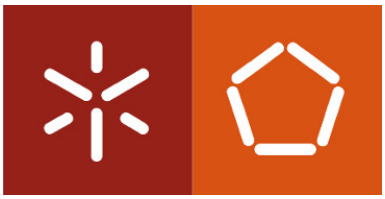


Computação Gráfica

Trabalho Prático - Fase 2

4 de abril de 2021

Patrícia Pereira, A89578

Meriem Khammasi, A85829

Jaime Oliveira, A89598

Luís Magalhães, A89528

Conteúdo

1 *Abstract*

Segunda fase do trabalho prático realizado no âmbito da Unidade Curricular de Computação Gráfica da Universidade do Minho. Esta fase consiste na adição de novas funcionalidades e modificações ao trabalho já realizado, com o intuito de tornar possível a realização de transformações geométricas, tais como rotações, translações e alterações de escala. Para além disto, adicionou-se uma nova figura ao *generator*, o *Torus*, que permite a criação de anéis para alguns planetas.

2 Introdução

Neste relatório vamos esclarecer os aspetos mais importantes relativamente à segunda fase do nosso trabalho, onde faremos uma síntese e explicação do código elaborado, tal como os resultados obtidos.

Iremos também descrever e explicar as abordagens que tomamos quanto à realização do *generator* e do *engine*, bem como o seu funcionamento apresentando algumas imagens de exemplo para facilitar a compreensão. Iremos também especificar os diversos raciocínios utilizados para ultrapassar as barreiras encontradas.

3 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, o *Generator* e o *Engine*.

3.1 *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, e como já foi referidos, decidimos construir a figura *Torus*.

3.1.1 *Torus*

O *torus* acabou por ser uma primitiva que teve que ser adicionada a este projeto, pois será utilizada na recriação de certos objetos como por exemplo o anel de Saturno.

A função responsável por criar a primitiva em si segue muitos dos princípios que foram mencionados para a criação de uma esfera na fase anterior. Utiliza dois *"for loops"*, um exterior e um interior.

O *"loop"* exterior vai desenhando uma *"slice"* do *"torus"* por cada iteração, e vai repetindo este processo várias vezes descrevendo um movimento circular até o desenho ficar completo. Por sua vez, o *"loop"* interior é responsável por descrever os pontos de

uma *"slice"*, e vai fazendo isto no sentido positivo da coordenada Y, ou seja, os pares de triângulos estão a ser desenhados de baixo para cima.

Deve ser menciado que uma das principais diferenças entre a esfera e o *torus* é o facto de este ter um buraco no meio, por isso, o método usado para desenhar uma *slice* terá que ser diferente. Desta vez, o *"for loop"* interior irá desenhar a parte de fora e a parte de dentro do *torus* simultaneamente, e ambas secções devem manter sempre a seguinte regra: a distância entre os pontos da superfície do *torus* e os pontos de um circulo imaginário no centro do seu interior, deve ser sempre igual a metade do valor da espessura do *torus*.

```
p1.x = radius * sin(ang)  - or1 * sin(ang);
p1.y = oy;
p1.z = radius * cos(ang) - or1 * cos(ang);

p2.x = radius * sin(ang) - or2 * sin(ang);
p2.y = oy + h;
p2.z = radius * cos(ang) - or2 * cos(ang);

p3.x = radius * sin(ang + delt) - or1 * sin(ang + delt);
p3.y = oy;
p3.z = radius * cos(ang + delt) - or1 * cos(ang + delt);

points.push_back(p1);
points.push_back(p2);
points.push_back(p3);

p1.x = radius * sin(ang)  - or2 * sin(ang);
p1.y = oy + h;
p1.z = radius * cos(ang) - or2 * cos(ang);

p2.x = radius * sin(ang + delt) - or2 * sin(ang + delt);
p2.y = oy + h;
p2.z = radius * cos(ang + delt) - or2 * cos(ang + delt);

p3.x = radius * sin(ang + delt) - or1 * sin(ang + delt);
p3.y = oy;
p3.z = radius * cos(ang + delt) - or1 * cos(ang + delt);

points.push_back(p1);
points.push_back(p2);
points.push_back(p3);

p1.x = radius * sin(ang)  + or1 * sin(ang);
p1.y = oy;
p1.z = radius * cos(ang) + or1 * cos(ang);

p2.x = radius * sin(ang + delt) + or1 * sin(ang + delt);
p2.y = oy;
p2.z = radius * cos(ang + delt) + or1 * cos(ang + delt );

p3.x = radius * sin(ang) + or2 * sin(ang);
p3.y = oy + h;
p3.z = radius * cos(ang) + or2 * cos(ang);

points.push_back(p1);
points.push_back(p2);
points.push_back(p3);

p1.x = radius * sin(ang)  + or2 * sin(ang);
p1.y = oy + h;
p1.z = radius * cos(ang) + or2 * cos(ang);

p2.x = radius * sin(ang + delt) + or1 * sin(ang + delt);
p2.y = oy;
p2.z = radius * cos(ang + delt) + or1 * cos(ang + delt);

p3.x = radius * sin(ang + delt) + or2 * sin(ang + delt);
p3.y = oy + h;
p3.z = radius * cos(ang + delt) + or2 * cos(ang + delt);

points.push_back(p1);
points.push_back(p2);
points.push_back(p3);
```

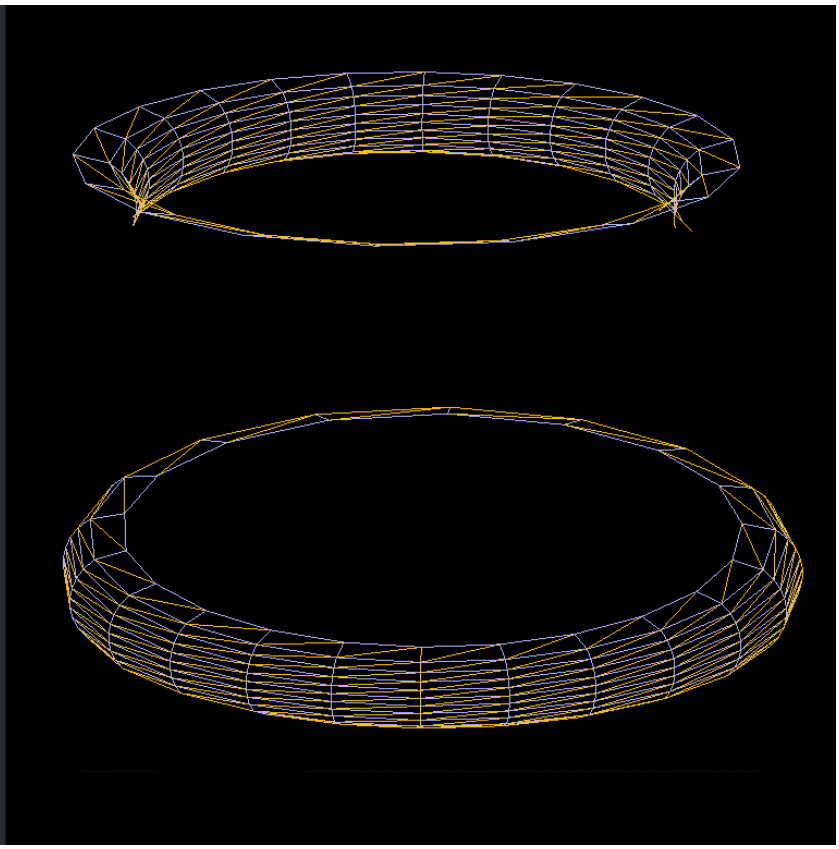
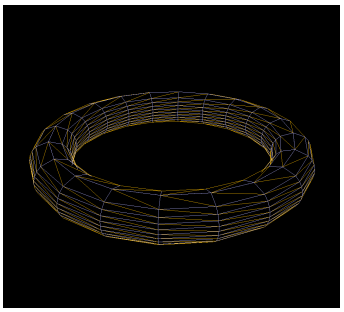


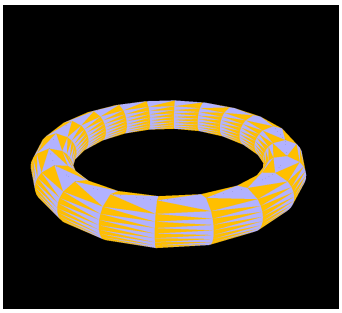
Figura 1: Segmento do código do *"for loop"* interior.

3.1.2 Interpretação do *Torus* no *Engine*

Representação picotada.



Representação por linhas.



Representação completa.

3.2 *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.

3.2.1 *GeoTansforms*

Na primeira fase do projeto optamos por criar uma estrutura Point, que continuamos a recorrer à sua utilização no *generator*, tal como no novo ficheiro de classe *geoTransforms* no *Engine*.

Uma vez que pretendíamos armazenar, não só, todos os pontos de uma primitiva (lida de um ficheiro .3d) num vetor de "Point"s, mas também as duas cores que serão usadas para desenhar uma primitiva (as cores que alternam entre os triângulos).

```
//struct que guarda coordenadas de um ponto
typedef struct point{

  float x;
  float y;
  float z;

} Point;

//struct que guarda as duas cores de cada planeta
typedef struct color{
  //cor 1
  float r1;
  float g1;
  float b1;
  // cor 2
  float r2;
  float g2;
  float b2;
} Color;
```

Figura 2: Structs Point e Colour usadas para guardar as cores de uma primitiva e as coordenadas de um ponto.

Nesta classe serão também guardadas as informações relativas a todas as translações, rotações e escalas que serão usadas para transformar o grupo num vetor de "Transformations"e todos os astros secundários que também sofrem as transformações mencionadas, mas que possuem transformações, cores, primitivas e filhos próprios num vetor da mesma classe.

```
class geoTransforms{

  private:

      vector<Point> points; //conjunto dos pontos de uma primitiva
      vector<Transformations*> transformations; //conjunto das transformações que serão aplicadas ao grupo
      Color cor; //as duas cores da primitiva
      vector<geoTransforms*> child; //vetor que guarda os grupos filhos deste grupo

  public:
    geoTransforms();
    geoTransforms(vector<Point> p, vector<Transformations*> t, Color c, vector<geoTransforms*> chi);
    vector<Point> getPoints();
    vector<Transformations*> getTransformations();
    vector<geoTransforms*> getChild();
    void addPoint(float x, float y, float z);
    void addTransformations(string t, float px, float py, float pz, float a);
    void addChild(geoTransforms* chi);
    void popPoints();
    void popTransformations();
    float getR1();
    float getG1();
    float getB1();
    float getR2();
    float getG2();
    float getB2();
    void setColor(float red1, float gre1, float blu1, float red2, float gre2, float blu2);

};
```

Figura 3: Classe que guarda os pontos e as transformações geométricas das figuras.

3.2.2 Transformations

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 Transformations {
  private :
    string type;
    float x, y, z;
    float angle;

  public :
      Transformations();
      Transformations(string t, float px, float py, float pz, float a);
      //void apply();
      string getType();
        float getX();
        float getY();
        float getZ();
        float getAngle();

};
```

Figura 4: Classe que guarda as transformações geométricas das figuras.

3.2.3 Representação XML

Para esta fase foi necessário alterar a forma como efetuamos o *parsing* do XML, dado que 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, podendo estas estar encadeadas. Assim, apresentamos de seguida um exemplo de um ficheiro XML.

```
<scene>

    <!--Sol -->
    <group>
      <scale  X = "3.6f" Y="3.6f" Z = "3.6f" />
      <models>
        <color R1 = "1.0f" G1 = "1.0f" B1 = "0.0f" R2 = "1.0f" G2 = "0.8f" B2 = "0.0f" />
        <model file="../files/sphere.3d"/>
      </models>
    </group>
    <!--Mercury -->
    <group>
      <scale  X = "0.24f" Y = "0.24f" Z = "0.24f"/>
      <translate X = "-10.0f" Z = "20.0f"/>
      <models>
        <color R1 = "0.435f" G1 = "0.431f" B1 = "0.443f" R2 = "0.549f" G2 = "0.549f" B2 = "0.576f"/>
        <model file="../files/sphere.3d"/>
      </models>
    </group>
    <!--Venus-->
    <group>
      <scale  X = "0.6f" Y = "0.6f" Z = "0.6f"/>
      <translate X = "30.0f"/>
      <models>
        <color R1 = "0.784f" G1 = "0.764f" B1 = "0.749f" R2 = "0.827f" G2 = "0.819f" B2 = "0.811f"/>
        <model file="../files/sphere.3d"/>
      </models>
    </group>
```

Figura 5: Representação do Sistema Solar em XML.

Assim sendo, tal como na primeira fase do projeto recorremos à API do *tinyxml2* para realizar o parsing dos ficheiros, sendo alterado apenas o modo de leitura. A primeira etapa deste processo consiste em carregar o ficheiro para a memória, o que no nosso caso é efetuado na função *readXML*. Esta é também responsável por invocar em caso de sucesso a *readGroup* que lê um grupo do ficheiro XML. Para tal, é efetuado um ciclo que percorre todos os nodos e para cada nodo, verifica o caso em que se encontra.

Assim, na eventualidade de este corresponder a uma transformação geométrica, adiciona à estrutura de dados associada a esse grupo (*addTransformations*), aplicando essas transformações na *renderScence* onde esta é invocada.

Na hipótese de um nodo corresponder a primitivas, recorremos à *readFile*, tal como na primeira fase, para ler os pontos de cada ficheiro .3d.

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 *readGroup*:

```
geoTransforms* readGroup(XMLElement* group){

    XMLElement *element, *innerElement;
    float x, y, z, ang;
    float r1, g1, b1, r2, g2, b2;
    geoTransforms* obj = new geoTransforms();

    element = group->FirstChildElement();

    while (element != nullptr){

        if (strcmp(element->Value(), "translate") == 0) {

                x = element->FloatAttribute("X");
                y = element->FloatAttribute("Y");
                z = element->FloatAttribute("Z");
                obj->addTransformations("translate",x,y,z,0.0f);
        }

        else if (strcmp(element->Value(), "rotate") == 0) {
                x = element->FloatAttribute("X");
                y = element->FloatAttribute("Y");
                z = element->FloatAttribute("Z");
                ang = element->FloatAttribute("Angle");
            obj->addTransformations("rotate",x,y,z,ang);
        }

        else if (strcmp(element->Value(), "scale") == 0) {

                x = element->FloatAttribute("X");
                y = element->FloatAttribute("Y");
                z = element->FloatAttribute("Z");
                obj->addTransformations("scale",x,y,z,0.0f);
        }
```

```
        else if (strcmp(element->Value(), "models") == 0){

            innerElement = element->FirstChildElement();

            while (innerElement != nullptr){

                if (strcmp(innerElement->Value(), "model") == 0){

                    string file;
                    file = innerElement->Attribute("file");

                    if (!file.empty() && readFile(file, obj) == -1) return nullptr;
                }

                else if (strcmp(innerElement->Value(), "color") == 0){

                    r1 = innerElement->FloatAttribute("R1");
                    g1 = innerElement->FloatAttribute("G1");
                    b1 = innerElement->FloatAttribute("B1");
                    r2 = innerElement->FloatAttribute("R2");
                    g2 = innerElement->FloatAttribute("G2");
                    b2 = innerElement->FloatAttribute("B2");

                    obj->setColor(r1,g1,b1,r2,g2,b2);

                }

                    innerElement = innerElement->NextSiblingElement();
            }
        }

    else if (strcmp(element->Value(), "group") == 0){
        geoTransforms* chi = readGroup(element);
        obj->addChild(chi);
        }

        element = element->NextSiblingElement();
    }

    return obj;
}
```

Figura 6: Representação da função *readGroup*.

3.2.4 Representação do sistema solar

Tal como na fase anterior, é o *engine* que será responsável pela representação gráfica, nomeadamente a função *drawAndColor*. No entanto, para além de desenhar as figuras terá também de ter em atenção as transformações a aplicar. 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. 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, como pode ser observado de seguida:

```
void drawAndColor(geoTransforms* obj) {


        glPushMatrix();


        vector<Transformations*> tran = obj->getTransformations();

        //transformações
        for (Transformations *transf : obj->getTransformations()){

        if (transf->getType() == "translate") {
            glTranslatef(transf->getX(), transf->getY(), transf->getZ());
        }
        if (transf->getType() == "scale") {
            glScalef(transf->getX(), transf->getY(), transf->getZ());
        }
        if (transf->getType() == "rotate") {
            glRotatef(transf->getAngle(), transf->getX(), transf->getY(), transf->getZ());
        }
    }

    int i=0;
    bool cor = true;

  glBegin(GL_TRIANGLES);
    for (const Point pt : obj->getPoints())  {

        if( i==3 ) {
            cor = !cor;
            i=0;
        }

        if(cor) {
            glColor3f(obj->getR1(), obj->getG1(), obj->getB1());
            glVertex3f(pt.x, pt.y, pt.z);
        } else {
            glColor3f(obj->getR2(), obj->getG2(), obj->getB2());
            glVertex3f(pt.x, pt.y, pt.z);
        }
        i++;
    }
    glEnd();

        for (geoTransforms *chi : obj->getChild()) drawAndColor(chi);

        glPopMatrix();
}
```

Figura 7: Representação da função *drawAndColor*.

4 Demonstração

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

```
meriam@meriam-E205SA:~$ cd Desktop/CGFase2/Generator/
meriam@meriam-E205SA:~/Desktop/CGFase2/Generator$ g++ generator.cpp  -o
generator -lGL -lGLU -lglut
meriam@meriam-E205SA:~/Desktop/CGFase2/Generator$ ./generator torus 3 2
10 20 torus.3d
```

Figura 8: *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*

```
meriam@meriam-E205SA:~/Desktop/CGFase2/Engine$ g++ engine.cpp geoTransfo
rms.cpp  Transformations.cpp tinyxml2.cpp -o engine -lGL -lGLU -lglut
meriam@meriam-E205SA:~/Desktop/CGFase2/Engine$ ./engine solarSystem.xm
```

Figura 9: *Engine*.

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

4.1 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. No entanto, decidimos acrescentar a possibilidade de interagir com o sistema usando o rato e algumas teclas, como podemos verificar na seguinte imagem.

```
void keyboard (unsigned char k, int i, int j){
switch (k){

    case 'w':
                movX += sin(alpha);
              movZ += cos(alpha);
              break;

    case 's':
                movX -= sin(alpha);
                movZ -= cos(alpha);
              break;

 case 'd':
            movX += sin(alpha - M_PI / 2.0f);
              movZ += cos(alpha -  M_PI / 2.0f);
              break;

 case 'a':
              movX -= sin(alpha - M_PI / 2.0f);
              movZ -= cos(alpha - M_PI / 2.0f);
            break;

    case 'q':
                    movY -= 0.5f;
                    break;

 case 'e':
                    movY += 0.5f;
                    break;

    //draw mode
    //points
    case 'p':
            line = GL_POINT;
            break;
    //lines
    case 'l':
            line = GL_LINE;
            break;

    //full
    case 'f':
            line = GL_FILL;
            break;

        default:
```

```
void mouseActiveMovEvent(int x, int y){

    mposX1 = (float)x;
    mposY1 = (float)y;

    mdeltX = (mposX1 - mposX2) * 0.001f;
    mdeltY = (mposY1 - mposY2) * 0.001f;

    alpha += mdeltX;
    beta += mdeltY;

    glutPostRedisplay();
    mposX2 = mposX1;
    mposY2 = mposY1;

}

//função que atualiza a posição do rato quando este não é pressionado
void mousePassiveMovEvent(int x, int y){
    mposX1 = (float)x;
    mposY1 = (float)y;

    mposX2 = mposX1;
    mposY2 = mposY1;
}
```

Figura 10: Funcionalidades do teclado e do rato.

Tal como foi criado na primeira fase um menu guideline que nos indica as teclas usadas para interagir com o sistema, este foi então modificado nesta segunda fase com as novos comandos mencionados anteriormente, como podemos verificar na seguinte figura.

```
std::cout << "                          MOVEMENT:                       " <<std::endl;
std::cout << "                                                          " <<std::endl;
std::cout << "- W :                                                     " <<std::endl;
std::cout << "            Move forwards                                 " <<std::endl;
std::cout << "                                                          " <<std::endl;
std::cout << "- S :                                                     " <<std::endl;
std::cout << "            Move backwards                                " <<std::endl;
std::cout << "                                                          " <<std::endl;
std::cout << "- D :                                                     " <<std::endl;
std::cout << "            Move to the right                             " <<std::endl;
std::cout << "                                                          " <<std::endl;
std::cout << "- A :                                                     " <<std::endl;
std::cout << "            Move to the left                              " <<std::endl;
std::cout << "                                                          " <<std::endl;
std::cout << "- E :                                                     " <<std::endl;
std::cout << "            Move Up                                       " <<std::endl;
std::cout << "                                                          " <<std::endl;
std::cout << "- Q :                                                     " <<std::endl;
std::cout << "            Move Down                                     " <<std::endl;
```

Figura 11: Parte adicionada no guia dos comandos.

4.2 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. 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, atribuindo também diferentes cores ao sol, planetas e satélites naturais,sendo o resultado final o seguinte.

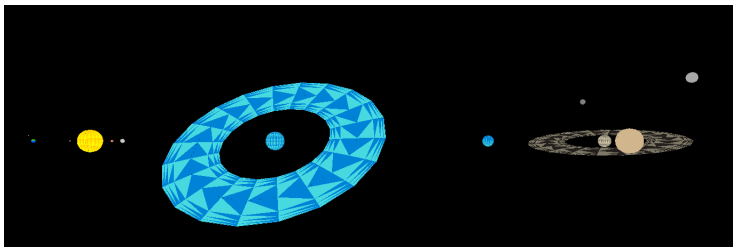


Figura 12: Modelo do sistema Solar preenchido com cores.

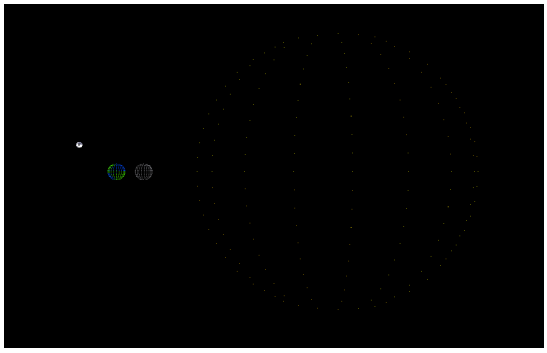


Figura 13: Modelo do sistema Solar com pontos.

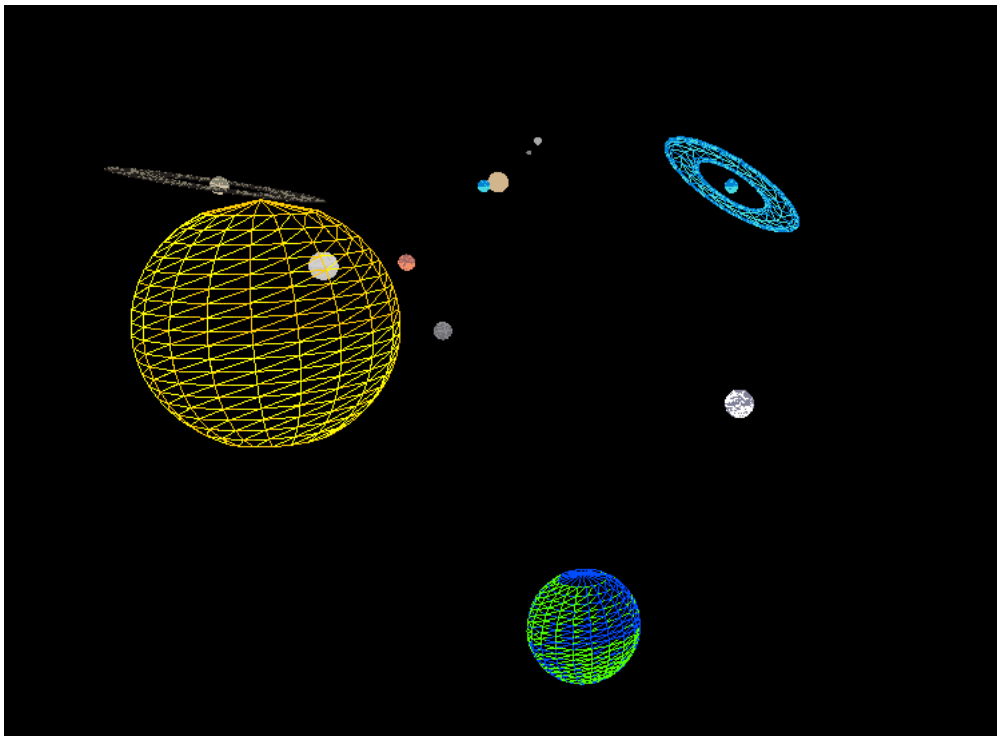


Figura 14: Modelo do sistema Solar com zoom em planetas com linhas .

5 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. 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.