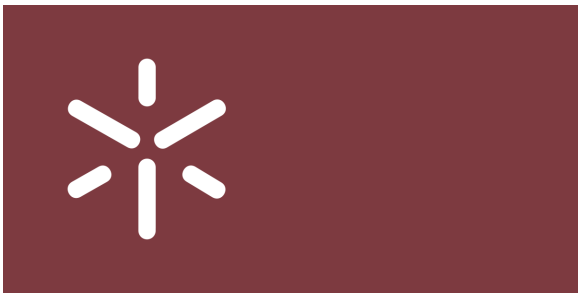

Universidade do Minho

Fase 2 - Transformações Geométricas

Trabalho Prático de Computação Gráfica

Mestrado Integrado em Engenharia Informática

João Amorim A74806

02 de Abril de 2021

1 Introdução

No âmbito da Unidade Curricular de Computação Gráfica, foi proposto que se desenvolvesse um cenário baseado em gráficos 3D. Este projeto consiste em 4 fases, sendo o objetivo desta segunda a criação de cenas hierárquicas usando transformações geométricas.

Para esta fase irão ser feitas modificações ao Engine, de modo a que cumpra com os requisitos propostos, sendo o objetivo final o de obter um modelo do Sistema Solar, fazendo uso das já mencionadas transoormações geométricas. Será também adicionada uma primitiva gráfica no Generator para melhorar o aspeto da cena final.

Este relatório tem como objetivo descrever toda a aplicação, ajudando a executar a mesma, contendo obviamente uma descrição de todo o trabalho desenvolvido.

2 Arquitectura da Solução

Nesta segunda fase, continuamos a dividir a aplicação em duas partes, Generator e Engine. No entanto, é necessário apresentar as Transformações Geométricas que serão abordadas nesta fase:

2.1 Transformações Geométricas

2.1.1 Rotação

A rotação de uma figura geométrica requer a fixação de, pelo menos, um ponto num eixo à escolham, dando-se a mesma em torno desse ponto, não havendo qualquer alteração das dimensões do objeto sobre o qual se dá a rotação. Um exemplo de uma rotação no eixo Z, de um objeto com coordenadas (x,y,z), sendo as novas coordenadas (x',y',z'):

```
x' = x * cos(angle) + y * sin(angle)
y' = x * sin(angle) + y * cos(angle)
z' = z

Sendo angle o ângulo da rotação. Em termos de definição matricial, obtemos o segu
```

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Figure 1: Representação matricial relativa à rotação sobre o eixo do Z.

2.1.2 Translação

A translação desloca um objeto segundo uma direção, sentido e comprimento (vetor), sem qualquer alteração das dimensões do objeto sobre o qual se dá a translação. Sendo as coordenadas de um objeto (x,y,z) e o vetor de translação (vx, vy, vz), as novas coordenadas (xr,yr,zr) são:

```
xr= x + vx
yr = y + vy
zr = z + vz
```

Em termos de definição matricial, obtemos o seguinte:

$$\begin{bmatrix} x_T \\ y_T \\ z_T \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Figure 2: Representação matricial relativa à translação.

2.1.3 Escala

A escala é definida como a razão entre as medidas reais de um objeto e as medidas de desenho. Nesta UC, a aplicação de uma escala a uma figura, altera o tamanho da mesma, havendo assim a multiplicação das coordenadas da mesma por fatores (ex, ey, ez). Um objeto que tenha como coordenadas (x,y,z), após aplicação de uma escala passa a ter as coordenadas (xs,ys,zs) de acordo com:

```
xs = x * ex
ys = y * ey
zs = z * ez
```

$$\begin{bmatrix} x_s \\ y_s \\ z_s \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Figure 3: Representação matricial relativa à escala.

2.2 Generator

Para esta fase, foi necessário adicionar uma primitiva gráfica para a construção do Sistema Solar.

Esta primitva é o Torus e vai servir para representar o anel de Saturno.

Tal como na primeira fase, o Generator cria um ficheiro 3d que contêm os pontos necessários à criação do Torus.

2.2.1 Torus

Esta primitiva apresenta um formato semelhante à de uma câmara de pneu. Pode ser definido como o lugar geométrico tridimensional formado pela rotação de uma superfície circular plana de raio r, em torno de uma circunferência de raio R.

A construção do Torus baseia-se então nos dois raios existentes, isto é, são usadas duas circunferências para o seu desenho, uma para a definição dos pontos internos, sendo o primeiro raio considerado como sendo o raio interno, e outra para definição dos pontos externos, sendo o segundo considerado como sendo o raio externo.

Para a criação do Torus, é ainda necessário definir os eixos responsáveis pela definição das duas circunferências. Assim sendo, os eixos X e Y definem a circunferência interior de raio R e os eixos X,Y e Z a exterior de raio r. As próximas figura demonstram o que foi dito anteriormente:

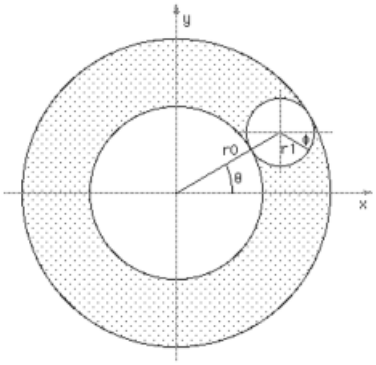

Figure 4: Representação do formato do Torus. r0 corresponde a R e r1 ao r.

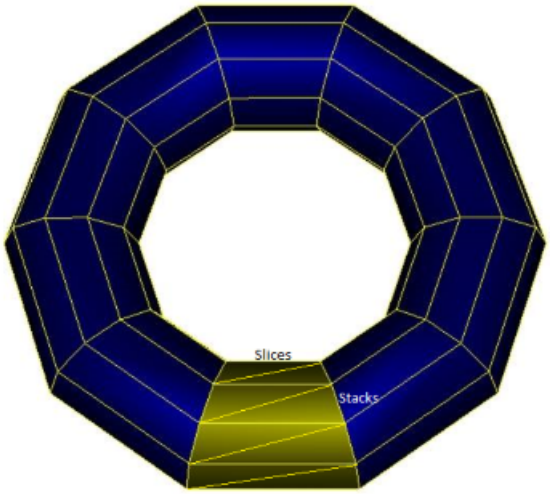

Figure 5: Construção do Torus

De maneira a percorrer as circunferências definidas, usamos os parâmetros slices e stacks para dividir a execução em diferentes partes com amplitudes shiftT e shiftP:

```
shiftT = 2* M_PI / slices
shiftP = 2* M_PI / stacks
```

Com a utilização das funções sin e cos obtemos os pontos que compões a triangulação de cada circunferência. Os pontos de refrência de amplitude têm como coordenadas (x1,y1,z1) e, após o deslocamento, (x2,y2,z2), sendo estes os ponto delimitadores do anel a ser descrito, isto é, uma slice. Para a construção de uma stack, os pontos delimitadores são (x3,y3,z3) e (x4,y4,z4):

```
x1 = cos(theta) * (raioI + raioE * cos(phi));
y1 = sin(theta) * (raioI + raioE * cos(phi));
z1 = raioE * sin(phi);

x2 = cos(theta + shiftT) * (raioI + raioE * cos(phi));
y2 = sin(theta + shiftT) * (raioI + raioE * cos(phi));
z2 = raioE * sin(phi);

x3 = cos(theta + shiftT) * (raioI + raioE * cos(phi + shiftP));
y3 = sin(theta + shiftT) * (raioI + raioE * cos(phi + shiftP));
```

```
z3 = raioE * sin(phi + shiftP);

x4 = cos(theta) * (raioI + raioE * cos(phi + shiftP));
y4 = sin(theta) * (raioI + raioE * cos(phi + shiftP));
z4 = raioE * sin(phi + shiftP);
```

Desta maneira conseguimos percorrer tanto a circunferência interna ao adicionar shiftP, como a externa ao adicionar shiftT. Theta e phi representam o ângulo interno e externo, respetivamente.

Com a criação do ciclo, é feita a iteração de cada circunferência onde i corresponde à slice em questãp e j corresponde à stack. No final de cada iteração da stack, incrementa-se phi de forma a que seja formado um anel. No final de cada anel, incrementa-se theta para que se dê inicio à construção do anel seguinte até completar o torus.

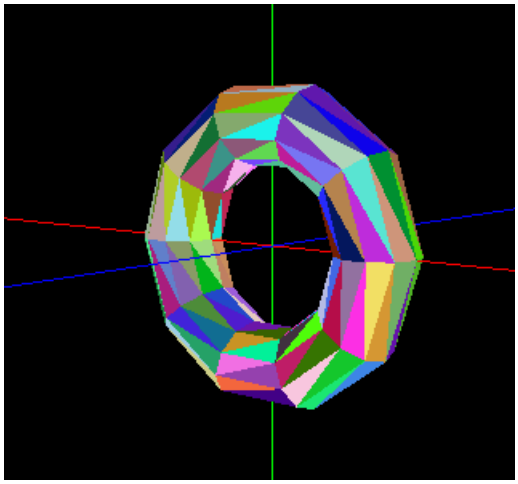

Figure 6: Resultado final do Torus obtido

2.3 Engine

Será aqui no Engine que ocorrerão as maiores alterações, quando comparado com a primeira fase.

Como primeira observação, reparei que na primeira fase do trabalho, para correr o Engine com o nome do ficheiro de uma das primitivas passado como argumento, era ncessário introduzir todo o Path desde a pasta Engine, algo como "../Scenes"+ Nome.xml. Esta foi então a primeira alteração

feita ao engine, permitindo que seja apenas necessário introduzir o nome do ficheiro XML que se pretende ler.

2.3.1 Classes Auxiliares

Antes de entrarmos na nova versão do parser do XML, é necessário explicar o conceito de Operations utilizado nesta fase.

Em primeiro lugar, foram criados ficheiros auxiliares correspondentes a novas classes, em que cada uma destas corresponderá a uma Operation. Estas não só possuem métodos para definir e mostrar valores sobre os objetos que definem, como implementam métodos para execução dos diversos comandos GL que serão necessários. Isto é possível com a criação da classe Operation que invoca o método run, que por sua vez é implementado nas classes já mencionadas. A criação de todas estas classes permitiu que não só o código do Engine ficasse mais claro, como o de toda a aplicação.

As classes criadas que representam uma Operation correspondem a:

- Color - Cria um novo objeto Color com getters e setters. Run executa o comando glColor3f().

```
 #include "Operation.h"

class Color: public Operation {
    float r, g, b;
public:
    Color();
    Color(float r, float g, float b);
    float getR() {return this->r;}
    float getG() {return this->g;}
    float getB() {return this->b;}
    void setR(float x){this->r=r;}
    void setG(float y){this->g=g;}
    void setB(float z){this->b=b;}
    void run();
};
```

- Draw - Run recorre aos comandos GL_TRIANGLES E glVertex3f para preencher o vetor de Point's já utilizado na 1ª fase.

```
#include <vector>
#include "Point.h"
#include "Operation.h"

class Draw: public Operation{

    std::vector<Point> vertices;

public:
 Draw(std::vector<Point> vertices){
     this->vertices=vertices;
};

void run();

};
```

- Point - Cria um novo objeto Point com getters e setters. Já foi utilizado na 1ª fase com o nome de Vértice na 1ª fase, tendo nesta sido passado para uma classe separada.

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

public:
    Point();
    Point(float a, float b, float c);
    float getX() { return this->x; }
    float getY() { return this->y; }
    float getZ() { return this->z; }
    void setX( float b) { this->x = b; }
    void setY( float b) { this->y = b; }
    void setZ( float b) { this->z = b; }

};
```

- Pop - Run executa o comando glPopMatrix();

```
#include "Operation.h"

class Pop: public Operation {

public:
    Pop();
    void run();
};
```

- Push - Run executa o comando glPushMatrix();

```
class Push: public Operation{

public:
    Push();
    void run();
};
```

- Rotation - Cria um novo objeto Rotate com getters e setters. Run executa o comando glRotatef().

```
#include "Operation.h"

class Rotation : public Operation {
    float angle, x, y, z;

public:
    Rotation();
    Rotation(float angle, float x, float y, float z);
    float getAngle() { return this->angle; }
    float getX() { return this->x; }
    float getY() { return this->y; }
    float getZ() { return this->z; }
    void setAngle(float angle) { this->angle = angle; }
    void setX(float x) { this->x = x; }
    void setY(float y) { this->y = y; }
    void setZ(float z) { this->z = z; }
```

```
    void run();
};
```

- Scale - Cria um novo objeto Scale com getters e setters. Run executa o comando glScalef().

```
#include "Operation.h"

class Scale : public Operation {
    float x, y, z;

public:
    Scale();
    Scale(float x, float y, float z);
    float getX() { return this->x; }
    float getY() { return this->y; }
    float getZ() { return this->z; }
    void setX(float x) { this->x = x; }
    void setY(float y) { this->y = y; }
    void setZ(float z) { this->z = z; }
    void run();
};
```

- Translacao - Cria um novo objeto Translacao com getters e setters. Run executa o comando glTranslatef().

```
#include "Operation.h"

class Translacao: public Operation{
    float x, y, z;

public:
    Translacao();
    Translacao(float x, float y, float z);
    float getX() {return this->x;}
    float getY() {return this->y;}
    float getZ() {return this->z;}
    void setX(float x){this->x=x;}
```

```
    void setY(float y){this->y=y;}
    void setZ(float z){this->z=z;}
    void run();
};
```

2.3.2 Parsing do XML

Enquanto que na primeira fase, o processo da leitura do ficheiro XML era muito simples, sendo preciso apenas encontrar o nome do ficheiro 3d a ler, nesta fase é necessário interpretar, guardar e executar as diversas transformações geométricas/operações, tendo em conta uma hierarquia especifica. Em relação ao parser, continuou a ser utilizado o TinyXML2.

Em primeiro lugar, temos um novo vetor:

```
vector<Operation*> ops;
```

Figure 7: Vetor que guarda as diferentes operações

Este vetor vai guardar não só as Transformações Geométricas como todas as Operations que vão ter que ser realizadas, tais como a definição da Color, o preenchimento do vetor com os vértices a desenhar, entre outros.

Aquele que era o método que ia buscar o nome do ficheiro 3d, passa agora a introduzir a leitura do XML num grupo. Este grupo por sua vez irá ser passado como argumento ao método que irá efetuar o parsing do ficheiro XML.

```
void readXML(string fich) {
    string s = "../Scenes/";
    string file = s + fich;
    XMLDocument doc;

    if (!(doc.LoadFile(file.c_str()))){

        XMLElement* scene = doc.FirstChildElement("scene");
        XMLElement* group = scene->FirstChildElement("group");

        xmlParserTransf(group);
    }
    else {
        cout << "Ficheiro XML não foi encontrado" << endl;
    }
}
```

Figure 8: Método readXML()

O método xmlParserTransf(XMLElement* group) tem como objetivo ler o grupo e guardar os dados no vetor "ops", ou seja, à medida que é lida a hierarquia do group/tree, são inseridas todas as Operations, em conjunto com os seus valores/argumentos, no vetor, para que no final seja possível desenhar o resultado pretendido. No inicio da leitura de um group, é inserido um push e no final um pop.

Para a leitura dos child groups e dos sibling groups, o método xmlParserTransf é chamado recursivamente, permitindo assim a existência de vários grupos e de grupos dentro de outros grupos, tal como pretendido.

```
void xmlParserTransf(XMLElement* group) {

    float tX, tY, tZ, angleR, sX, sY, sZ, rX, rY, rZ, cR, cG, cB;

    if (strcmp(group->FirstChildElement()->Value(), "group") == 0) {
        group = group->FirstChildElement();
    }

    ops.push_back(new Push());

    for (XMLElement* t = group->FirstChildElement(); (strcmp(t->Value(), "models") != 0); t = t->NextSiblingElement()) {
        if (!strcmp(t->Value(), "translate")) {
            const char* a1 = t->Attribute("X");
            const char* a2 = t->Attribute("Y");
            const char* a3 = t->Attribute("Z");

            if (a1) {
                tX = stof(a1);
            }
            else tX = 0;

            if (a2) {
                tY = stof(a2);
            }
            else tY = 0;

            if (a3) {
                tZ = stof(a3);
            }
            else tZ = 0;
            ops.push_back(new Translacao(tX, tY, tZ));
        }
        if (!strcmp(t->Value(), "rotate")) {
            angleR = stof(t->Attribute("angle"));
            rX = stof(t->Attribute("axisX"));
            rY = stof(t->Attribute("axisY"));
            rZ = stof(t->Attribute("axisZ"));

            ops.push_back(new Rotation(angleR, rX, rY, rZ));

        }
        if (!strcmp(t->Value(), "scale")) {
            sX = stof(t->Attribute("X"));
            sY = stof(t->Attribute("Y"));
            sZ = stof(t->Attribute("Z"));

            ops.push_back(new Scale(sX, sY, sZ));
```

Figure 9: Método readParserTransf()

```
        }
        if (!strcmp(t->Value(), "scale")) {
            sX = stof(t->Attribute("X"));
            sY = stof(t->Attribute("Y"));
            sZ = stof(t->Attribute("Z"));

            ops.push_back(new Scale(sX, sY, sZ));

        }
        if (!strcmp(t->Value(), "color")) {
            cR = stof(t->Attribute("R"));
            cG = stof(t->Attribute("G"));
            cB = stof(t->Attribute("B"));
            ops.push_back(new Color(cR, cG, cB));
        }
    }

    for (XMLElement* modelo = group->FirstChildElement("models")->FirstChildElement("model"); modelo; modelo = modelo->NextSiblingElement("model")) {

        readFile(modelo->Attribute("file"));

        ops.push_back(new Draw(vertices));
        //Para limpar o vetor com os vértices
        vertices.clear();
    }

    if (group->FirstChildElement("group")) {
        xmlParserTransf(group->FirstChildElement("group"));
    }

    ops.push_back(new Pop());

    if (group->NextSiblingElement("group")) {
        xmlParserTransf(group->NextSiblingElement("group"));
    }
```

Figure 10: Método readParserTransf()

2.4 Solar System

Nesta fase, o objetivo final era o de criar uma representação do Sistema Solar. Nesta representação, todos os corpos celestes são desenhados utilizando o modelo da esfera desenvolvido na primeira fase. Este corpos incluem:

- Sol
- Planetas (Excluíndo Plutão)
- Satélites Naturais da Terra, Júpiter e Saturno

Em Saturno, a representação do Anel é feita com o torus que já foi demonstrado na 1ª secção.

Em termos de proporções relativamente ao tamanho dos corpos e distância entre eles, foram escolhidos valores que mantivessem os aspetos principais do Sistema Solar real, mas que permitissem uma visualização mais detalhada do mesmo.

As cores foram selecionadas com base numa aproximação das cores reais.

Se todos os valores utilizados correspondessem à mesma escala, seria bastante dificil, na minha opinião, de obter uma imagem apelativa e que permitisse ver o que realmente foi desenvolvido neste projeto. Qualquer das maneiras, deixo uma tabela com os valores reais sobre os tamanhos de todos os Corpos (excluíndo os Satélites) em relação à Terra, assim como a distância dos mesmos ao Sol:

Nome	Distância ao Sol(km)	Tamanho em Relação à Terra
Sol	0	109.19
Mercúrio	57 910 000	0.3829
Vénus	108 200 000	0.9499
Terra	149 600 000	1
Marte	227 900 000	0.5319
Júpiter	778 500 000	10.9733
Saturno	1 429 000 000	6.0
Urano	2 871 000 000	3.9809
Neptuno	4 495 000 000	3.8647

2.4.1 Ficheiros XML

Para representação do Sistema Solar, foram criadas duas cenas hierárquicas com transformações geométricas, ou seja, dois ficheiros XML diferentes, nomeadamente o "solarsystem.xml" e o "solarsystemNoR.xml.

O primeiro contêm uma representação do Sistema Solar com rotações, o que cria uma imagem mais representativa de um Sistema original, uma vez que não estão todos alinhados no eixo de coordenadas Z. Isto acontece porque primeiro são efetuadas as Rotações e só depois são efetuadas as Translações. No segundo ficheiro foram simplesmente retiradas todas as rotações. O objetivo era o de apresentar duas representações do mesmo Sistema Solar, que permitam apreciar visões diferentes do trabalho desenvolvido.

Como será possível ver nas imagens que se seguem, é possível definir para cada Corpo Celeste uma cor, uma rotação, uma translação, uma escala e que tipo de modelo queremos que seja representado.

```
<!--VENUS-->
<group>
    <color R="0.5469" G="0.4922" B="0.5117" />
    <rotate angle="-10.0" axisX="0.0" axisY="1.0" axisZ="0.0" />
    <translate X="0.0" Y="0.0" Z="-10.5" />
    <scale X="0.28" Y="0.28" Z="0.28" />
    <models>
        <model file="sphere.3d" />
    </models>
</group>

<!--TERRA-->
<group>
    <color R="0.0" G="0.0" B="1.0" />
    <rotate angle="-40.0" axisX="0.0" axisY="1.0" axisZ="0.0" />
    <translate X="0.0" Y="0.0" Z="-15.5" />
    <scale X="0.3" Y="0.3" Z="0.3" />
    <models>
        <model file="sphere.3d" />
    </models>
```

Figure 11: Excerto do ficheiro solarsystem.xml

```
<!--VENUS-->
<group>
    <color R="0.5469" G="0.4922" B="0.5117" />
    <translate X="0.0" Y="0.0" Z="-10.5" />
    <scale X="0.28" Y="0.28" Z="0.28" />
    <models>
        <model file="sphere.3d" />
    </models>
</group>

<!--TERRA-->
<group>
    <color R="0.0" G="0.0" B="1.0" />
    <translate X="0.0" Y="0.0" Z="-15.5" />
    <scale X="0.3" Y="0.3" Z="0.3" />
    <models>
        <model file="sphere.3d" />
    </models>
```

Figure 12: Excerto do ficheiro solarsystemNoR.xml

2.4.2 Resultado Final

Para apresentação dos resultados finais, foram utilizadas as opções "GL_FILL". De notar que por defeito, se não for passado como argumento o nome do ficheiro XML a ser lido, o Sistema Solar apresentado é o que contêm rotações:

2.4.3 Sistema Solar com Rotações

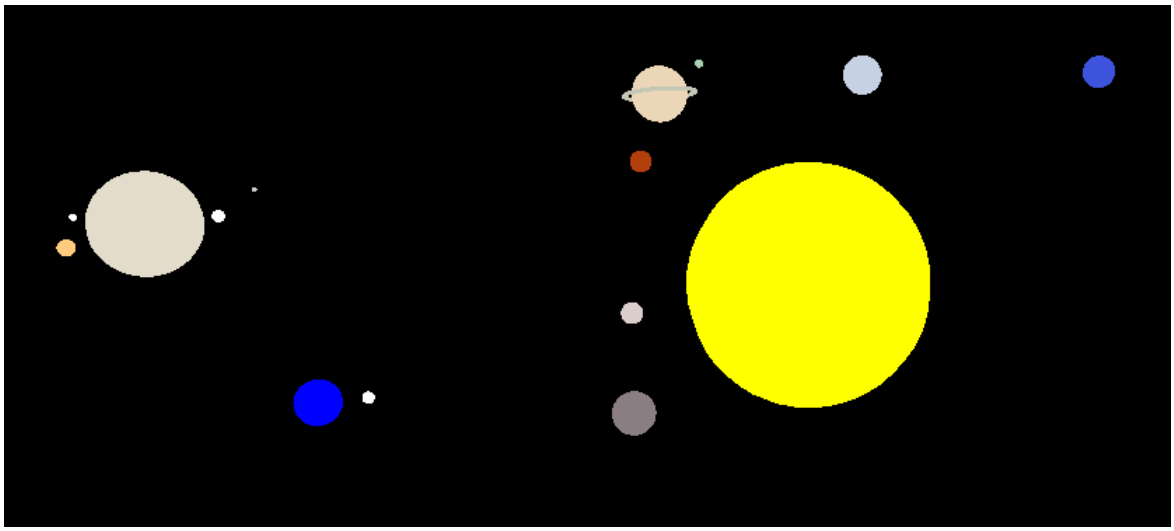

Figure 13: Sistema Solar com Rotações.

2.4.4 Sistema Solar sem Rotações

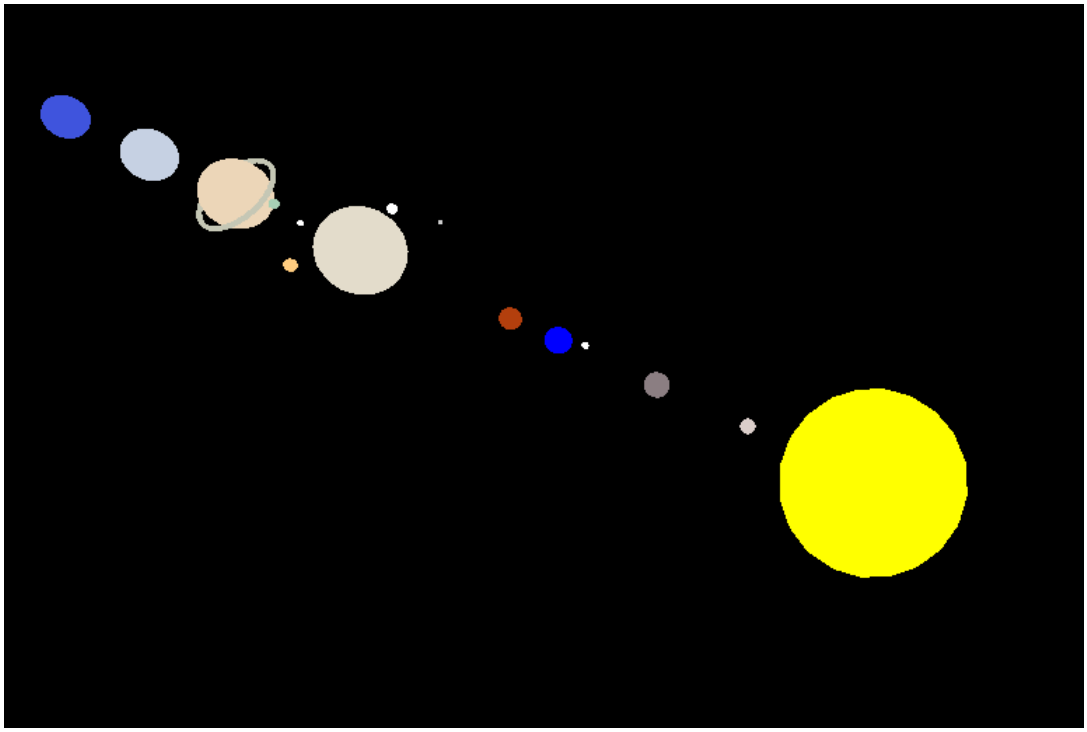

Figure 14: Sistema Solar sem Rotações.

3 Conclusão

Em suma, posso dizer que estou satisfeito com o trabalho realizado. Esta segunda parte deu-me conhecimentos sobre a criação de cenas hierárquicas utilizando transformações geométricas, ficando a conhecer todas as regras que deverão ser seguidas. O desenvolvimento desta fase foi sem dúvida algo mais trabalhoso que o desenvolvimento das primitivas mas que em termos de resultado final me deu mais satisfação. Outro beneficio desta foi o aumento de conhecimento com a linguagem utilizada, uma vez que ao longo do trabalho foram encontrados e resolvidos vários problemas que diziam respeito à estruturação do código.

Dito isto, diria que o balanço final do conhecimento adquirido é bastante positivo. Um aspeto a melhorar nesta 2ª fase seria a criação de um Sistema Solar que não seja estático.