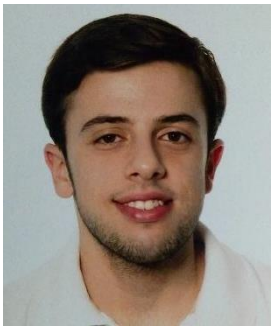
Universidade do Minho

Departamento de Informática

Geometric Transforms

Mestrado Integrado em Engenharia Informática

Computação gráfica



Jaime Leite (A80757)

Bruno Veloso (A78352)

Índice

Índice de figuras

Introdução

Partindo do enunciado que nos foi proposto, apresentaremos neste relatório o trabalho desenvolvido na segunda fase do projeto relativo à unidade curricular de *Computação Gráfica*.

Iremos abordar o método de desenvolvimento do disco, bem como as estruturas usadas para armazenar a informação sobre as transformações (rotações, translações e scales), e informação relativa às figuras em si.

Numa fase seguinte abordaremos o parser de XML desenvolvido pelo grupo. No ficheiro XML, cada nodo contém informação relativa a figuras do sistema solar a desenhar.

Por fim, explicaremos as transformações geométricas, como sendo translates, rotates e scales.

Fase 2

1.Disco

Para ser possível desenhar o disco é necessário saber a distância entre as bordas do mesmo (dist), bem como o raio do centro do disco (raioCentro) e o número de slices.

Para desenhar a figura, foi utilizada a seguinte metodologia: definiu-se um ângulo dAlpha que é igual a (2.0f * M_PI) / slices. São feitos dois ciclos, em que ambos vão desde 0 até ao número de slices.

No primeiro ciclo: por cada uma destas iterações, são desenhados dois triângulos. O primeiro deles é formado pelos pontos P1, P2 e P3. P1 tem como coordenadas (`raioCentro * cosf(alpha), 0.0, raioCentro * sinf(alpha)`); P2 tem como coordenadas (`raioCentro * cosf(pAlpha), 0.0, raioCentro * sinf(pAlpha)`); P3 tem como coordenadas (`(raioCentro + dist) * cosf(pAlpha), 0.0, (raioCentro + dist) * sinf(pAlpha)`), em que alpha é i* dAlpha e pAlpha é alpha + dAlpha, sendo i o número da iteração. O segundo triângulo é formado pelos pontos P3, P4, P1, em que P4 tem como coordenadas (`(raioCentro + dist) * cosf(alpha), 0.0, (raioCentro + dist) * sinf(alpha)`).

O segundo ciclo é em quase tudo semelhante ao primeiro, contudo os ângulos têm valores simétricos aos usados no ciclo anterior.

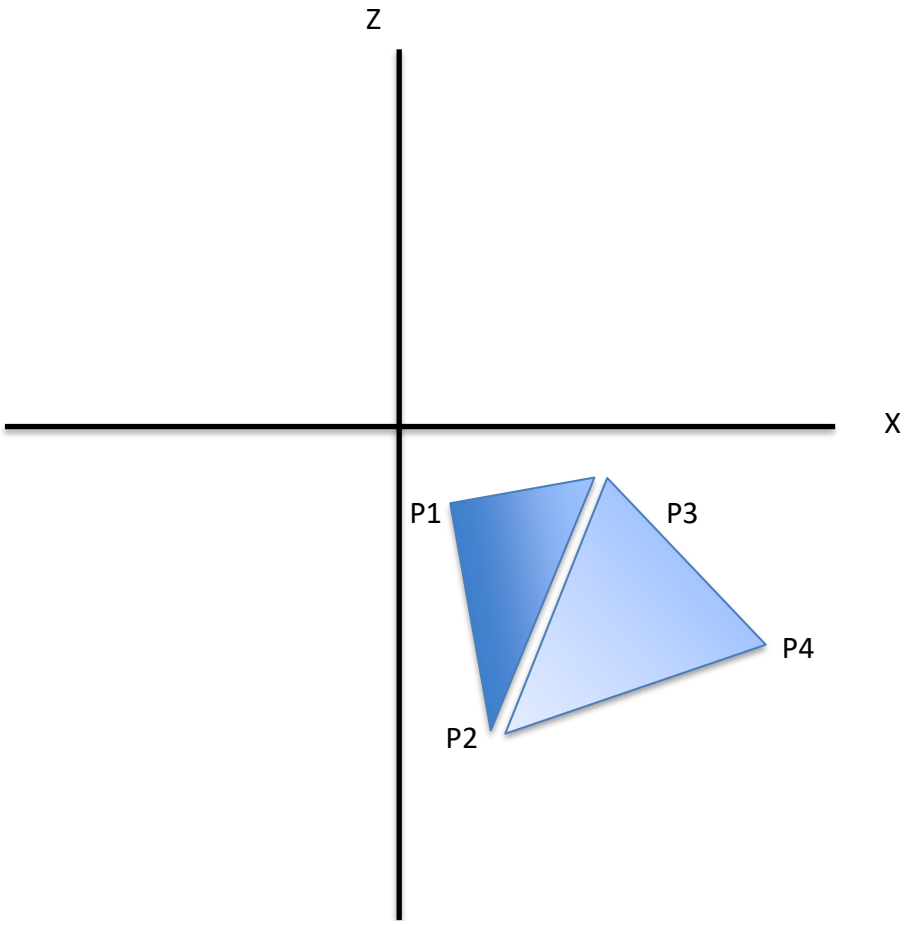


Figura 1: desenho de uma slice do disco

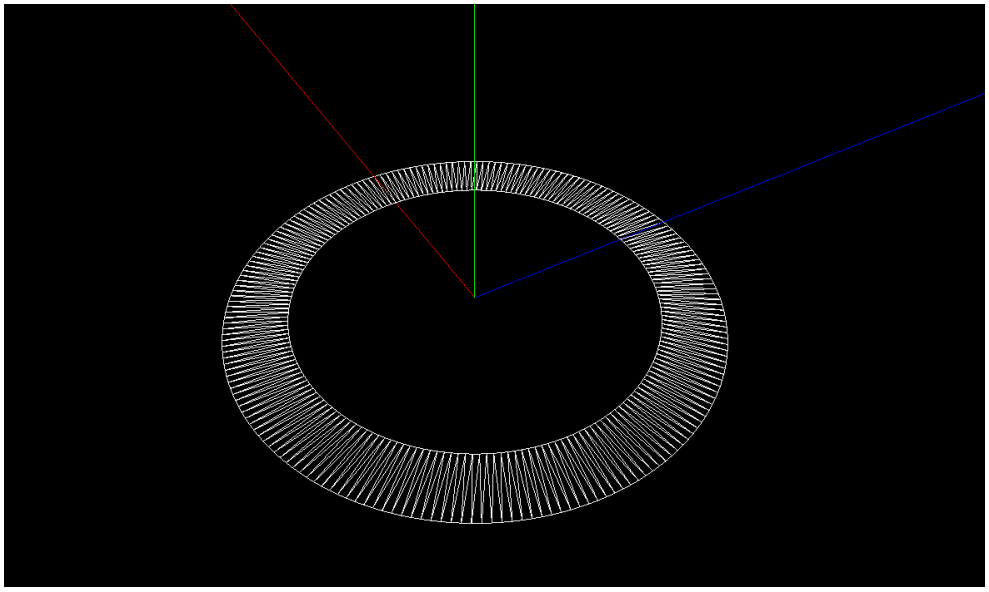


Figura 2: desenho de um disco

2.Estruturas

Para esta fase foi necessário criar novas estruturas de dados para guardar toda a informação proveniente de um parsing do xml.

A única estrutura que se manteve da fase 1 foi o struct Triangulo, no qual guarda informação relativamente a um e um só triângulo, em que a struct é constituída por 9 valores float, x1,y1 e z1 correspondem às coordenadas do primeiro ponto, x2,y2,z2 correspondem às coordenadas do segundo ponto e por fim x3,y3 e z3 correspondem às coordenadas do terceiro ponto.

```
struct Triangulo {
    float x1;
    float y1;
    float z1;

    float x2;
    float y2;
    float z2;

    float x3;
    float y3;
    float z3;
};
```

Figura 3: struct para armazenar um triângulo

```
struct Xml{
    float ang;
    float rot_x;
    float rot_y;
    float rot_z;
    float trans_x;
    float trans_y;
    float trans_z;
    float scale_x;
    float scale_y;
    float scale_z;
    char** file;
    int nfiles;
    struct Triangulo ** fig;
    unsigned int *fig_length;
    struct Xml *child;
};

static struct Xml * documents = NULL;
```

Figura 4: struct para armazenar todos os nodos de Xml (group)

A nova struct criada serve para guardar toda a informação relativamente ao pai e seus filhos. Esta é a struct Xml e guarda informações relativamente às transformações como: um "ang", que representa o ângulo de rotação um "rot_x" que representa a rotação no eixo X; um "rot_y", que representa a rotação no eixo Y e um "rot_z" que representa a rotação no eixo Z. Todos estes 4 valores vão ser passados na tag rotate do xml. Depois guarda informação como "trans_x", "trans_y" e "trans_z", que representam as translações a efetuar no eixo X,Y e Z por esta ordem, esta informação estará na tag translate do xml. Também guarda a informação relativamente ao scale a efetuar nas variáveis "scale_x", "scale_y" e "scale_z" e estes valores estarão na tag scale do Xml.

A struct Xml, para além dos valores das transformações, vai guardar um array de strings "file". Teve de ser um array pois um group pode ter uma tag models com várias tag model dentro, em que cada tag model representa um ficheiro, e para além de guardar os ficheiros guarda também o número de ficheiros guardados em "nfiles".

Esta também tem de guardar o tamanho de cada figura que esteja no array de strings "file" e para isto usa um array de unsigned int chamado "fig_length", em que o tamanho para guardar no "fig_length" estará definido na primeira linha do ficheiro a dar parse (número de triângulos).

É guardado também uma matriz de struct Triangulo chamada "fig" em que esta corresponderá aos triângulos de cada figura (ficheiro em "file"), ou seja fig[0][fig_length[0]] corresponde aos triângulos da file[0] que tem de tamanho fig_length[0].

Por fim tem um array em que vai guardar as struct Xml dos filhos, esta designa-se "child".

3. Parser

Foram estipulados todos os tipos de informação que o Xml iria conter.

A nomenclatura do ficheiro Xml baseia-se numa tag, e uma só, **<scene>**, em que dentro desta tag **<scene>** podemos ter as tags **<group>** que quisermos. Dentro de uma tag **<group>** podemos ter uma tag **<translate>**, uma tag **<rotate>**, uma tag **<scale>** e uma tag **<models>**. Dentro de **<models>** podemos ter várias tags **<model>,** em que cada uma destas conterá um ficheiro para dar parse.

Para além destas tags, dentro de um **<group>** podemos ter outro **<group>**, ou seja, este segundo **<group>** será um filho do primeiro, do qual todas as transformações feitas pelo pai serão adquiridas por este filho. Dentro do filho podemos ter outro filho e assim sucessivamente.

Para dar parse de toda esta informação foi feita uma função que calcula o número de pais que vamos ter para conseguir alocar memória corretamente.

Uma outra função que faz o parse dos pais em que começa numa tag **<group>** e procura pelas tags **<translate>**, **<rotate>**, **<scale>** em que se não tiver **<translate>** ou **<rotate>** então os parâmetros a guardar relativamente a estes serão 0. Já no **<scale>**, se não tiver, os parâmetros a guardar deste serão 1, pois se fossem 0 a imagem não iria aparecer. Depois procura pelo **<models>** e guarda todos os ficheiros encontrados dentro das tags **<model>**. Por fim verifica se tem filhos, se tiver chama uma função para dar parse da informação do filho, em que se este filho tiver um filho, volta a chamar a função recursivamente. Se não tiver filhos, então passa para o próximo **<group>** (pai).

A informação é guardada com sucesso, independentemente se uma tag **<translate>** vem primeiro que uma tag **<rotate>** e vice-versa, ou uma outra qualquer.

Nesta fase o campo "fig" (triângulos da figura) e "fig_length" (número de triângulos de uma figura) da struct Xml ainda estará vazio.

Para isto, existe uma função que recebe um ficheiro, uma struct Xml (pai/filho) e um índice(se um **<models>** tem dois ficheiros então índice é 0 e 1, 0 para o primeiro ficheiro e 1 para o segundo), e vai dar parse do ficheiro e guardar esta informação em "fig_length" conforme o valor lido na primeira linha do ficheiro, ou seja, se este é o ficheiro 3 do **<models>**, por exemplo então fig_lenght[2] = tamanho (índice = 2), e em "fig" vai guardar o parse dos triângulos, ou seja "fig[2][fig_length[2]] = triângulos " (matriz em que linha é o número do ficheiro e coluna que tem triângulos).

No fim de todos estes processos, temos o parsing completamente feito e pronto para as figuras serem desenhadas. Antes de desenhar um nodo pai (com os seus filhos), temos de fazer um glPushMatrix() para guardar o estado atual da matriz e no final do desenho do nodo pai fazemos um glPopMatrix() para voltar ao estado que ficou guardado pelo glPushMatrix().

```
parseFig(const char * file, struct Xml * dc,int index){
FILE * ptr_file = fopen(file, "r");
if (!ptr_file)
    return false;

unsigned ntriangles = 0;
fscanf(ptr_file, "%u\n", &ntriangles);

struct Triangulo * ret =(struct Triangulo*) calloc(ntriangles, sizeof(struct Triangulo));
if (!ret) {
    fclose(ptr_file);
    return false;
}

unsigned i = 0;
for (; i < ntriangles; i++) {
    struct Triangulo tmp;
    int res = fscanf(ptr_file,
        "%f %f %f %f %f %f %f %f %f\n",
        &tmp.x1, &tmp.y1, &tmp.z1,
        &tmp.x2, &tmp.y2, &tmp.z2,
        &tmp.x3, &tmp.y3, &tmp.z3);

    if (res != 9)
        break;
    ret[i] = tmp;
}
fclose(ptr_file);
if (i == ntriangles) {

    dc->fig_length[index] = ntriangles;
    dc -> fig[index] = ret;

}
else {
    free(ret);
}
return i == ntriangles;
```

Figura 5: parse de um ficheiro txt

```
void parseXml(char* file, struct Xml ** dc, int size) {

    tinyxml2::XMLDocument doc;
    doc.LoadFile(file);
    int i = 0;
    int z = 0;
    const char * name;
    float x;

    if (doc.ErrorID() == 0) {
        tinyxml2::XMLElement *a;
        tinyxml2::XMLElement *group;
        tinyxml2::XMLElement *aux;

        a = doc.FirstChildElement("scene");

        struct Xml * ret = (struct Xml*) calloc(size, sizeof(struct Xml));
        group = a->FirstChildElement("group");

        while (group) {
            tinyxml2::XMLElement *translate;
            tinyxml2::XMLElement *rotate;
            tinyxml2::XMLElement *models;
            tinyxml2::XMLElement *model;
            tinyxml2::XMLElement *scale;

            translate = group->FirstChildElement("translate");
            if (translate) {
                name = translate->Attribute("X");
                if (name) ret[z].trans_x = atof(name);
                name = translate->Attribute("Y");
                if (name) ret[z].trans_y = atof(name);
                name = translate->Attribute("Z");
                if (name) ret[z].trans_z = atof(name);

            }
```

Figura 6: parse do ficheiro Xml, <group> pais

```
    rotate = group->FirstChildElement("rotate");
    if (rotate) {
        name = rotate->Attribute("angle");
        if (name) ret[z].ang = atof(name);
        name = rotate->Attribute("axisX");
        if (name) ret[z].rot_x = atof(name);
        name = rotate->Attribute("axisY");
        if (name) ret[z].rot_y = atof(name);
        name = rotate->Attribute("axisZ");
        if (name) ret[z].rot_z = atof(name);
    }

    scale = group->FirstChildElement("scale");
    ret[z].scale_x = 1;
    ret[z].scale_y = 1;
    ret[z].scale_z = 1;
    if (scale) {
        name = scale->Attribute("X");
        if (name) ret[z].scale_x = atof(name);
        name = scale->Attribute("Y");
        if (name) ret[z].scale_y = atof(name);
        name = scale->Attribute("Z");
        if (name) ret[z].scale_z = atof(name);
    }

    models=group->FirstChildElement("models");
    if (models) model = models->FirstChildElement("model");
    i = 0;
    ret[z].file = (char**)(malloc(5 * sizeof(char*)));
```

Figura 7: continuação do parse do Xml

```
        while (model) {

            name = model->Attribute("file");
            if (name) {
                ret[z].file[i]=strdup(name);
                i++;
            }
            model = model->NextSiblingElement("model");
        }

        ret[z].nfiles = i;

        aux = group->FirstChildElement("group");
        if (aux) {
            struct Xml * ret2 = (struct Xml*) calloc(3, sizeof(struct Xml));
            ret[z].child=parseChild(file, aux, ret2 , 0);
        }

        group = group->NextSiblingElement("group");
        z++;

    }
    *dc = ret;
}
```

Figura 8: continuação do parse de Xml

```
struct Xml * parseChild(char* file, tinyxml2::XMLElement *group, struct Xml *n,int index) {
    tinyxml2::XMLDocument doc;
    doc.LoadFile(file);
    int i = 0;
    const char * name;
    float x;
    if (doc.ErrorID() == 0) {
        tinyxml2::XMLElement *aux;

        tinyxml2::XMLElement *translate;
        tinyxml2::XMLElement *rotate;
        tinyxml2::XMLElement *models;
        tinyxml2::XMLElement *model;
        tinyxml2::XMLElement *scale;

        if (group) {
            translate = group->FirstChildElement("translate");

            if (translate) {

                name = translate->Attribute("X");
                if (name) n[index].trans_x = atof(name);
                name = translate->Attribute("Y");
                if (name) n[index].trans_y = atof(name);
                name = translate->Attribute("Z");
                if (name) n[index].trans_z = atof(name);

            }

            rotate = group->FirstChildElement("rotate");
            if (rotate) {

                name = rotate->Attribute("angle");
                if (name) n[index].ang = atof(name);
                name = rotate->Attribute("axisX");
                if (name) n[index].rot_x = atof(name);
                name = rotate->Attribute("axisY");
                if (name) n[index].rot_y = atof(name);
                name = rotate->Attribute("axisZ");
                if (name) n[index].rot_z = atof(name);
            }
```

Figura 9: parse Xml para o filho

```
            scale = group->FirstChildElement("scale");
            n[index].scale_x = 1;
            n[index].scale_y = 1;
            n[index].scale_z = 1;
            if (scale) {

                name = scale->Attribute("X");
                if (name) n[index].scale_x = atof(name);
                name = scale->Attribute("Y");
                if (name) n[index].scale_y = atof(name);
                name = scale->Attribute("Z");
                if (name) n[index].scale_z = atof(name);
            }

            models = group->FirstChildElement("models");
            if (models) model = models->FirstChildElement("model");
            i = 0;
            n[index].file = (char**)(malloc(5 * sizeof(char*)));

            while (model) {

                name = model->Attribute("file");
                if (name) {
                    n[index].file[i] = strdup(name);
                    i++;
                }
                model = model->NextSiblingElement("model");
            }

            n[index].nfiles = i;

            aux = group->FirstChildElement("group");
            if (aux) n=parseChild(file, aux, n, index + 1);
            return n;

        }
    }
}
```

Figura 10: continuação do parse Xml para o filho

```
int nFather(char* file) {

    tinyxml2::XMLDocument doc;
    doc.LoadFile(file);
    int z = 0;

    if (doc.ErrorID() == 0) {
        tinyxml2::XMLElement *a;
        tinyxml2::XMLElement *group;

        a = doc.FirstChildElement("scene");

        group = a->FirstChildElement("group");

        while (group) {

            group = group->NextSiblingElement("group");
            z++;

        }
    }
    return z;
}
```

Figura 11: função que calcula o número de pais no Xml

4. Exemplo da figura da fase 2

```
<scene>
      <group>
            <translate X=1 />
            <models>
                  <model file="sphere.3d" />
            </models>
            <group>
                  <translate Y=1 />
                  <models>
                        <model file="cone.3d" />
                  </models>
            </group>
      </group>
</scene>
```

Figura 12: exemplo de uma estrutura Xml do enunciado

Um dos exemplos do enunciado proposto é este em que se encontra um pai com um filho, no qual o filho recebe todas as transformações do pai.

A forma como esta informação vai ser guardada será (tendo um array de struct Xml chamado documents):

documents[0].ang=0

documents[0].rot_x=0

documents[0].rot_y=0

documents[0].rot_y=0

documents[0].trans_x=1

documents[0].trans.y=0

documents[0].trans.z=0

documents[0].scale_x=1

documents[0].scale_y=1

documents[0].scale_z=1

documents[0].nfiles=1

documents[0].file[0] = "sphere.3d"

documents[0].fig_length[0]= (primeira linha sphere.3d)

documents[0].fig[0][fig_length[0]] = Triângulos de sphere.3d (fig_length[0] número de triângulos)

Filho do pai 0:

documents[0].child[0].ang=0

documents[0].child[0].rot_x=0

documents[0].child[0].rot_y=0

documents[0].child[0].rot_y=0

documents[0].child[0].trans_x=0

documents[0].child[0].trans.y=1

documents[0].child[0].trans.z=0

documents[0].child[0].scale_x=1

documents[0].child[0].scale_y=1

documents[0].child[0].scale_z=1

documents[0].child[0].nfiles=1

documents[0].child[0].file[0] = "cone.3d"

documents[0].child[0].fig_length[0]= (primeira linha cone.3d)

documents[0].child[0].fig[0][fig_length[0]] = Triângulos de sphere.3d (fig_length[0] número de triângulos)

Ou seja, no final deste processo teremos uma esfera centrada em (1,0,0) e um cone centrado em (1,1,0), pois apesar de este cone só ter um translate de Y=1, recebe a transformação do pai que era um translate de X=1.

5. Escalas

As escalas presentes na tabela seguinte serviram para nos guiar no desenho dos planetas.

Figura	Distância real ao Sol (Km)	Raio real da figura (Km)	Escala usada para todos os eixos (relativamente a uma esfera de tamanho 80)
Sol	0	695 700	70
Mercúrio	57 910 000	2 439.7	12
Vénus	108 200 000	6 051.8	18
Terra	149 600 000	6 371	25
Lua	149 600 000	1 737.1	5
Marte	227 900 000	3 389.5	22
Júpiter	778 500 000	69 911	35
Saturno	1 429 000 000	58 232	27
Úrano	2 871 000 000	25 362	28
Neptuno	4 495 000 000	24 622	26

6. Ficheiro Xml

```
<scene>
  <group>
    <scale X="70" Y="70" Z="70"/>
    <models>
      <model file="..\..\..\Fig_files\sphere.txt" />
    </models>
  </group>

  <group>
    <scale X="12" Y="12" Z="12"/>
    <translate Z="250" />
    <models>
      <model file="..\..\..\Fig_files\sphere.txt" />
    </models>
  </group>

  <group>
    <translate Z="400" />
    <scale X="18" Y="18" Z="18"/>
    <models>
      <model file="..\..\..\Fig_files\sphere.txt" />
    </models>
  </group>

  <group>
    <translate Z="550" />
    <scale X="25" Y="25" Z="25"/>
    <models>
      <model file="..\..\..\Fig_files\sphere.txt" />
    </models>
    <group>
      <translate X="-3" />
      <scale X="0.20" Y="0.20" Z="0.20"/>
      <models>
        <model file="..\..\..\Fig_files\sphere.txt" />
      </models>
    </group>
  </group>
```

Sol com escala de 70

Mercúrio com escala de 12 e um translate no eixo do Z de 250

Vénus com uma escala de 18 e um translate de 400 no eixo do Z

Terra com uma escala de 25 e um translate de 550 no eixo do Z; Lua com uma escala de 0.20 (escala real de 0.20 * 25 = 5) e um translate de -3 (valor real: -3 * 25 = -75) no eixo do X.

Figura 13: Xml representativo do sistema solar

Marte com escala de 22 e translate de 700 no eixo do Z

Júpiter com escala de 35 e translate de 850 no eixo do Z

Saturno com escala de 27 e translate de 1050 no eixo do Z; disco de Saturno com escala de 0 (escala real 27 que vem de Saturno) e um ângulo de 60 graus no plano XY

Figura 14: continuação do Xml representativo do sistema solar

Urano com escala de 28 e translate de 1250 no eixo do Z

Neptuno com escala de 26 e translate de 1400 no eixo do Z

Figura 15: continuação do Xml representativo do sistema solar

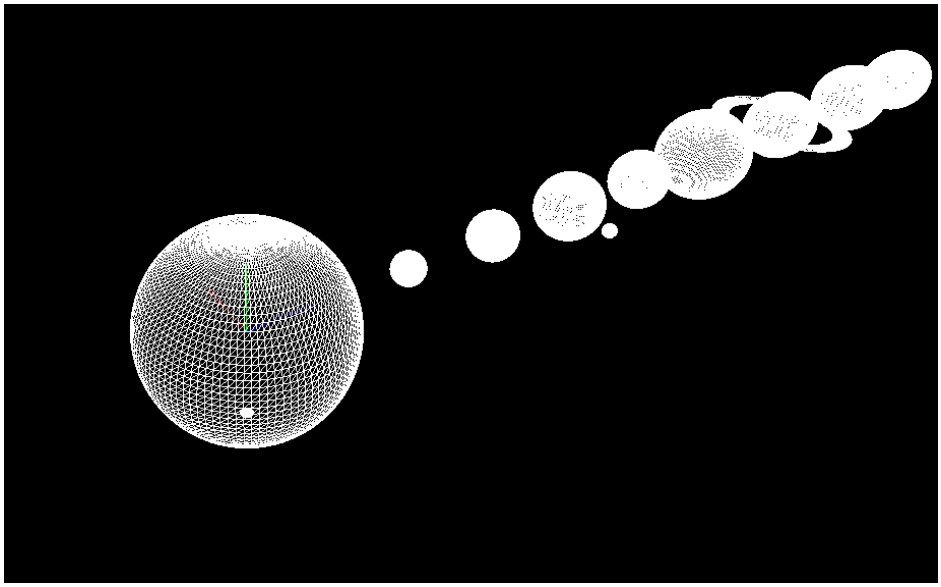


Figura 16: desenho do sistema solar

Conclusão

O grupo considera que alcançou os objetivos propostos para a segunda fase do projeto. Foram desenhados os planetas do sistema solar, bem como o sol e a lua. Para serem visíveis as distâncias entre os diferentes planetas, foram aplicados translates sobre as esferas (que representam esses mesmos planetas).

Foi cumprida a estruturação do Xml, com a inclusão das tags **<scene>**, **<group>**, **<models>, <model>**,**<translate>**,**<rotate>** e **<scale>**. Desenvolvemos estruturas capazes de armazenar triângulos, tipos de transformações geométricas e ficheiros de onde vai ser lida a informação.

Esta fase será importante para as fases subsequentes do projeto, com conhecimento adquirido na estruturação de ficheiros Xml e leitura da informação aí contida.