20                        <translate X="13" Y="10" Z="13" />
21                        <models>
22                              <model file="sphere.3d" />
23                        </models>
24                  </group>
25            </group>
26            //...
27    </group>
28 </scene>
```

Assim sendo, tal como na primeira fase do projeto recorremos à API do *tinyxml2* para realizar o parsing dos ficheiros. No entanto, alteramos o modo de leitura destes que será explicado de seguida.

A primeira etapa deste processo consiste em carregar o ficheiro para a memória, o que no nosso caso é efetuado na função *loadXMLfile*. Esta é também responsável por invocar em caso de sucesso a *parseGroup* que processa todos os nodos. Para tal, é efetuado um ciclo que percorre todos os nodos e para cada um verifica o caso em que se encontra.

Assim, na eventualidade de este corresponder a uma transformação geométrica é invocada a função responsável por recolher as informações necessárias e adicioná-las à estrutura. É importante referir que na possibilidade de a transformação pretendida

ser uma translação é necessário armazenar as informações relativas à órbita.

Nesta fase, optamos por considerar que no ficheiro *XML* pode vir indicado a cor com que se pretende representar a figura geométrica, como podemos ver no pequeno exemplo apresentado abaixo.

```
<scene>
      <group>
            <group>
                  <!-- Sun -->
                  <scale X="3.6" Y="3.6" Z="3.6" />
                  <colour R="1" G="0.55" B="0.0" />
                  <models>
                        <model file="sphere.3d" />
                  </models>
            </group>
            //...
    </group>
</scene>
```

Deste modo, na possibilidade de o nodo representar a cor é invocada a função *parseColour* responsável por recolher e armazenar a informação na estrutura.

Na hipótese de um nodo corresponder a primitivas é invocada a *parseModels* que recorre à *readPointsFile*, tal como na primeira fase, para ler os pontos de cada ficheiro .3d e os armazena num vetor de formas geométricas que será armazenado na estrutura principal.

A última situação é a opção de serem grupos filhos e para tal a função é invocada recursivamente, de modo a que estes sejam igualmente processados.

Com o intuito de demonstrar como é efetuado o processamento de todos os nodos do ficheiro explicado anteriormente, será apresentada a função *parseGroup*:

```
void parseGroup (Group *group, XMLElement *gElement, vector<Point*>
    *orbits, int d){
     XMLElement *element = gElement->FirstChildElement();

     while (element)
     {
          if (strcmp(element->Name(),"translate") == 0)
```

```
7               parseTranslate(group,element,orbits,d);
8
9           else if (strcmp(element->Name(),"scale") == 0)
10              parseScale(group,element);
11
12          else if (strcmp(element->Name(),"rotate") == 0)
13              parseRotate(group,element);
14
15          else if (strcmp(element->Name(),"models") == 0)
16              parseModels(group, element);
17
18      else if (strcmp(element->Name(),"colour") == 0)
19              parseColour(group, element);
20
21          else if (strcmp(element->Name(),"group") == 0)
22          {
23              Group *child = new Group();
24              group->addGroup(child);
25              parseGroup(child,element,orbits,d+1);
26          }
27
28          element = element->NextSiblingElement();
29      }
30  }
```

2.2.9 Representação do sistema solar

Tal como na fase anterior, é o *engine* que será responsável por representar a cena, nomeadamente a função *drawSystem*. No entanto, para além de desenhar as figuras terá também de ter em atenção as transformações a aplicar. Assim, esta função irá receber como argumento uma variável *Group** que contém toda a informação obtida no parsing do ficheiro *XML*.

Antes de desenhar as primitivas, foi necessário verificar quais as transformações existentes, isto é, se corresponde a uma rotação, translação ou escala, para que estas possam ser realizadas de acordo com os parâmetros fornecidos. Foi preciso ter em atenção se era indicada a cor da primitiva, dado que consideramos a hipótese de no ficheiro *XML* vir informações referentes a esta, como já foi explicado anteriormente.

De seguida, são desenhadas as diferentes formas geométricas presentes na *Group**.

Para cada uma, são representados os diferentes pontos recorrendo à *glVertex3f* tal como foi efetuado na primeira fase do projeto.

Uma vez desenhado o grupo principal é realizado o mesmo processo para os seus filhos de modo recursivo.

É importante referir, que no início do processo tornou-se essencial guardar o estado inicial da matriz e no fim repô-lo, dado que as transformações aplicadas alteram as posições dos eixos. Tal é possível através de um *glPushMatrix()* e *glPopMatrix()* respetivamente, tal como pode ser observado de seguida:

```
void drawSystem(Group *system){
    glPushMatrix();
    const char* type;

    for (Transformation *t: system->getTransformations()){
        type = t->getType().c_str();
        if(!strcmp(type,"translation")) {
            glTranslatef(t->getX(), t->getY(), t->getZ());
        }


        else if(!strcmp(type, "rotation")) {
            glRotatef(t->getAngle(),
                      t->getX(),
                      t->getY(),
                      t->getZ());
        }

        else if(!strcmp(type,"scale")) {
            glScalef(t->getX(), t->getY(), t->getZ());
        }
    else if(!strcmp(type,"colour")) {
            glColor3f(t->getX(), t->getY(), t->getZ());
        }
    }

    glBegin(GL_TRIANGLES);
    for (Shape *shape : system->getShapes()){

        for (Point *p : shape->getPoints())
            glVertex3f(p->getX(), p->getY(), p->getZ());
```

```
32      }
33      glEnd();
34
35      for (Group *g : system->getGroups())
36          drawSystem(g);
37
38      glPopMatrix();
39  }
```

Foi ainda necessário representar as órbitas dos planetas que se encontram armazenadas numa variável global *vector<Points> orbits*. Este vetor é construído com base nas coordenadas usadas para realizar a translação. É importante realçar, que apenas consideramos as translações sobre planetas principais, pelo que neste vetor só guardamos os que tinham uma profundidade inferior a 2.

Assim, para cada ponto do vetor calculamos a sua distância à origem e obtemos os diferentes pontos que nos permitem calcular a circuferência que deverá representar a órbita pretendida.

$$\text{radius} = \text{sqrt}(x*x + y*y + z*z)$$

Tal processo é efetuado na função *drawOrbits* que se encontra ilustrada abaixo:

```
1  void drawOrbits(){
2      glColor3f(1.0f, 1.0f, 0.94f);
3      for(auto const& p : orbits){
4          glBegin(GL_POINTS);
5          for (int j = 0 ; j < 200 ; j++)
6          {
7              float x = p->getX() * p->getX();
8              float y = p->getY() * p->getY();
9              float z = p->getZ() * p->getZ();
10             float radius = sqrtf(x + y + z);
11             float alpha = j * 2 * M_PI / 200;
12             glVertex3f(radius * cos(alpha), 0, radius * sin(alpha));
13         }
14         glEnd();
15     }
16 }
```

3 Demonstração

Para demonstrar as aplicações previamente referidas, iremos demonstrar de que forma somos capazes de executar cada uma delas:

- **Generator**

```
$ cd generator
$ mkdir build && cd build
$ cmake ..
$ make
$ ./generator torus 0.5 3 20 20 torus.3d
$ ./generator sphere 3 20 20 sphere.3d
```

Figura 4. Generator

O projeto inclui uma pasta *files* que contém todos os ficheiros *XML* e onde serão criados os ficheiros do torus e da esfera .3d que posteriormente serão lidos para a geração do Sistema Solar.

- **Engine**

```
$ cd engine
$ mkdir build && cd build
$ cmake ..
$ make
$ ./engine SolarSystem.xml
```

Figura 5. Engine

Como podemos visualizar na Figura 5, irá ser passado como parâmetro o ficheiro *XML* de todo o Sistema Solar resultante da leitura dos ficheiros .3d previamente gerados.

3.1 Usabilidade

Da primeira fase mantivemos um menu que permite ao utilizador conhecer os comandos com os quais poderá interagir com o sistema, que contém agora alguns comandos extra. A forma como foi implementada a rotação e o formato como as figuras geométricas são apresentadas manteve-se. Contudo, as funcionalidades relativas à câmara passaram a ser implementadas na classe apresentada *Camara*. Acrescentamos ainda a possibilidade de fazer reset da câmara usando a tecla F6, como podemos verificar na seguinte imagem.

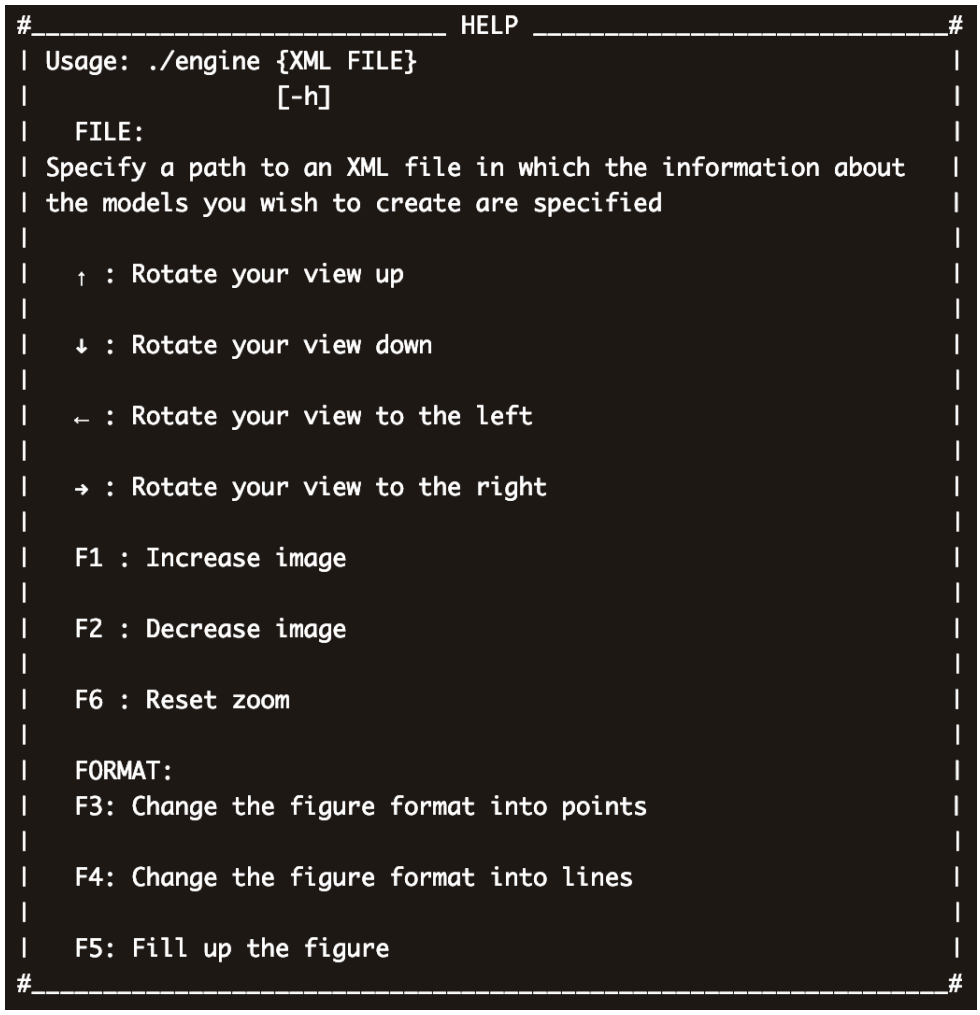


Figura 6. Menu

3.1.1 Menu do Sistema Solar

De modo a facilitar a interação com o utilizador, optamos também por adicionar um menu. Para aceder a este, basta pressionar o lado direito do rato. Os sub-menus do menu *Planet* foram criados de forma *responsive*. Assim, por exemplo, no caso apresentado de seguida, existem 8 planetas pelo que é apresentado um sub-menu para o sol e para cada um dos planetas que constituem o sistema solar. Neste menu será possível:

- Focar num determinado planeta
- Fechar janela

Desta forma, selecionando um dos sub-menus serão realizadas alterações na camâra de modo a focar-se no planeta que o utilizador pretende.

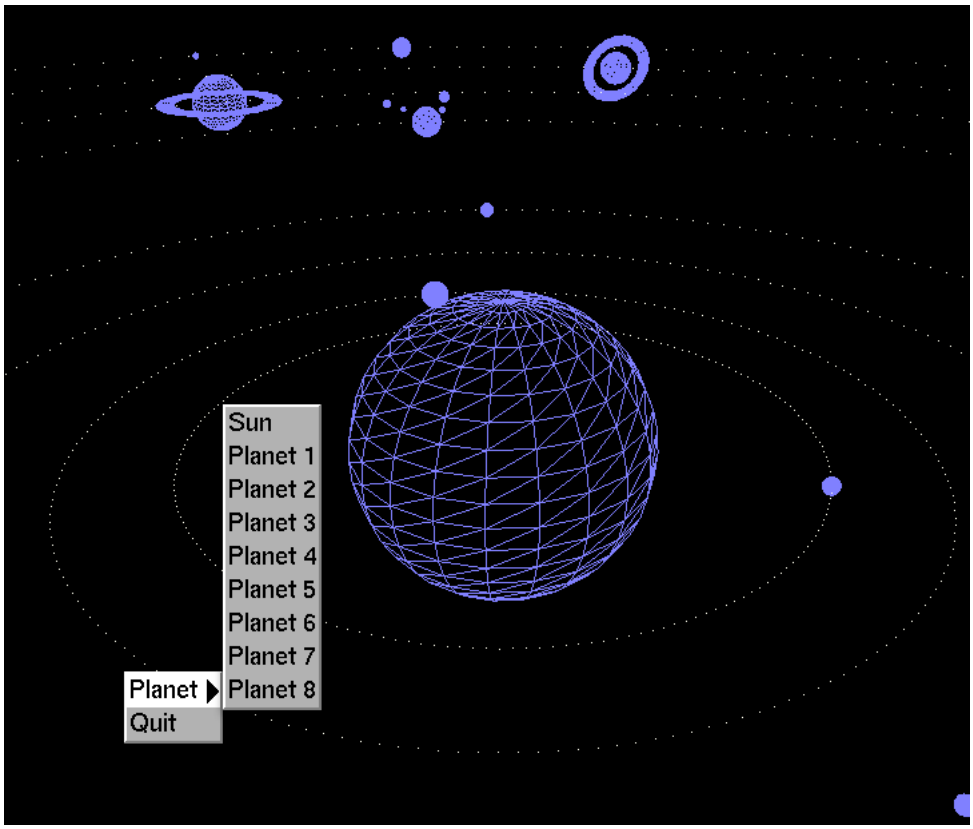


Figura 7. Menu dos Sistema Solar

3.2 Sistema Solar

Para a construção do sistema solar, começamos por ter em conta os planetas que o constituem, a sua disposição bem como as suas formas e escalas. Para além disto, tivemos o cuidado de verificar quais destes possuem satélites naturais.

Tendo por base a constituição do sistema tal como na ilustração apresentada na figura 8, procuramos criar um sistema mais próximo possível da realidade. Numa fase inicial começamos por elaborar um ficheiro *XML* que permitisse a construção do sistema solar tendo em atenção as formas, escalas, rotações e translações, obtendo o seguinte resultado.

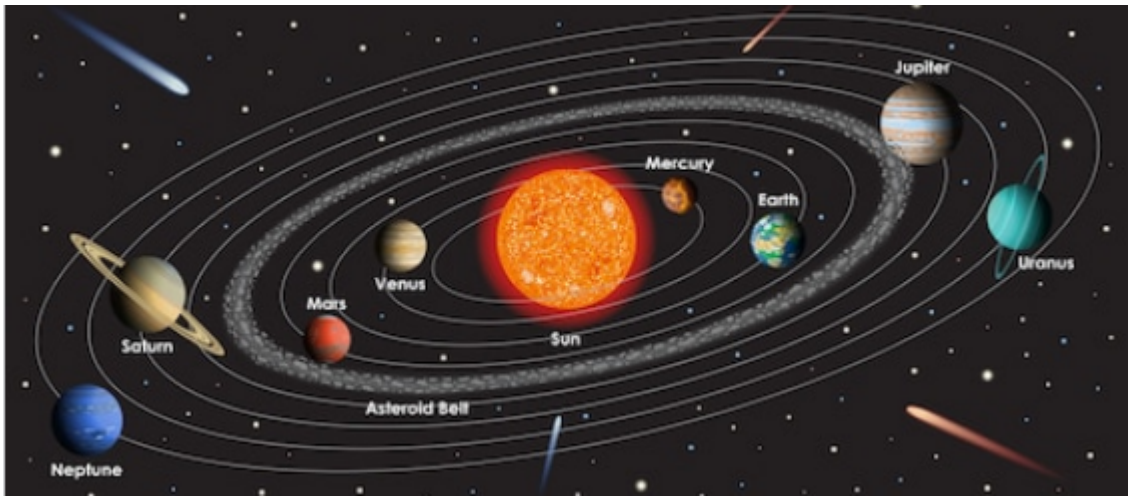


Figura 8. Sistema Solar

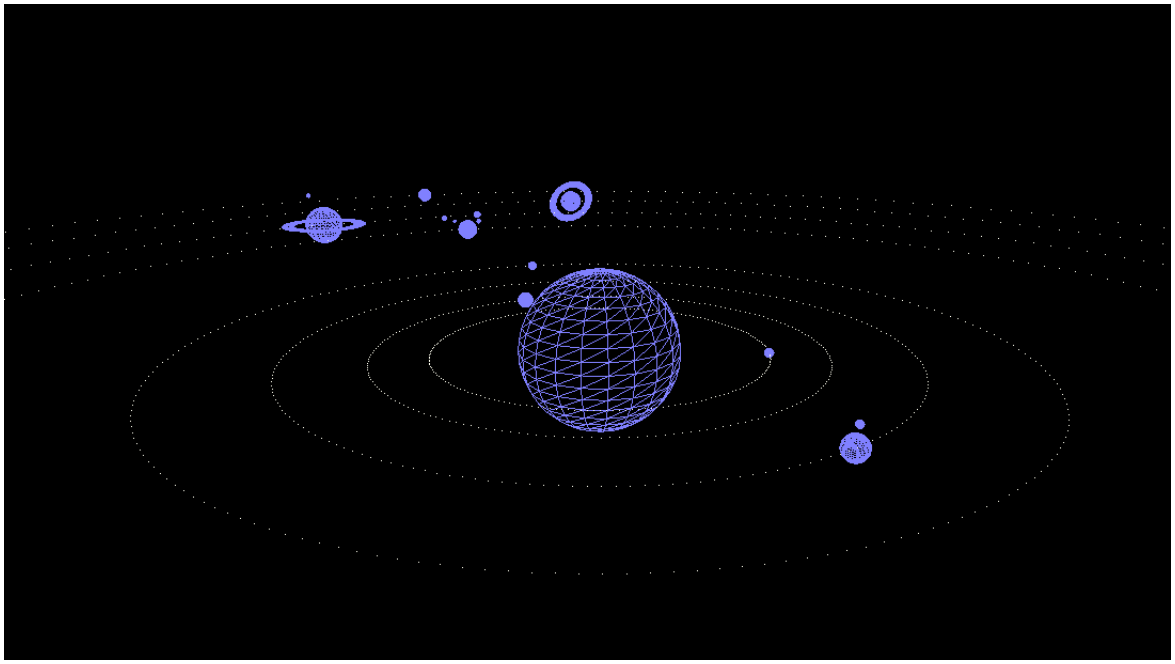


Figura 9. Sistema Solar

Por fim, decidimos atribuir diferentes cores ao sol, planetas e satélites naturais, sendo o resultado final o seguinte.

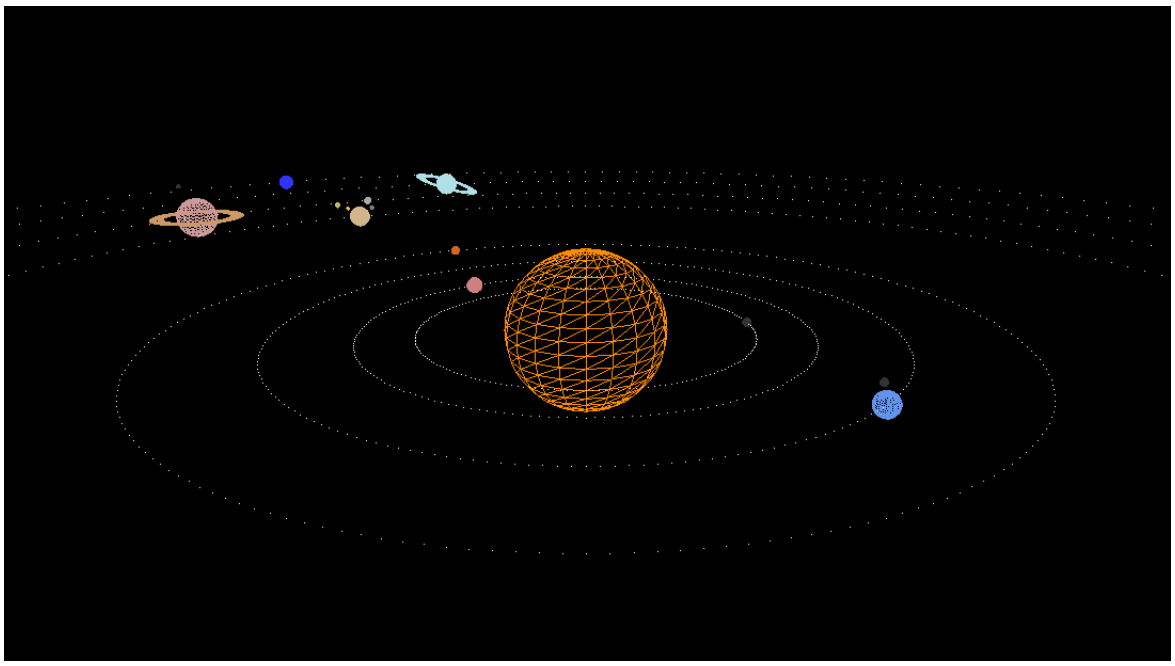


Figura 10. Sistema Solar com cores

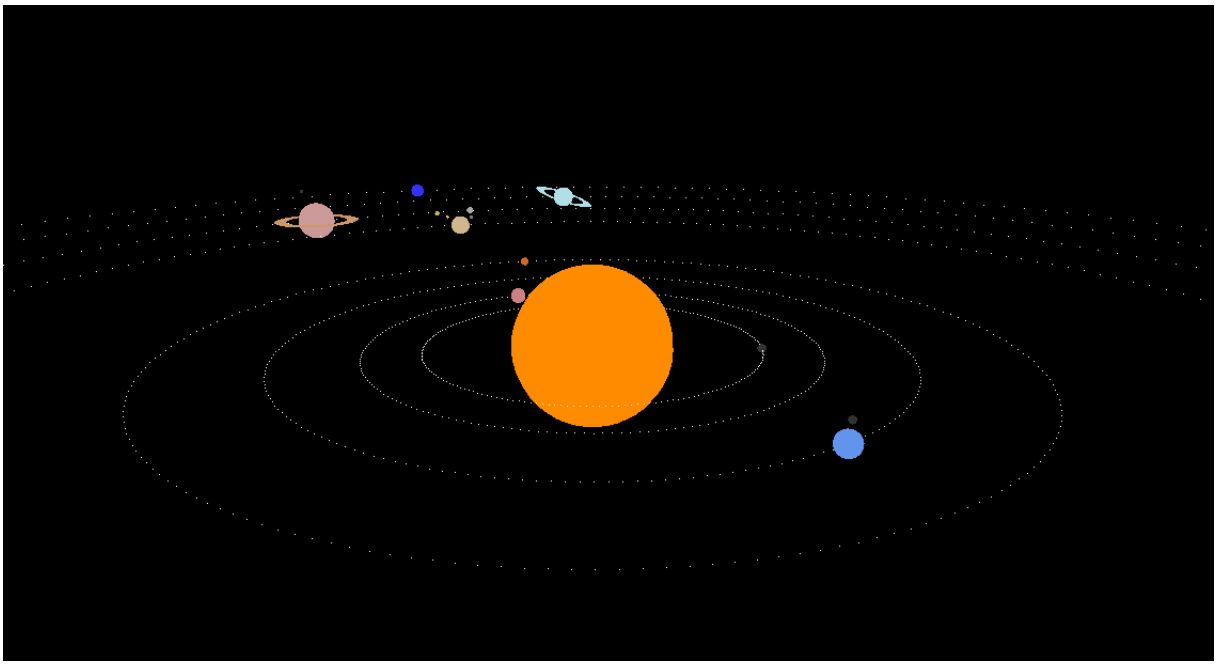


Figura 11. Modelo do Sistema Solar preenchido com cores

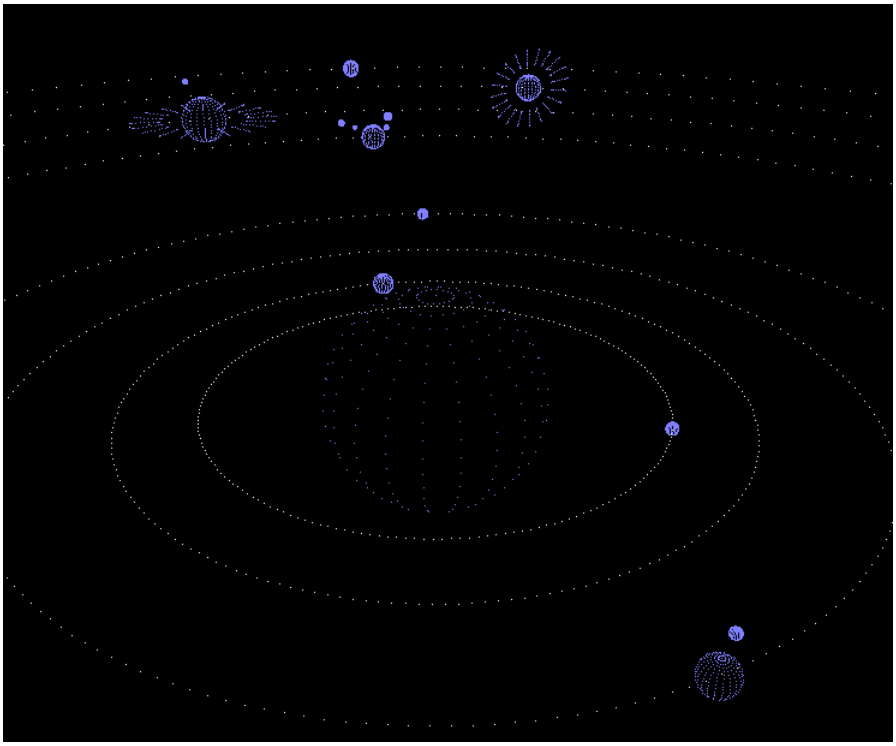


Figura 12. Sistema Solar com pontos

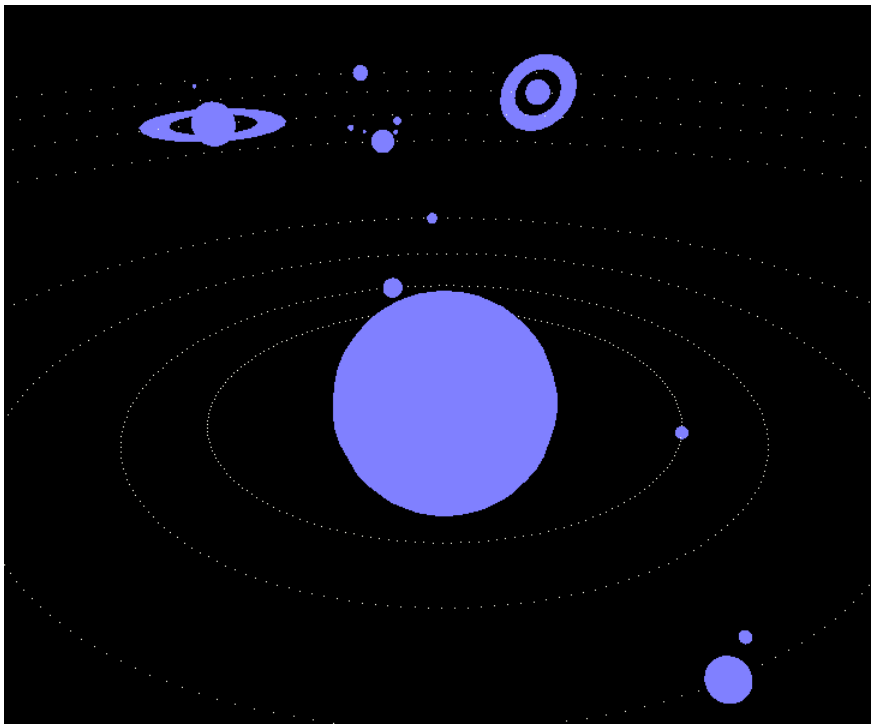


Figura 13. Sistema Solar preenchido

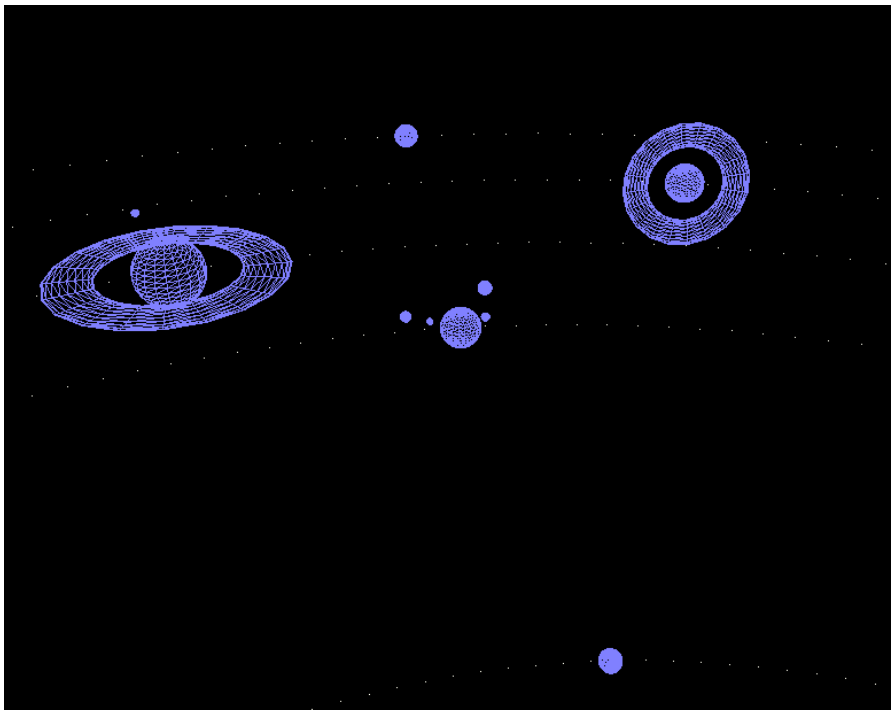


Figura 14. Sistema Solar na perspetiva do planeta 5 e com zoom in

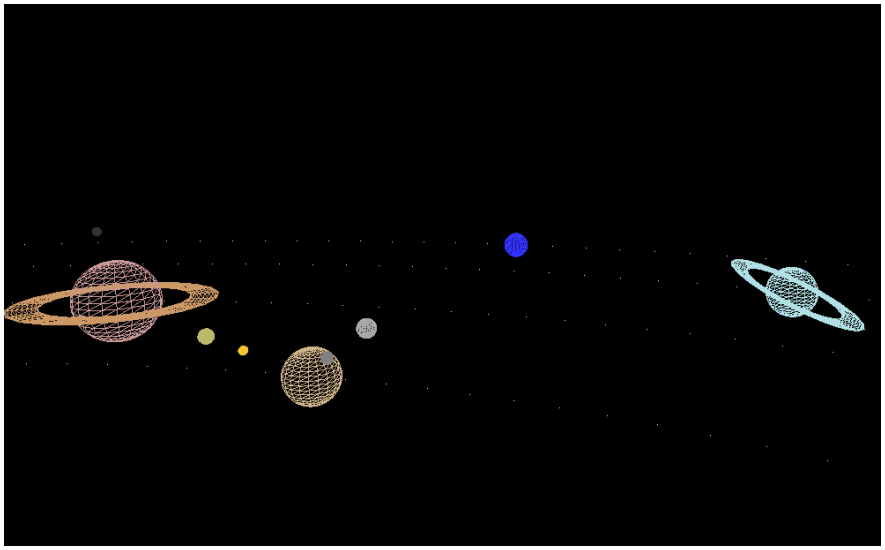


Figura 15. Sistema Solar com cores na visão do planeta 8

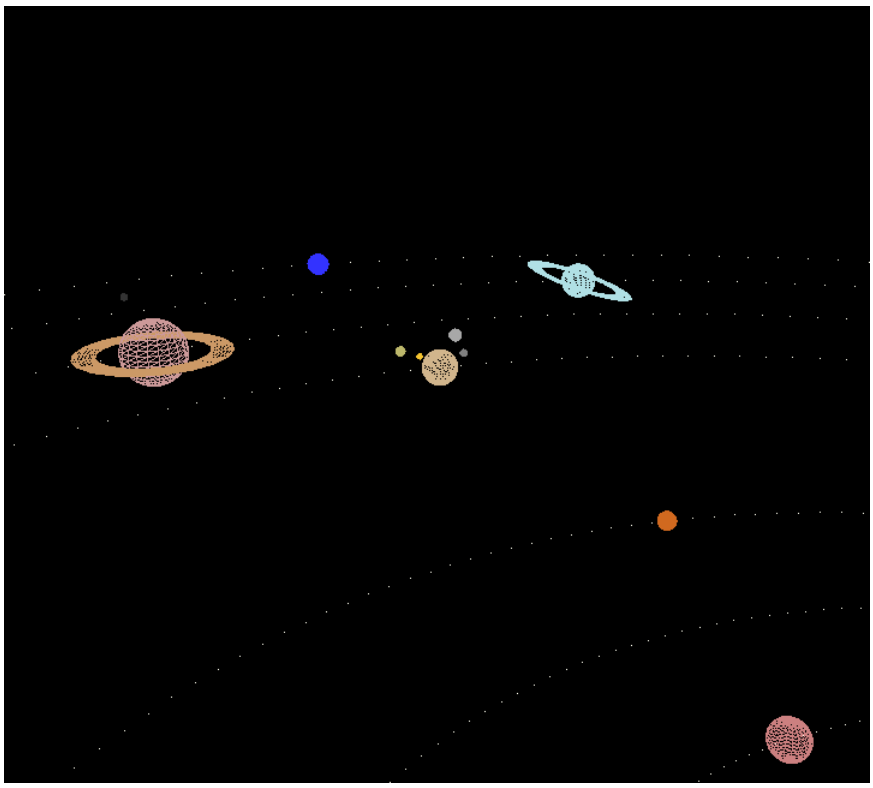


Figura 16. Sistema Solar com cores na prespetiva do planeta 5

4 Conclusão

Na realização desta fase, devido à complexidade da estrutura usada na primeira fase e à grande quantidade de informação armazenada decidimos alterar as estruturas, sendo ao nível do engine feita uma reformulação do código.

De modo a construirmos o Sistema Solar foi necessário a criação de um novo ficheiro *XML* pelo que foram realizadas alterações na realização do parsing do mesmo.

Foi necessária a implementação da primitiva Torus, de modo a conseguir uma representação mais realista do Sistema Solar, devido à criação de anéis. De forma a tornar a visualização do Sistema mais acessível e prática recorreu-se à utilização do rato para a movimentação da câmera, sendo esta de fácil usabilidade. Com o intuito de fornecer diferentes perspetivas do Sistema Solar foi implementado um menu que permite focar nos diferentes planetas.

Em última instância, esperamos que o resultado obtido nesta fase corresponda às expectativas e sirva de suporte para a elaboração das próximas etapas do projeto.