Universidade do Minho
Mestrado Integrado em Engenharia Informática
Departamento de Informática

Computação Gráfica

Transformações Geométricas

Isabel Sofia da Costa Pereira A76550
José Francisco Gonçalves Petejo e Igreja Matos A77688
Maria de La Salete Dias Teixeira A75281
Tiago Daniel Amorim Alves A78218

12 de Março de 2018

Conteúdo

1 Introdução

Neste projeto, desenvolvido no âmbito da UC de Computação Gráfica, foi proposto a realização de uma mini cena gráfica 3D.

Para o desenvolvimento do mesmo foi necessário utilizar certos recursos tais como C++ e OpenGL.

O trabalho está dividido em quatro fases, estando presente neste relatório uma explicação da abordagem tomada para o desenvolvimento da segunda fase, que consiste na criação de uma cena hierárquica usando transformações geométricas.

1.1 Alterações e Conservações

Tendo em conta que esta é a segunda fase do projeto, existem, naturalmente, algumas mudanças na estrutura do código, assim como conservações, ou seja, funcionalidades implementadas na primeira fase que se mantêm.

Na primeira fase, o *XML* apenas continha o nome dos ficheiros 3d com as primitivas a elaborar. No entanto, nesta segunda fase o ficheiro contém também as transformações (`traslate`, `rotate` ou `scale`) dos eixos a aplicar sobre os ficheiros 3d que são criados através das primitivas anteriormente desenvolvidas. Estes últimos estão organizados hierarquicamente, isto é, cada grupo pode conter zero ou mais transformações, herdando as transformações do grupo pai. Para além das transformações, considerou-se conveniente a adição da informação da cor (`color`) que a figura que se pretende desenhar possua.

Com as mudanças do ficheiro *XML* foi, logicamente, necessário reestruturar o *engine* e o *parser*. Além disso, foi essencial desenvolver novas classes para o armazenamento de toda a informação.

Para a demonstração do funcionamento da cena, teve-se como objetivo a criação de um modelo estático do Sistema Solar, definido em hierarquia, que inclui o sol, os planetas, as várias luas e o cinturão de asteroides.

2 Desenvolvimento do Projeto

Para o desenvolvimento do trabalho foi conveniente utilizar o *generator* e o *engine*, que representam as duas aplicações requeridas.

2.1 Gerador

O gerador, *generator*, tal como na primeira fase, é responsável pelo cálculo dos vértices de uma figura primitiva (`plane`, `box`, `cone` e `sphere`), guardando todos esses num ficheiro 3d passado como paramêtro. Para tal, o *generator* utiliza a classe *vertex* como auxílio.

Nesta fase, considerou-se útil a adição de uma nova figura, um `torus`, na classe *vertex* para o desenho do anel de Saturno.

2.1.1 Torus

Um Torus é um sólido geométrico equivalente a um *donut*. Para gerar esta figura são necessários os parâmetros raio interior(intRadius), raio exterior(extRadius), número de fatias(slices) e número de camadas(stacks).

O ponto fundamental do algoritmo implementado está presente nos raios interior e exterior. O *torus* é construído por "anéis", sendo cada slice um anel diferente. Cada um destes anéis está divido por stacks. Para isto, é crucial saber o raio de um anel, que é o raio interior considerado, e também o raio exterior.

Para o desenho de cada um destes anéis e para ser bem separado por stacks, o ângulo *angleStack* é extremamente importante, pois fornece a distância exata que separa cada uma das stacks. Com isto, e percorrendo o ângulo *angleSlice* até 2π, obtem-se a construção final do torus.

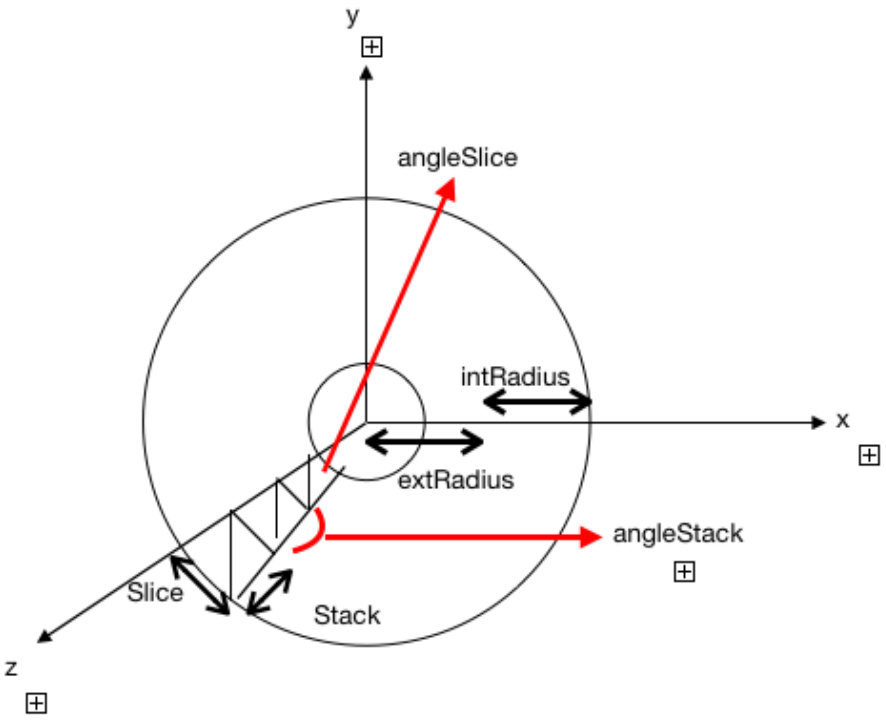

Figura 1: Torus

```
for(int i=0; i<stacks; i++){

      act = angleSlice * i;
      next = act + angleSlice;

      for(int j=0; j<slices+1; j++){

          actSt = angleStack * j;
          actStR = intRadius * cos(actSt) + extRadius;
          actStZ = intRadius * sin(actSt);

          nextSt = (j+1) * angleStack;
          nextStR = intRadius * cos(nextSt) + extRadius;
          nextStZ = intRadius * sin(nextSt);

          pointsList.push_back(
             new Point(cos(act)*actStR, sin(act)*actStR, actStZ));
          pointsList.push_back(
                    new Point(cos(next)*actStR, sin(next)*actStR, actStZ));
          pointsList.push_back(
                    new Point(cos(act)*nextStR, sin(act)*nextStR, nextStZ));

          pointsList.push_back(
             new Point(cos(act)*nextStR, sin(act)*nextStR, nextStZ));
          pointsList.push_back(
                    new Point(cos(next)*actStR, sin(next)*actStR, actStZ));
          pointsList.push_back(
                    new Point(cos(next)*nextStR, sin(next)*nextStR, nextStZ));
       }
}
```

2.2 Engine

O *engine* tem como objetivo apresentar uma janela exibindo os resultados processados através da leitura de um ficheiro *XML*. Devido à grande mudança deste ficheiro, a classe *engine* e *parser* foram sujeitas a alterações. Estas serão explicadas mais pormenorizadamente adiante, assim como as classes criadas para guardar toda a informação necessária.

2.3 Classes

Além das classes criadas na primeira fase do projeto, *Point* e *vertex*, acrescentou-se a classe *Transform* e *Struct*.

2.3.1 Point

Esta classe representa um ponto num referencial a três dimensões, com as coordendas x, y e z, o que se torna bastante útil para a representação dos vértices utilizados posteriormente para o desenho dos triângulos que elaboram as figuras primitivas.

```
class Point{

        float x;
        float y;
        float z;

        public:
                Point();
                Point(float,float,float);
                float getX();
                float getY();
                float getZ();
                void setX(float);
                void setY(float);
                void setZ(float);
};
```

2.3.2 Vertex

Todos os algoritmos para a criação das figuras primitivas (`plane`, `box`, `cone`, `sphere` e `torus`) estão presentes e descritos neste ficheiro, sendo que os quatro primeiros já foram apresentados e explicados no relatório da primeira fase e o quinto neste.

```
class Vertex{

        vector<Point*> pointsList;
```

```
        public:
                Vertex();
                Vertex(vector<Point*>);
                vector<Point*> getPointsList();
                void setPointsList(vector<Point*>);
                void makePlane(float);
                void makeBox(float, float, float, int);
                void makeCone(float, float, int, int);
                void makeSphere(float, int, int);
                void makeTorus(float, float, int, int);
};
```

2.3.3 Transform

Esta classe tem como objetivo o armazenamento dos dados de uma transformação geométrica (`traslate`, `rotate` ou `scale`) ou de uma cor (`color`). Para tal, guarda-se o nome da informação que se está a guardar, as coordenadas (ou os valores para, sendo estas representadas como *Point*, e finalmente o ângulo, que dependendo da transformação, poderá ou não existir.

```
class Transform{

        string name;
        float ang;
        Point* point;

    public:
        Transform();
        Transform(string,float,Point*);
        string getName();
        float getAngle();
        Point* getPoint();
        void setName(string);
        void setAngle(float);
        void setPoint(Point*);
        Transform* clone();
};
```

2.3.4 Struct

Tal como sugere o nome, esta é a classe principal de armazenamento que guarda os dados retirados do ficheiro *XML*.

Nesta estrutura é possível armazenar todas as informações que estão associadas a uma determinada figura tal como as transformações e cor que a define. A figura, por sua vez, é representada por um ficheiro com extenção 3d onde se pode encontrar todos os pontos que a constitui. Sendo assim, achou-se conveniente armazenar igualmente o nome do ficheiro e todos os vértices contidos neste.

```
class Struct{

    string file;
    vector<Transform*> refit;
    vector<Point*> points;

    public:
        Struct();
        Struct(string,vector<Transform*>,vector<Point*>);
        string getFile();
        vector<Transform*> getRefit();
        vector<Point*> getPoints();
        void setFile(string);
        void setRefit(vector<Transform*>);
        void setPoints(vector<Point*>);
        void addTransform(Transform*);
        void addTransform(vector<Transform*>);
        void clear();
        int size();
};
```

2.4 Outros Ficheiros

2.4.1 tinyxml2

O *tinyxml2* é uma ferramenta que processa ficheiros *XML*. Desta forma, esta é de extrema importância para o funcionamento do *Parser*.

```
namespace tinyxml2
{
class XMLDocument;
class XMLElement;
class XMLAttribute;
class XMLComment;
class XMLText;
class XMLDeclaration;
class XMLUnknown;
class XMLPrinter;
...
}
```

2.4.2 Parser

Este ficheiro é crucial para o bom funcionamento do *engine* porque é este que efetua o parsing do ficheiro *XML*. Como o *XMl* sofreu alterações, este ficheiro também foi modificado. Desta forma, o *Parser* é responsável por inserir toda a informação encontrada no documento *XML* num vetor de *Struct*. Para além disso, também é função deste ficheiro processar os documentos com extensão 3d identificando os vários vértices contidos neste, elaborando assim um vector de *Point**.

Para que tal fosse possível foram desenvolvidas cinco funções essenciais:

- *parserXML* que verifica se o documento *XML* apresenta um formato correto;
- *readFile* que dado um documento com extensão 3d extrai todos os vértices presentes retornando assim um vector de *Point**;
- *lookFiles* que percorre todo o documento *XML* elaborando um vetor de *Struct*, com o auxílio da função *lookAux*;
- *lookAux* que percorre um *group* do *XML* e extrai a informação correspondente, isto é, retornará um vector de *Struct*. Se o grupo em questão contiver outros groupos dentro de si, essa informação também é processada pelo *lookAux*. Caso a informação a ler seja uma transformação ou uma cor, a função *lookupT* é invocada;

- *lookupT* que processa a informação sempre que é encontrada um transformação (`traslate`, `rotate` ou `scale`) ou uma cor (`color`) criando um *Transform* e adicionando-o à *Struct* que está a ser elaborada pelo *lookAux*.

```
vector<Struct> lookAux(XMLElement*);
vector<Struct> lookFiles(char*);
vector<Point*> readFile(string);
int parseXML(char*);
```

3 Engine

O ficheiro *engine*, tal como referido anteriormente, deve processar a informação de um documento *XML*. Com o auxílio do *parser* essa ação é realizada obtendo-se assim o vetor de *struct* explicado acima.

```
estruturas = lookFiles(argv[1]);
```

Como o *engine* nesta fase deve ser capaz de elaborar um protótipo do Sistema Solar, este foi alterado para assim conseguir lidar com essa informação de forma dinâmica. No entanto, deve continuar a ser possível processar informação de figuras primitivas de uma forma estática.

Assim, foram desenvolvidas duas funções principais:

- void sistemaSolar(Struct);
- void figuraPrimitiva(Struct);

Estas funções são invocadas na *RenderScene* sendo verificado qual o nome do ficheiro de forma a que se possa chamar a função correta para o processo de desenho.

```
for(vector<Struct>::const_iterator f = estruturas.begin();
f != estruturas.end(); f++) {
    Struct s = (*f);
    nf = s.getFile().c_str();
    if(!strcmp("asteroide.3d",nf)  || !strcmp("callisto.3d",nf)
```

```
        || !strcmp("europa.3d",nf)  || !strcmp("ganymede.3d",nf)
        || !strcmp("io.3d",nf)  || !strcmp("jupiter.3d",nf)
        || !strcmp("lua.3d",nf) ||!strcmp("marte.3d",nf)
        || !strcmp("mercurio.3d",nf) ||!strcmp("neptuno.3d",nf)
        || !strcmp("plutao.3d",nf) ||!strcmp("saturno.3d",nf)
        || !strcmp("anel.3d", nf) || !strcmp("sol.3d",nf)
        ||!strcmp("terra.3d",nf) ||!strcmp("titan.3d",nf)
        ||!strcmp("triton.3d",nf) ||!strcmp("urano.3d",nf)
        ||!strcmp("venus.3d",nf))
          sistemaSolar(s);
    else figuraPrimitiva(s);
}
```

De seguida, ambas as funções são apresentadas e explicadas.

3.1 Figuras Primitivas

Tal como o nome sugere, esta função está encarregue de processar uma figura primitiva de forma estática.

É importante referir que esta se encontra rodeada por um *glPushMatrix()* e um *glPopMatrix()* para assim, sempre que for aplicada uma transformação aos eixos para o desenho da figura em questão, ser mantido o eixo de origem como a posição de referência para o desenho da próxima figura.

Antes do desenho da figura em si, é necessário aplicar a cor, `color`, e as transformações dos eixos, mais especificamente `traslate`, `rotate` e `scale`. Posteriormente, os pontos para desenho dos triângulos são obtidos e para cada um destes gerado uma cor aleatória caso não tenha sido definida uma para a figura. Com a geração de cores aleatórias, é possível desenhar os vários triângulos com cores diferentes o que é vantajoso para que se distinguir melhor cada triângulo que constitui a figura.

De seguida é apresentado o código desta função.

```
void figuraPrimitiva(Struct s){

    vector<Transform*> vt = s.getRefit();
    const char* nameTransf;
    float angle, x, y, z;
    int cl=0;
```

```
    glPushMatrix();

//aplicação da cor e de todas as transformações da figura
    for (vector<Transform *>::const_iterator t = vt.begin();
      t != vt.end(); t++) {
        nameTransf = (*t)->Transform::getName().c_str();

        if (!strcmp(nameTransf,"rotate")) {
            angle = (*t)->Transform::getAngle();
        }

        x = (*t)->Transform::getPoint()->Point::getX();
        y = (*t)->Transform::getPoint()->Point::getY();
        z = (*t)->Transform::getPoint()->Point::getZ();

        if (!strcmp(nameTransf,"translate")){
            glTranslatef(x,y,z);
        }
        else if (!strcmp(nameTransf,"rotate")) {
            glRotatef(angle,x,y,z);
        }
        else if (!strcmp(nameTransf,"scale")) {
            glScalef(x,y,z);
        }
        else if (!strcmp(nameTransf,"color")) {
            glColor3f(x,y,z);
            cl=1;
        }
    }

    vector<Point*> vp;
    Point p;

    srand (time(NULL));
    int color=0;
    float a, b, c;
```

```
//desenho da figura
//caso não tenha cor, gerar uma aleatória para cada triângulo
    glBegin(GL_TRIANGLES);
    vp=s.getPoints();
    for (vector<Point *>::const_iterator i = vp.begin();
      i != vp.end(); ++i, color++) {
        p = **i;
        if (cl!= 1 && color % 3 == 0) {
            a = (float) rand() / (float) RAND_MAX;
            b = (float) rand() / (float) RAND_MAX;
            c = (float) rand() / (float) RAND_MAX;

            if (a <= 0.1 && b <= 0.1 && c <= 0.1) a = 1;

            glColor3f(a, b, c);
        }
        glVertex3f(p.getX(), p.getY(), p.getZ());
    }
    glEnd();

    glPopMatrix();
}
```

Como exemplo de utilização desta função foram desenvolvidos três cenários, um básico com apenas transformações, outro de um protótipo de um boneco de neve com transformações e cores associadas e um do Sistema Solar.

O cenário básico possui o seguinte aspeto:

Figura 2: Cenário básico com figuras primitivas

De seguida, segue-se o ficheiro *XML* que produz este cenário. De notar a utilização de todos os tipos de transformações, `translate`, `rotate` e `scale`.

```
<scene>
    <group>
        <translate x="35" y="0" z="0" />
        <scale x="5" y="10" z="15" />
        <models>
                <model file = "plane.3d" />
        </models>
    </group>
    <group>
        <translate x="25" y="0" z="0" />
        <scale x="2" y="1" z="1" />
        <models>
                <model file = "box.3d" />
        </models>
    </group>
    <group>
        <translate x="12" y="0" z="0" />
        <scale x="5" y="5" z="5" />
        <models>
                <model file = "sphere.3d" />
```

```
        </models>
    </group>
    <group>
        <rotate angle="90" x="0" y="0" z="1" />
        <models>
                <model file = "cone.3d" />
        </models>
    </group>
    <group>
        <translate x="-40" y="0" z="0" />
        <models>
                <model file = "torus.3d" />
        </models>
    </group>
</scene>
```

O ficheiro *XML* para o caso do boneco de neve é parecido ao exposto acima. No entanto, prestou-se uma atenção meticulosa à posição de cada uma das figuras primitivas e à cor destas de forma a que formasse o boneco de neve desejado.

```
<scene>
    <group>
        <color x="1.0" y="1.0" z="1.0" />
        <models>
                <model file = "bonecoC1.3d"/>
        </models>
    </group>
    <group>
        <color x="1.0" y="1.0" z="1.0" />
        <translate x="0" y="17" z="0"/>
        <models>
                <model file = "bonecoC2.3d"/>
        </models>
    </group>
    <group>
        <color x="1.0" y="1.0" z="1.0" />
        <translate x="0" y="27" z="0"/>
```

```
        <models>
                <model file = "bonecoC3.3d"/>
        </models>
    </group>
    <group>
        <color x="0.8" y="0.2" z="0.0" />
        <translate x="0" y="27" z="2.75"/>
        <rotate angle="90" x="1" y="0" z="0" />
        <models>
                <model file = "bonecoN.3d"/>
        </models>
    </group>
    <group>
        <color x="0.3" y="0.2" z="0.1" />
        <translate x="1.5" y="28" z="2.75"/>
        <models>
                <model file = "bonecoO1.3d"/>
        </models>
    </group>
    <group>
        <color x="0.3" y="0.2" z="0.1" />
        <translate x="-1.5" y="28" z="2.75"/>
        <models>
                <model file = "bonecoO2.3d"/>
        </models>
    </group>
</scene>
```

Com esse ficheiro é possível gerar o seguinte boneco de neve:

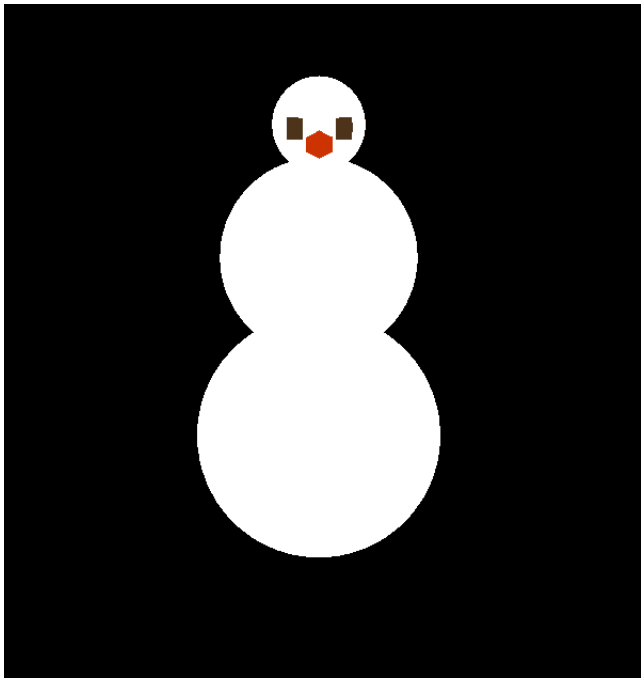

Figura 3: Boneco de neve usando figuras primitivas, transformações e cores

Para a geração do terceiro caso, o Sistema Solar estático, foi fundamental efetuar pesquisas sobre quais as distâncias a que os corpos celestes se encontram uns dos outros e qual o tamanho e cor de cada um destes. Como as medidas reais possuem valores colossais, aplicaram-se valores mais pequenos às distâncias e ao tamanho tendo-se o cuidado que estas permanecessem proporcionais às reais.

Desta forma, obteu-se um protótipo do Sistema Solar estático com o seguinte aspeto:

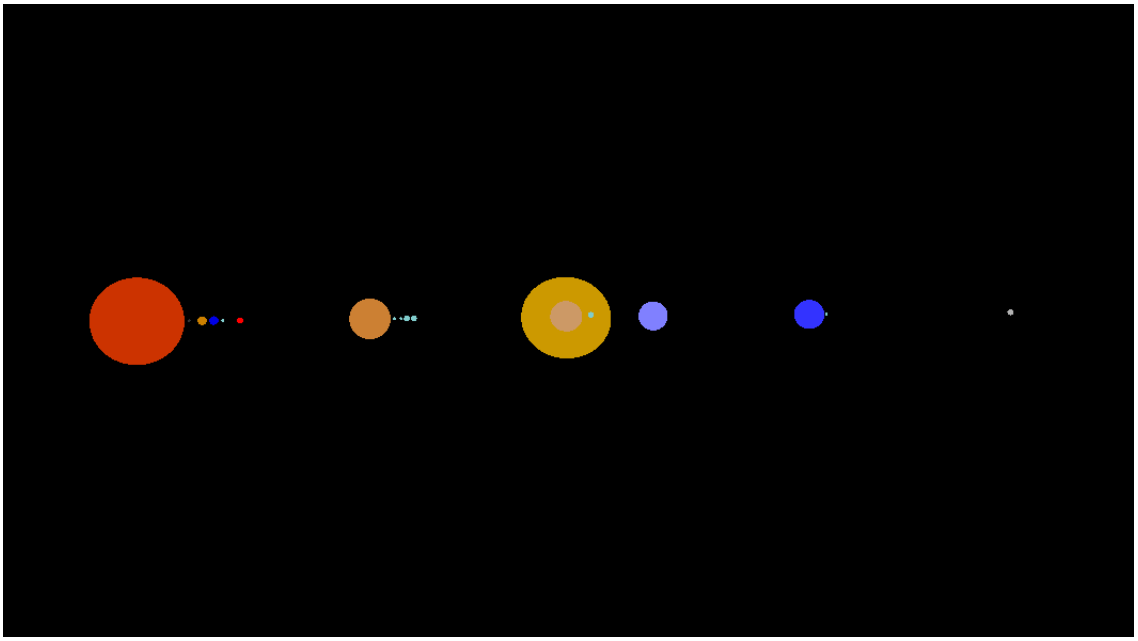

Figura 4: Sistema Solar estático

O ficheiro *XML* para este caso possui o seguinte formato:

```
<scene>
    <group>
        <!--Sol-->
        <color x="0.8" y="0.2" z="0.0" />
        <models>
            <model file="sol2.3d" />
        </models>
    </group>
    <group>
        <!--Mercurio-->
        <color x="0.2" y="0.2" z="0.2" />
        <translate x="35" y="0" z="0" />
        <models>
            <model file="mercurio2.3d" />
        </models>
    </group>
```

```
<group>
    <!--Venus-->
    <color x="0.8" y="0.5" z="0.0" />
    <translate x="44" y="0" z="0" />
    <models>
        <model file="venus2.3d" />
    </models>
</group>
<group>
    <!--Terra-->
    <color x="0.0" y="0.0" z="0.9" />
    <translate x="52" y="0" z="0" />
    <models>
        <model file="terra2.3d" />
    </models>
    <group>
        <color x="0.5" y="0.8" z="0.8" />
        <translate x="6" y="0" z="0" />
        <models>
            <model file="lua2.3d" />
        </models>
    </group>
</group>
<group>
    <!--Marte-->
    <color x="1.0" y="0.0" z="0.0" />
    <translate x="70" y="0" z="0" />
    <models>
        <model file="marte2.3d" />
    </models>
</group>
<group>
    <!--Asteroides-->
    <color x="0.7" y="0.7" z="0.7" />
    <rotate angle="90" x="1" y="0" z="0" />
    <models>
        <model file="asteroide2.3d" />
    </models>
```

```
</group>
<group>
    <!--Jupiter-->
    <color x="0.8" y="0.5" z="0.2" />
    <translate x="159" y="0" z="0" />
    <models>
        <model file="jupiter2.3d" />
    </models>
    <group>
        <color x="0.5" y="0.8" z="0.8" />
        <translate x="16.8" y="0" z="0" />
        <models>
            <model file="io2.3d" />
        </models>
    </group>
    <group>
        <color x="0.5" y="0.8" z="0.8" />
        <translate x="21.3" y="0" z="0" />
        <models>
            <model file="europa2.3d" />
        </models>
    </group>
    <group>
        <color x="0.5" y="0.8" z="0.8" />
        <translate x="25.3" y="0" z="0" />
        <models>
            <model file="ganymede2.3d" />
        </models>
    </group>
    <group>
        <color x="0.5" y="0.8" z="0.8" />
        <translate x="30.3" y="0" z="0" />
        <models>
            <model file="callisto2.3d" />
        </models>
    </group>
</group>
<group>
```

```
    <!--Saturno-->
    <translate x="294" y="0" z="0" />
    <color x="0.8" y="0.6" z="0.4" />
    <models>
        <model file="saturno2.3d" />
    </models>
    <group>
        <color x="0.8" y="0.6" z="0.0" />
        <rotate angle="-70" x="1" y="0" z="0" />
        <models>
            <model file="anel2.3d" />
        </models>
    </group>
    <group>
        <color x="0.5" y="0.8" z="0.8" />
        <rotate angle="-70" x="1" y="0" z="0" />
        <rotate angle="90" x="1" y="0" z="0" />
        <translate x="0" y="3" z="0" />
        <translate x="16.8" y="0" z="0" />
        <models>
            <model file="titan2.3d" />
        </models>
    </group>
</group>
<group>
    <!--Urano-->
    <color x="0.5" y="0.5" z="1.0" />
    <translate x="354" y="0" z="0" />
    <models>
        <model file="urano2.3d" />
    </models>
</group>
<group>
    <!--Neptuno-->
    <color x="0.2" y="0.2" z="1.0" />
    <translate x="462" y="0" z="0" />
    <models>
        <model file="neptuno2.3d" />
```

```
        </models>
        <group>
            <color x="0.5" y="0.8" z="0.8" />
            <translate x="12" y="0" z="0" />
            <models>
                <model file="triton2.3d" />
            </models>
        </group>
    </group>
    <group>
        <!--Plutão-->
        <color x="0.7" y="0.7" z="0.7" />
        <translate x="602" y="0" z="0" />
        <models>
            <model file="plutao2.3d" />
        </models>
    </group>
</scene>
```

3.2 Sistema Solar

De forma a elaborar um Sistema Solar dinâmico, começou-se por desenhar este num projeto à parte com o intuito de ter uma melhor percepção dos comandos a aplicar.

Exemplo com dois corpos celestes (Terra e Lua):

```
//utilização do tempo para as órbitas dos corpos celestes
float time=1, re, gr;
re = glutGet(GLUT_ELAPSED_TIME)/100.f;
gr = (re*360) / (time * 1000);

//Terra + Lua
glPushMatrix();

glColor3f(0.5,0.8,0.8); //definição da cor da órbita da Terra
glBegin(GL_POINTS); //desenho da órbita da Terra com pontos
for(int i=0;i<360;++i) {
    glVertex3f(52 * sin(i),0, 52 * cos(i));
```

```
}
glEnd();

glRotatef(gr*6,0,1,0); //rotação da Terra à volta do Sol
glTranslatef(52,0,0); //posição à qual a Terra está do Sol
glRotatef(gr*6,0,1,0); //rotação da Terra sobre si própria
glColor3f(0.0,0.0,0.9); //definição da cor da Terra
glutSolidSphere(7.834/2,100,100); //desenho da Terra

glColor3f(0.5,0.8,0.8); //definição da cor da órbita da Lua
glBegin(GL_POINTS); //desenho da órbita da Lua com pontos
for(int i=0;i<360;++i) {
    glVertex3f(6 * sin(i),0, 6 * cos(i));
}
glEnd();

glRotatef(gr*20,0,1,0); //rotação da Lua à volta da Terra
glTranslatef(6,0,0); //posição à qual a Lua está da Terra
glRotatef(gr*6,0,1,0); //rotação da Lua sobre si própria
glColor3f(0.5,0.8,0.8); //definição da cor da Lua
glutSolidSphere(2.12/2,100,100); //desenho da Lua

glPopMatrix();
```

Como se pode verificar pelo código apresentado acima, é possível identificar um padrão no mesmo.

É de extrema importância que todo este processo seja efetuado entre um *glPushMatrix()* e um *glPopMatrix()* para que os eixos possam ser novamente retomados ao estado inicial.

Antes do desenho do planeta é necessário desenhar a sua órbita. As órbitas são aplicadas em referência ao ponto (0,0,0), de forma a demonstrar o movimento do planeta à volta do Sol.

De seguida, é aplicada a rotação do planeta à volta do Sol e de forma a que esta seja percetível é necessário aplicar o factor tempo.

Com o eixo em rotação, é possível aplicar a translação para colocar o planeta no local correto. Ao efetuar essa translação, todas as figuras desenhadas serão afetadas pelo rotate anterior obtendo-se assim o movimento do planeta à volta do Sol.

De seguida, é necessário aplicar a rotação do planeta sobre si próprio.

Por fim, define-se a cor do planeta correspondente aplicando a função *glColorf* com os parâmetros corretamente associados seguido pelo desenho do planeta em si.

Desta forma conclui-se que todos os outros planetas seguirão o mesmo raciocínio. No entanto, as luas possuem um comportamento diferente pois são afetadas pela translação e rotação do seu planeta tendo de se ter cuidado no desenho destas.

Com este padrão elaborou-se o algoritmo que constitui a função *Sistema-Solar*. Como este é bastante extenso será abordado por partes.

3.2.1 Desenho da órbita

Tal como referido anteriormente, antes do desenho do planeta em si, é necessário desenhar as órbitas deste à volta do sol para utilizar a origem como referência. A órbita corresponde ao último translate aplicado à figura. Caso a figura seja o próprio sol, um asteroide, o anel de Saturno ou uma lua estas órbitas não são aplicadas. No entanto, o raio das órbitas das luas são obtidas nesta secção para não haver repetição de código.

```
//desenhar órbita dos planetas e obter o raio das órbitas das luas
    if(!sol || !asteroide || !anel){
        for (vector<Transform *>::const_iterator t = vt.begin();
          t != vt.end(); t++) {
            nameTransf = (*t)->Transform::getName().c_str();
            if (!strcmp(nameTransf,"translate")){
                raio = (*t)->Transform::getPoint()->Point::getX();
            }
        }

        if(!lua3d) {
            glColor3f(0.5,0.8,0.8);
            glBegin(GL_POINTS);
            for(int i=0;i<360;++i) {
                glVertex3f(raio * sin(i), 0, raio * cos(i));
            }
            glEnd();
        }
    }
```

3.2.2 Rotação à volta do sol

Neste passo obtem-se o valor que irá ser usado para o cálculo do movimento do corpo celeste à volta do sol para assim ser possível aplicar um *rotate* que utiliza o tempo. Esta informação é obtida através de uma função auxiliar *rotacao* que recebe o nome de um corpo celeste e devolve a velocidade de rotação do mesmo à volta do sol. Com este *rotate* as figuras desenhadas posteriormente irão herdar essas propriedades.

```
//utilização do tempo para as órbitas dos corpos celestes
float time, re, gr;
time=1;
re = glutGet(GLUT_ELAPSED_TIME)/100.f;
gr = (re*360) / (time * 1000);

//aplicar a rotação à volta do sol
if(!lua3d) glRotatef(gr*rotacao(nameFile),0,1,0);
else{
    if (!strcmp(nameFile, "lua.3d"))
        glRotatef(gr * rotacao("terra.3d"), 0, 1, 0);
    else if (!strcmp(nameFile, "io.3d") ||
             !strcmp(nameFile, "europa.3d") ||
             !strcmp(nameFile, "ganymede.3d") ||
             !strcmp(nameFile, "callisto.3d"))
        glRotatef(gr * rotacao("jupiter.3d"), 0, 1, 0);
    else if (!strcmp(nameFile, "titan.3d"))
        glRotatef(gr * rotacao("saturno.3d"), 0, 1, 0);
    else if (!strcmp(nameFile, "triton.3d"))
        glRotatef(gr * rotacao("neptuno.3d"), 0, 1, 0);
}
```

Exemplo da função auxiliar *rotacao*:

```
float rotacao(const char* nameFile){
    float r=0;

    if(!strcmp(nameFile,"sol.3d")) return 0;
    else if(!strcmp(nameFile,"mercurio.3d")) return 10;
    else if(!strcmp(nameFile,"venus.3d")) return 8;
```

```
    else if(!strcmp(nameFile,"terra.3d")) return 6;
    (...)
    else if(!strcmp(nameFile,"plutao.3d")) return 0.1;

    return r;
}
```

3.2.3 Aplicação de transformações e cores

De forma a definir a cor da figura e as transformações dos eixos para o posterior desenho, foi necessário um ciclo que percorresse o vector de transformações, *vector<Transform*> vt* existente na estrutura passada como argumento. De seguida é identificada qual o comando a executar (`traslate`, `rotate`, `scale` ou `color`) e aplica-se a sua respetiva função (*glTranslatef*, *glRotatef*, *glScalef* ou *glColor3f*).

Tal como referido anteriormente, as luas são afetadas pelas translações do seu planeta. Como é nesta secção que se obtém as transformações, decidiu-se tratar essa questão nesta divisão.

```
//aplicar translações/rotações/escalas dos eixos e cor do corpo celeste
int lua=0;
for (vector<Transform *>::const_iterator t = vt.begin();
     t != vt.end(); ++t) {
    nameTransf = (*t)->Transform::getName().c_str();
    if (!strcmp(nameTransf,"rotate")) angle = (*t)->Transform::getAngle();
        x = (*t)->Transform::getPoint()->Point::getX();
        y = (*t)->Transform::getPoint()->Point::getY();
        z = (*t)->Transform::getPoint()->Point::getZ();

        if (!strcmp(nameTransf,"translate")){
            if(lua3d) {
                (...tratamento das luas...)
            }
            else if(anel) {glTranslatef(x,y,z); glRotatef(gr*6, 0, 1, 0);}
            else glTranslatef(x,y,z);
        }
        else if (!strcmp(nameTransf,"rotate")) {
            glRotatef(angle,x,y,z);
```

```
        }
        else if (!strcmp(nameTransf,"scale")) {
            glScalef(x,y,z);
        }
        else if (!strcmp(nameTransf,"color")) {
            glColor3f( x,y,z);
        }
    }
```

No caso de se tratar de um *translate* de uma lua, é necessário que depois do primeiro *translate*, que se refere sempre ao *translate* do seu planeta em relação ao sol, se desenhe a órbita da lua à volta do seu planeta e se aplique a rotação da lua sobre este.

Existe ainda o caso da lua de Saturno, *Titan*, que possui uma rotação à volta do seu planeta ligeiramente diferente devido à existência do anel de Saturno.

Tratamento das luas:

```
if(lua3d) {
        glTranslatef(x, y, z);
        lua++;

        if(lua==1 && !strcmp(nameFile, "titan.3d"))
            glRotatef(gr*6, 0, 1, 0);

        //desenhar órbita da lua e
        //aplicar a sua rotação sobre o planeta correspondente
        if ((lua==1 && strcmp(nameFile, "titan.3d")) ||
            (lua==2 && !strcmp(nameFile, "titan.3d"))) {
                glColor3f(0.5, 0.8, 0.8);
                glBegin(GL_POINTS);
                for (int k = 0; k < 360; ++k) {
                        glVertex3f(raio * sin(k), 0, raio * cos(k));
                }
                glEnd();

                glRotatef(gr * rotacao(nameFile), 0, 1, 0);
        }
}
```

3.2.4 Desenho do corpo celeste e rotação sobre si próprio

Já com os eixos preparados torna-se possível o desenho dos corpos celestes e a aplicação da rotação sobre si próprios.

No caso da figura a desenhar seja um asteroide, utiliza-se um algoritmo para encontrar um local aleatório para o seu desenho, nas posições entre Marte e Júpiter, representando assim a cintura de asteroides principal do Sistema Solar.

Caso seja o anel de Saturno, este não possui rotação sobre si próprio então esta não é aplicada.

O processo de desenho em si é igual ao da fase anterior, através de triângulos cujos vertices foram obtidos através do *parser* e *generator*.

```
//desenhar corpo celeste e fazer rotação sobre si próprio
if(!strcmp(nameFile,"asteroide.3d")){
    for(int j=0; j<75; j++) {
        r=(rand()%8)+100; //r entre 100 e 108
        alpha=rand()%360; //alpha entre 0 e 360

        glPushMatrix();
        glTranslatef(r * sin(alpha), r * cos(alpha), 0);
        glRotatef(gr*6, 0, 1, 0);
        glBegin(GL_TRIANGLES);
        for (vector<Point *>::iterator i = vp.begin(); i != vp.end(); i++) {
            p = **i;
            glVertex3f(p.getX(), p.getY(), p.getZ());
        }
        glEnd();
        glPopMatrix();
    }
}
else {
    if(!anel) glRotatef(gr*6, 0, 1, 0);

    glBegin(GL_TRIANGLES);
    for (vector<Point *>::iterator i = vp.begin(); i != vp.end(); i++) {
        p = **i;
        glVertex3f(p.getX(), p.getY(), p.getZ());
```

```
    }
    glEnd();
}
```

Com este algoritmo é possível obter então o Sistema Solar dinâmico que possui o seguinte aspeto:

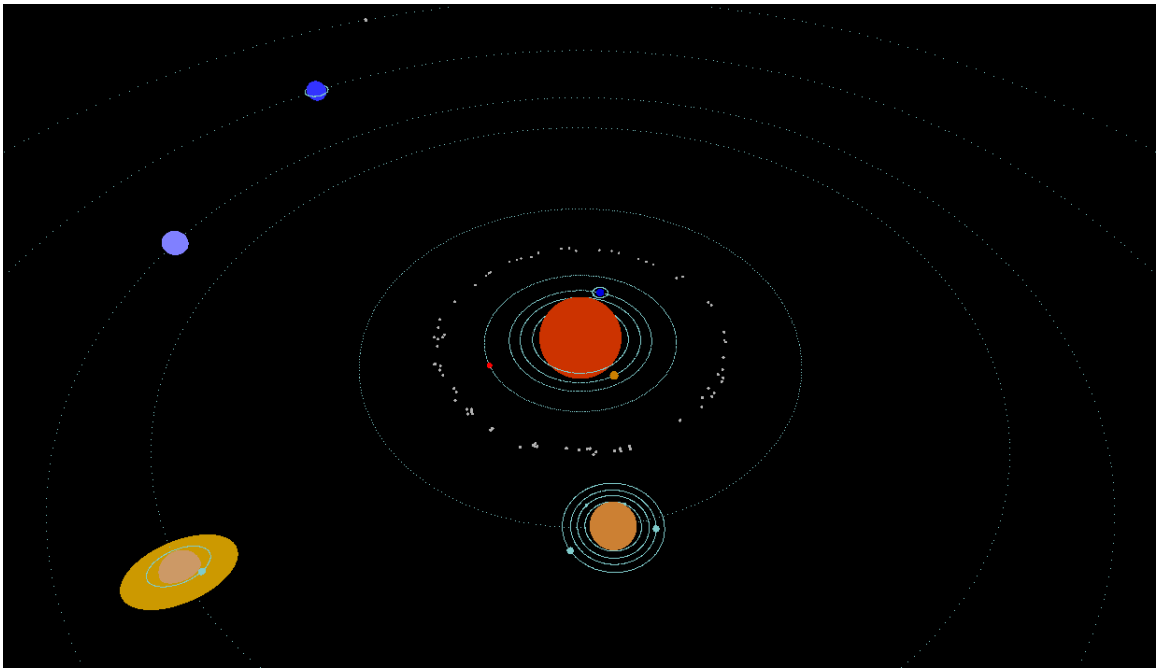

Figura 5: Sistema Solar usando figuras primitivas e transformações

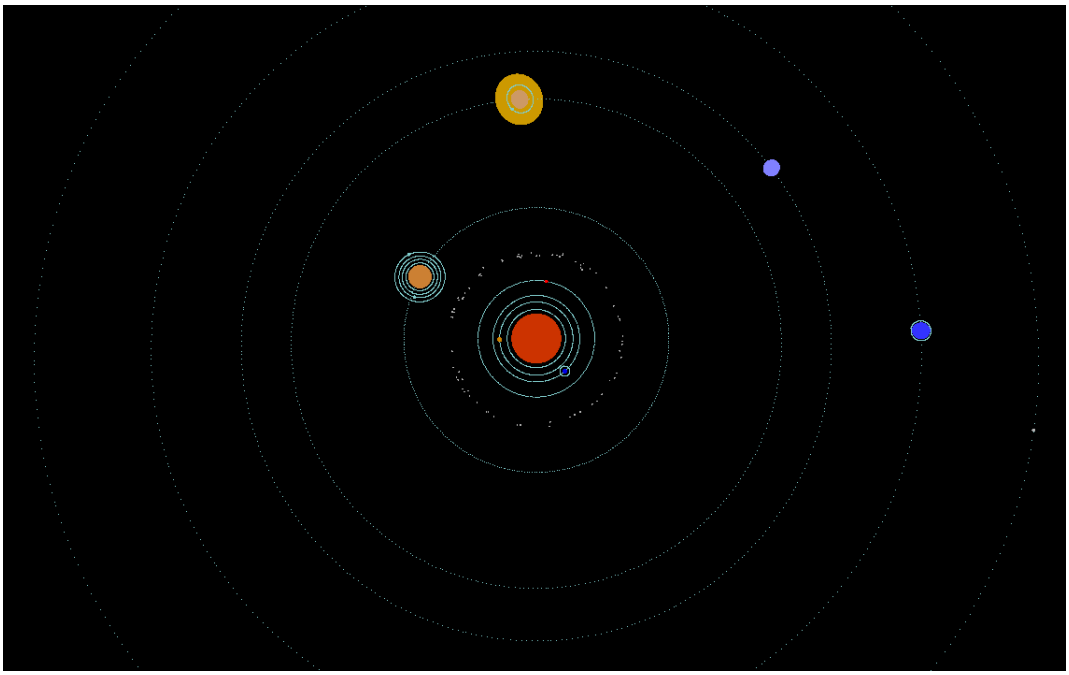

Figura 6: Sistema Solar, uma outra perspectiva

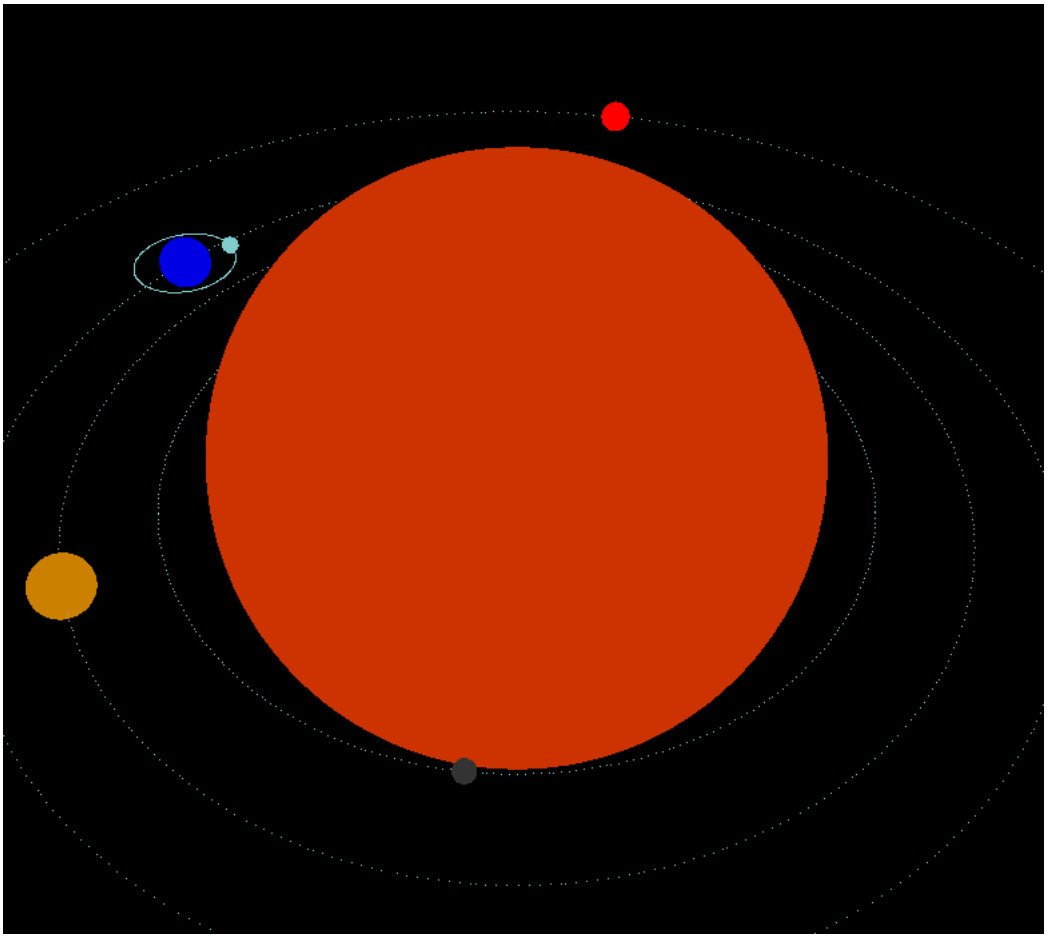

Figura 7: Planetas Rochosos do Sistema Solar e o Sol

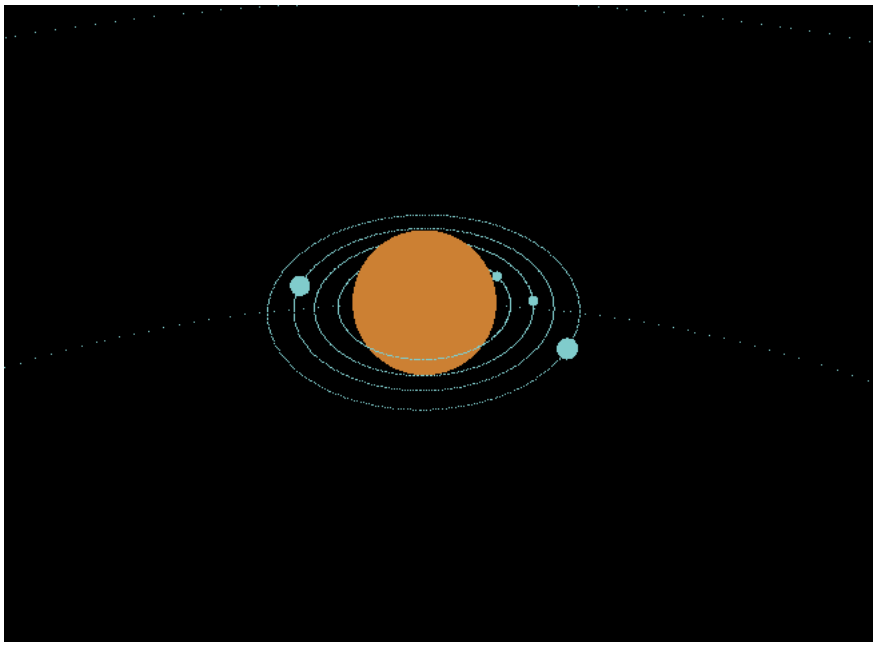

Figura 8: Júpiter

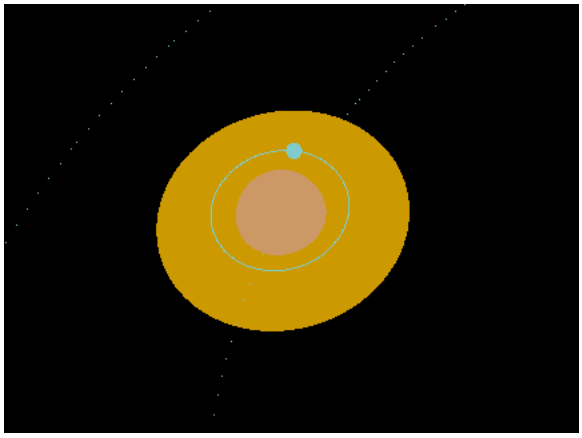

Figura 9: Saturno

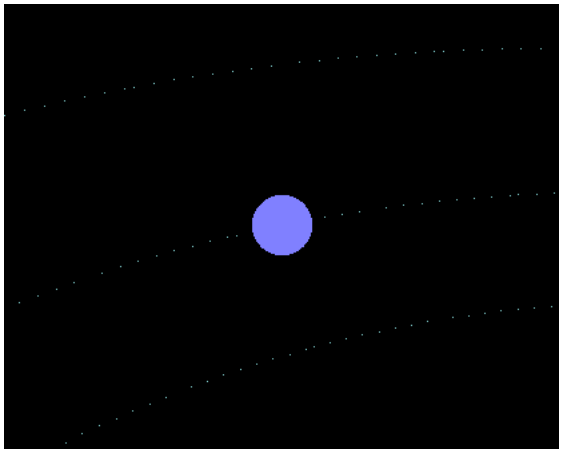

Figura 10: Urano

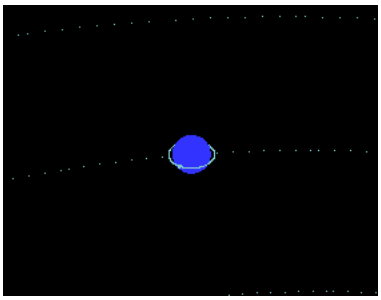

Figura 11: Neptuno

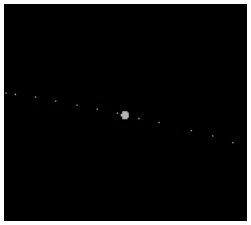

Figura 12: Plutão

O ficheiro XML que criou este cenário é igual ao utilizado no caso estático, no entanto, mudou-se o nome dos ficheiros de *sol2.3d*, por exemplo, para *sol.3d* para que na função *renderScene* estes fossem processados pela função *sistemaSolar* e não pela função *figuraPrimitiva*.

3.3 Câmera

Como no relatório da primeira fase apenas se mencionou a aplicação da câmera e não se procedeu à explicação desta, achou-se conveniente fazê-lo nesta fase.

Considera-se fundamental a movimentação da câmera de forma a melhorar a visualização dos cenários criados. Sendo assim, foi implementado o modo explorador, ou seja, a câmara encontra-se direcionada para um ponto de referência, inicialmente o centro, e movimenta-se numa superficie esférica.

Através do teclado, é possivel mudar a posição da câmera ao longo dessa mesma superfície esférica, o ponto de referência e a distância a este. Para tal, foram desenvolvidas as funções *processKeys* e *processSpecialKeys*.

```
void processKeys(unsigned char key, int xx, int yy) {
    //ESC: Close app.
    if(key == 27){
        exit(0);
    }

//z or x: Translate in the Z axis.
    if(key == 'z'){
        zt++;
        glutPostRedisplay();
    }

    if(key == 'x'){
        zt--;
        glutPostRedisplay();
    }

    //reset the angle and axis.
    if(key=='a'){
        xr=0;
```

```
    yr=0;
    zr=0;
    angle=0;
    glutPostRedisplay();
}

//zoom in
if(key == 'i'){
    r+=2.5;
    glutPostRedisplay();
}

//zoom out
if(key == 'k'){
    r-=2.5;
    glutPostRedisplay();
}

//move camera to the left.
if(key=='u'){
    cx+=inc;
    glutPostRedisplay();
}

//move camera to the right.
if(key=='o'){
    cx-=inc;
    glutPostRedisplay();
}

//move camera upwards.
if(key=='m'){
    cz+=inc;
    if(cz>1.5)
        cz=1.5;
    glutPostRedisplay();
}
```

```
    //move camera downwards.
    if(key=='n'){
        cz-=inc;
        if(cz<-1.5)
            cz=-1.5;
        glutPostRedisplay();
    }

    //q or w: Rotate in the X axis.
    if(key == 'q'){
        angle--;
        xr--;
        glutPostRedisplay();
    }

    if(key == 'w'){
        angle++;
        xr++;
        glutPostRedisplay();
    }

//e or r: Rotate in the Y axis.
    if(key == 'e'){
        angle--;
        yr--;
        glutPostRedisplay();
    }

    if(key == 'r'){
        angle++;
        yr++;
        glutPostRedisplay();
    }


//t or y: Rotate in the Z axis.
    if(key == 't'){
        angle--;
```

```
        zr--;
        glutPostRedisplay();
    }

    if(key == 'y'){
        angle++;
        zr++;
        glutPostRedisplay();
    }

    //Activate line mode.
    if(key == 'l'){
        glPolygonMode(GL_FRONT,GL_LINE);
        glutPostRedisplay();
    }

    //Activate fill mode.
    if(key == 'f'){
        glPolygonMode(GL_FRONT,GL_FILL);
        glutPostRedisplay();
    }

    //Activate point mode.
    if(key == 'p'){
        glPolygonMode(GL_FRONT,GL_POINT);
        glutPostRedisplay();
    }

//realização de uma translação sobre a câmara nos eixos X e Y
void processSpecialKeys(int key, int xx, int yy) {
    if(key == GLUT_KEY_LEFT){
        xt--;
        glutPostRedisplay();
    }

    if(key == GLUT_KEY_RIGHT){
        xt++;
        glutPostRedisplay();
```

```
    }

    if(key == GLUT_KEY_UP){
        yt++;
        glutPostRedisplay();
    }

    if(key == GLUT_KEY_DOWN){
        yt--;
        glutPostRedisplay();
    }
}
```

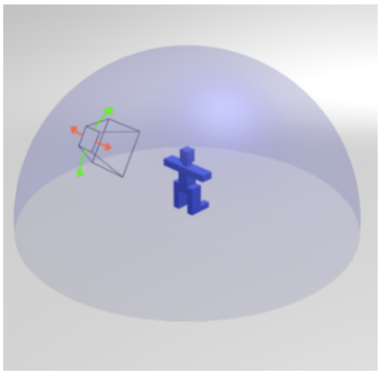

Figura 13: Maquete da Câmera

4 Conclusões

Nesta segunda fase conseguimos desenvolver o nosso conhecimento de OpenGL, especialmente na manipulação de matrizes para o desenho das figuras nas posições pretendidas.

Mais uma vez, cumprimos todos os requisitos propostos, tendo ainda desenvolvido mais uma figura primitiva, o Torus, e explorado o Sistema Solar mais detalhadamente, isto é, atribuindo lhe órbitas, desenhando a cintura de asteroides e as luas dos respetivos planetas. Além disso, foi desenvolvido um Sistema Solar dinâmico que utiliza o tempo para a rotação dos corpos solares.

Nas restantes fases do projeto esperamos melhorar o projeto base que já temos, de forma a otimizá-lo e melhorar a sua apresentação.