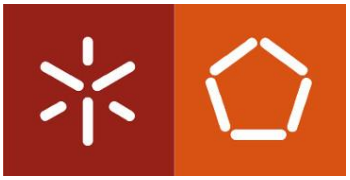

UNIVERSIDADE DO MINHO

DEPARTAMENTO DE INFORMÁTICA

TRABALHO PRÁTICO – Fase 2

Geometric Transforms

Computação Gráfica
Abril de 2021

Autores:

Ana Filipa Pereira A89589

Carolina Santejo A89500

Raquel Costa A89464

Sara Marques A89477

Índice

Índice de Figuras

Introdução

Esta segunda fase do projeto da cadeira de Computação Gráfica, consistiu em efetuar alterações no trabalho feito na fase 1 de forma a adicionar transformações geométricas, tais como a translação, a escala e a rotação.

Para realizar esta etapa, foi necessário considerar que os ficheiros XML possuem uma estrutura hierárquica em árvore, na qual cada nodo possui o ficheiro 3d onde estão os vértices, e transformações que são obrigatoriamente aplicadas aos nodos filhos.

As mudanças a realizar foram essencialmente a nível do motor, mas também foram precisos retoques no gerador.

A demonstração das transformações realizadas é feita através de um modelo do sistema solar com o sol, os planetas e as respetivas luas, bem como a cintura de asteroides que foi feita com recurso a um script em *python*.

Principais Objetivos

Para que esta etapa fosse realizada com sucesso, tornou-se necessário traçar vários objetivos importantes.

Inicialmente, foi preciso adicionar ao gerador uma nova primitiva, o Torus, de forma a desenhar os anéis de Saturno.

A nível do motor, foi necessário efetuar mudanças nas funções de “parser” e leitura do ficheiro XML, de forma a ser possível ler as transformações a efetuar. Visto que o ficheiro XML só é lido uma única vez, foram criadas estruturas de dados para guardar essa informação.

Após todos os objetivos anteriores serem cumpridos, o grupo teve de criar um modelo de um sistema solar.

Arquitetura do Programa - Atualização

Em geral, a arquitetura do programa tem como base a mesma que foi criada na fase passada, exceto alguns packages que foram necessários adicionar para auxiliar a realização da fase atual do projeto.

Entre eles, temos o package *Structs*, que surgiu devido à necessidade de aprimorar a estrutura de dados utilizada para armazenar os dados resultantes da leitura do ficheiro XML. Sendo esses mesmos motivos, a razão pela qual o *motor* faz "import" do mesmo.

Além disso, ainda foi adicionado o package *AsteroidScript* que contém um programa realizado em *python* que serviu como auxiliar para a conceção de uma parte do ficheiro XML utilizado como "demo scene", cujo propósito é representar um modelo estático do sistema solar.

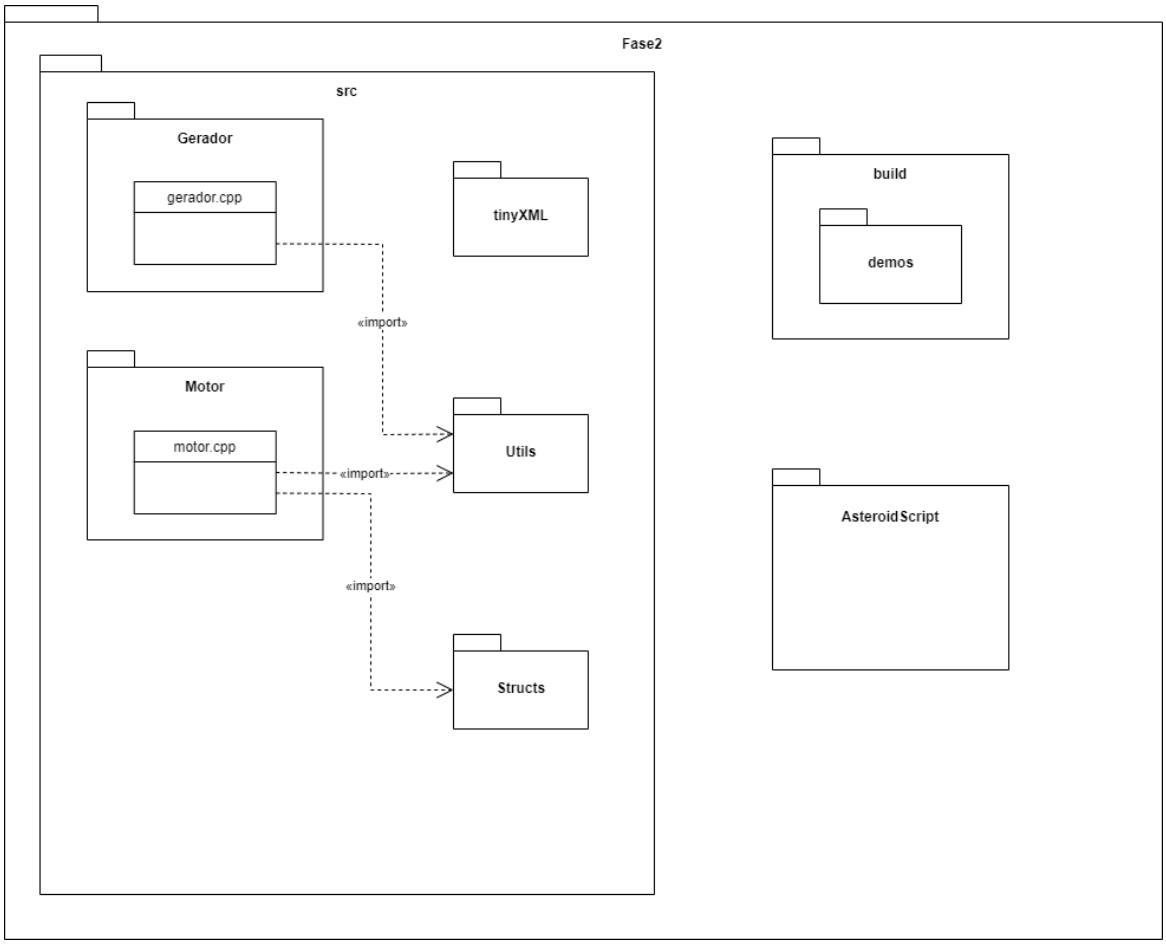

Figura 1 - Diagrama de Packages

Package *Structs*

De modo a facilitar o desenvolvimento da aplicação, foram criados novos *namespaces* (em ficheiros hpp, juntamente com os respetivos ficheiros cpp), que contêm as classes e funções necessárias à elaboração de transformações e à organização de dados.

transUtils

O *namespace transUtils* inclui uma classe *transform* constituída por quatro valores *float* e uma variável *transtype,* que corresponde a uma classe *enum* também definida no mesmo *namespace*. Um *transtype* é apenas um identificador do tipo de transformação, podendo tratar-se de uma rotação, translação, escala ou mudança de cor. Três dos valores *float* da classe *transform* correspondem a coordenadas x, y e z para estas mesmas transformações (à exceção da mudança de cor, caso em que correspondem aos valores RGB vermelho, verde e azul) enquanto que o último *float* é utilizado apenas para definir ângulos no caso das rotações.

Para além da classe *transform* são também definidas todas as funções *get* necessárias para aceder a qualquer uma das suas variáveis mantendo o encapsulamento, e funções set para definir variáveis *transform* com valores específicos.

groupUtils

O *namespace groupUtils* inclui a classe *group*, composta por vetores de *transforms*, *figures* e *groups*. Esta classe será a principal estrutura de dados responsável por armazenar a informação obtida a partir de ficheiros xml, sendo que cada grupo aí definido poderá conter não só transformações específicas mas também grupos "filhos" aos quais estas também terão de ser aplicadas. O uso desta classe irá então facilitar o cumprimento deste requisito.

O *groupUtils* inclui também as funções *get* necessárias para aceder a qualquer um dos vetores de um *group*, e funções responsáveis por adicionar novas transformações, figuras, ou grupos aos mesmos.

Desenho do “Torus” - Gerador

Como já foi referido, nesta fase foi necessário criar um modelo de um sistema solar, com o sol, planetas e respetivas luas. No entanto, no gerador, não havia nenhuma primitiva que servisse para desenhar os anéis de Saturno. Desta forma, o grupo adicionou no gerador a primitiva *Torus,* que recebe como argumentos, o raio da circunferência interior, a distância entre o centro do *torus* e o centro da circunferência interior (raio externo), o número de *stacks* e de *slices.*

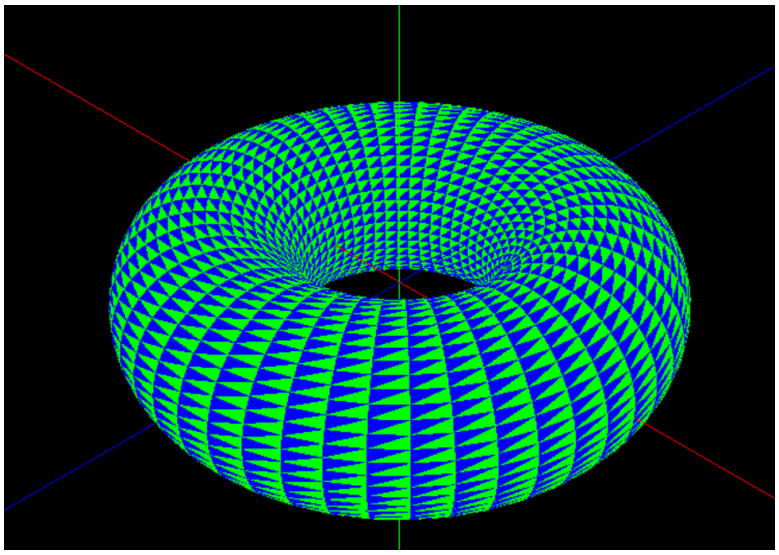

Figura 2 - Torus

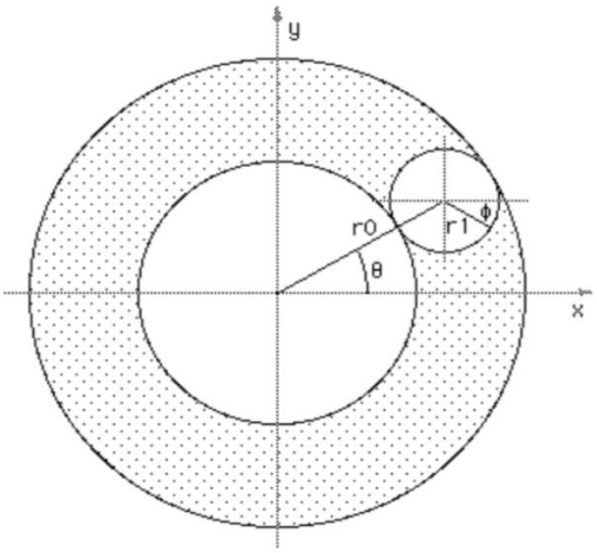

Figura 3 - Formato do Torus

Para o desenho do *Torus* foi necessário recorrer á sua parametrização, sendo que r é o raio interno e r axial o raio externo:

$$x = (r_{axial} + r\cos\phi)\cos\theta$$
$$y = (r_{axial} + r\cos\phi)\sin\theta$$
$$z = r\sin\phi$$

Figura 4 - Parametrização do Torus

```
figure generate::createTorus(float raio_in, float raio_out, int slices, int stacks) {
    figure f;

    float delta1 = 2 * M_PI / slices;
    float delta2 = 2 * M_PI / stacks;
    float phi = 0;
    float theta = 0;

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

            f.addPoint((raio_in + raio_out * cos(phi)) * cos(theta), (raio_in + raio_out * cos(phi)) * sin(theta), raio_out * sin(phi));

            f.addPoint((raio_in + raio_out * cos(phi)) * cos(theta + delta1), (raio_in + raio_out * cos(phi)) * sin(theta + delta1), raio_out * sin(phi));

            f.addPoint((raio_in + raio_out * cos(phi + delta2)) * cos(theta + delta1), (raio_in + raio_out * cos(phi + delta2)) * sin(theta + delta1),
                raio_out * sin(phi + delta2));

            f.addPoint((raio_in + raio_out * cos(phi + delta2)) * cos(theta + delta1), (raio_in + raio_out * cos(phi + delta2)) * sin(theta + delta1),
                raio_out * sin(phi + delta2));

            f.addPoint((raio_in + raio_out * cos(phi + delta2)) * cos(theta), (raio_in + raio_out * cos(phi + delta2)) * sin(theta),
                raio_out * sin(phi + delta2));

            f.addPoint((raio_in + raio_out * cos(phi)) * cos(theta), (raio_in + raio_out * cos(phi)) * sin(theta), raio_out * sin(phi));


            phi = delta2 * (j + 1);
        }

        theta = delta1 * (i + 1);
    }


    return f;
}
```

Figura 5 - Excerto de código C++

Como é possível ver no código acima, foram feitos dois ciclos *fo*r aninhados, sendo que o exterior percorre as *slices* e o interior as *stacks*. A cada iteração do ciclo interior são desenhados dois triângulos que correspondem ao plano delimitado pela *stack* e pela *slice* correspondentes.

Ambos os ângulos θ *e* Φ pertencem ao intervalo $[0\ ,\ 2\pi]$, sendo que a cada iteração pelas *stacks* $\Phi = \Delta 2 * (j + 1)$ e a cada iteração pelas slices $\theta = \Delta 1 * (i + 1)$.

Motor – Atualização

Nesta fase do projeto, foi necessário atualizar a aplicação "Motor" de forma a conseguir ler o novo formato de ficheiro XML proposto no enunciado, validando apenas os ficheiros que apresentarem esse mesmo formato. Além disso, foi fundamental também alterar a estrutura de dados utilizada para armazenar os conteúdos do ficheiro lido.

Estrutura de dados

Para armazenar os dados obtidos através do ficheiro XML dado pelo utilizador, usamos a classe criada *group* no ficheiro *Structs/groupUtils.hpp*. Esta classe é usada como estrutura de dados, e é constituída por 3 vetores, um vetor dedicado às figuras obtidas do ficheiro, outro às transformações e finalmente um último que contém os *group* filhos presentes nesse mesmo *group.*

```
class group
        std::vector<figure> figuras;
        std::vector<transform> transformacoes;
        std::vector<group> filhos;
```

Figura 6 - Class group

Foi essencial criar uma classe recursiva devido ao formato do ficheiro XML, uma vez que, um elemento <*group*> pode ter subelementos <*group*> como filhos (Figura x).

```
<scene>
    <group>
        <models>
            <model file="sphere.3d" />
        </models>
        <group>
            <models>
                <model file="box.3d" />
            </models>
        </group>
    </group>
</scene>
```

Figura 7 - Exemplo de ficheiro XML

Desta forma, no ficheiro *motor.cpp*, consideramos a variável global "*group* grupo", onde vamos adicionando, conforme o que é lido do ficheiro XML, as transformações, as figuras e os <group> filhos, aos vetores correspondentes de cada tipo.

Portanto, o uso de uma estrutura para armazenar os dados de um ficheiro facilita tanto o desenho das figuras como o das suas transformações, sendo que, basta aceder à estrutura global e percorrê-la.

Leitura e "Parse" do ficheiro XML

Para ler o ficheiro e fazer "parse" do mesmo, foi necessário modificar a função "lerFicheiroXML" feita na 1ª fase deste trabalho. Primeiramente, dividimos a função em duas, separando a parte responsável por ler os ficheiros .3d, daquela que efetivamente lê o ficheiro XML, criando assim, a função "lerFicheiro3D".

De seguida criamos uma nova função responsável por dividir e identificar os vários elementos presentes num *<group>* do ficheiro XML ("parseGroupXML"). Esta função recebe um apontador para um elemento *<group>* do ficheiro e a estrutura de dados *group* onde irá armazenar os diversos dados lidos.

Primeiramente, irá criar um ciclo, de forma a percorrer cada elemento filho do apontador passado no input. Dentro do ciclo, é analisado o *value* de cada um deles, que poderá corresponder a 3 tipos: *Transform*, *Figure* e *Group*. Quando o *value* é do tipo "translate", "rotate", "scale" ou "color" consideramos que é um *Transform*, e nesse caso iremos analisar os atributos do elemento para fazermos "set" da classe *Transform* inicializada e transcrevermos o valor de cada variável necessária para realizar a transformação. Caso o valor do elemento em questão seja "models", teremos de analisar os filhos desse mesmo elemento, de forma a obter os modelos, que contêm a referência para o ficheiro .3d que deve ser lido. De seguida, teremos de chamar a função "lerFicheiro3D", de forma a obter a *Figure* equivalente ao ficheiro, e por fim, adicioná-la à estrutura *group* que estamos a usar. Quando o elemento filho de um <group> for do próprio tipo *Group*, iremos inicializar uma nova classe *group,* com o nome de *childGr*. Depois, iremos chamar a própria função "parseGroupXML", passando como input o apontador para o elemento filho onde se encontra, e o *childGr,* voltando assim, a fazer todo este processo sucessivamente até chegar ao final do ficheiro.

Sendo assim, usamos a recursividade para conseguir atravessar o ficheiro XML e armazená-lo numa estrutura, que também ela própria é recursiva.

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

    if (translate==1){
        validate=1;
        printf("ERRO TRANSLATE - Couldn't parse XML file\n");
        return g;
    }
    else {
        x = atof(elem->Attribute("X"));
        y = atof(elem->Attribute("Y"));
        z = atof(elem->Attribute("Z"));

        t.setTransform(x, y, z, 0, transtype::translate);
        g.addTransform(t);
        translate=1;
    }
}
```

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

    if (scale==1){
        validate=1;
        printf("ERRO SCALE - Couldn't parse XML file\n");
        return g;
    }
    else {
        x = atof(elem->Attribute("X"));
        y = atof(elem->Attribute("Y"));
        z = atof(elem->Attribute("Z"));

        t.setTransform(x, y, z, 0, transtype::scale);
        g.addTransform(t);
        scale = 1;
    }
}
```

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

    if (color==1){
        validate=1;
        printf("ERRO COLOR - Couldn't parse XML file\n");
        return g;
    }
    else {
        x = atof(elem->Attribute("R"));
        y = atof(elem->Attribute("G"));
        z = atof(elem->Attribute("B"));

        t.setTransform(x, y, z, 0, transtype::color);
        g.addTransform(t);
        color=1;
    }
}
```

```
else if(strcmp(elem->Value(),"rotate")==0){

    if (rotate==1){
        validate=1;
        printf("ERRO ROTATE - Couldn't parse XML file\n");
        return g;
    }
    else {
        angle = atof(elem->Attribute("angle"));
        x = atof(elem->Attribute("axisX"));
        y = atof(elem->Attribute("axisY"));
        z = atof(elem->Attribute("axisZ"));

        t.setTransform(x, y, z, angle, transtype::rotate);
        g.addTransform(t);
        rotate=1;
    }
}
```

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

    if (models==1){
        validate=1;
        printf("ERRO MODELS - Couldn't parse XML file\n");
        return g;
    }
    else {

        TiXmlElement *model = elem->FirstChildElement("model");

        while (model) {
            const char *ficheiro = model->Attribute("file");

            //Abre o ficheiro .3d
            g = lerFicheiro3D((getPath() + ficheiro), g);

            //next sibling
            model = model->NextSiblingElement("model");
        }
        models=1;
    }
}
```

```
else if (strcmp(elem->Value(), "group") == 0) {
    group childGr;
    childGr = parseGroupXML(elem, childGr);
    g.addGroup(childGr);
}
```


Figura 8 - Parse dos vários elementos do XML

Sistema Solar

Para demostrar as funcionalidades do programa implementado foi solicitado o desenho de um sistema solar. Para a sua elaboração foi necessário escrever um ficheiro XML com todas as transformações e ficheiros .3d necessários.

De modo a proporcionar uma experiência visual mais agradável ao observador, não foram utilizadas as proporções reais dos planetas uma vez que os seus tamanhos iriam diferir demasiado, ou seja, os mais pequenos seriam quase impercetíveis.

Sendo assim, para o desenho do sistema solar foi utilizada a Figura 9 como base para as proporções e cores. Adicionalmente foram também incluídas algumas luas dos planetas Terra, Júpiter e Saturno, sendo que nos últimos dois foram escolhidas apenas 4 (Europa, Ganimedes, Io, Calíope e Titã, Encéfalo, Mimas, Tétis respetivamente). Por fim e para tornar o desenho o mais realista possível, foi também desenhada a cintura de asteroides entre marte e Júpiter com recurso a um script em *python*.

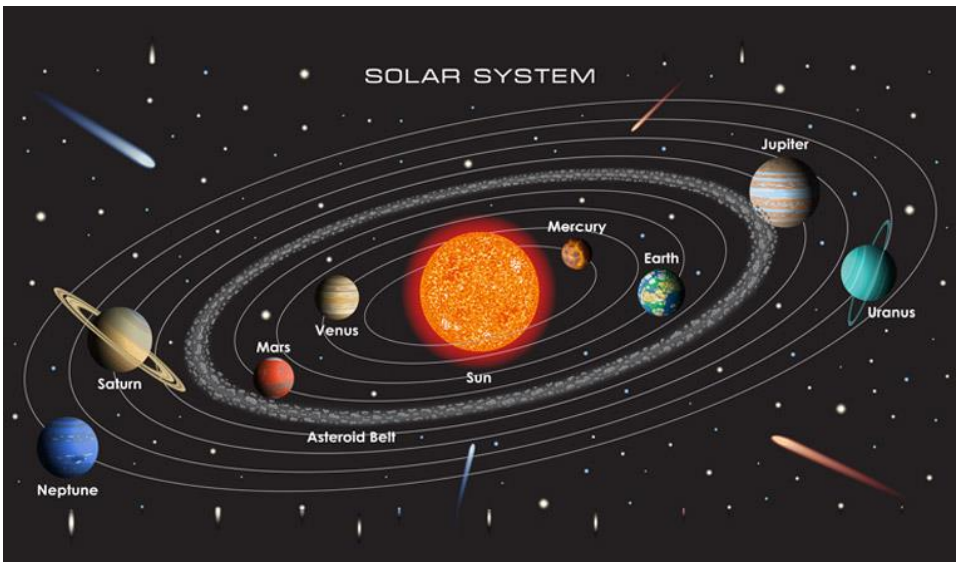

Figura 9 - Sistema Solar

Modelos utilizados

Com recurso ao gerador implementado na fase anterior, foram criados três ficheiros .3d para servirem de modelos no ficheiro XML. Em primeiro lugar foi gerado o ficheiro sphere.3d (esfera de raio 2, com 20 *stacks* e 20 *slices*) que será responsável por desenhar o sol e todos os planetas e respetivas luas. Para os anéis de saturno foi gerado o ficheiro torus.3d (toro de raio

interior 0.5, raio exterior 3, com 2 *stacks* e 30 *slices*). Por último, para os asteroides que são esferas mais pequenas foi gerado o ficheiro asteriod.3d (esfera de raio 0.1 com 10 *stacks* e 10 *slices*).

Criação do ficheiro XML

A criação do ficheiro XML final foi realizada de acordo com uma estrutura bem definida onde cada grupo (*group*) pode conter várias transformações (translação, rotação, escala ou cor) que afetam todos os grupos filhos e cada um deles tem de incluir dentro da *tag models* a referência a pelo menos um ficheiro *.3d*. A escrita do ficheiro foi feita de forma gradual, isto é, a cada novo grupo acrescentado era verificado o output da sua leitura.

Sendo assim, o ficheiro XML final contém vários grupos principais que referenciam um dos seguintes desenhos: o sol, um planeta (que poderá incluir subgrupos que referenciam as suas luas ou anéis) ou um asteroide.

De acordo com o output desejado foram escritos todos os conteúdos seguindo a ordem a seguir referida.

- Desenho do sol no centro do referencial (sem translações):

```
<group>
    <color R="1" G="0.6" B="0"/>
    <models>
        <model file="sphere.3d"/>
    </models>
</group>
```

Figura 10 - Excerto do ficheiro XML (1)

- Desenho dos planetas telúricos (Mercúrio, Vénus, Terra e Marte) num ponto qualquer da sua orbita (rotação em torno do eixo y) com intervalo de 0.5 unidades no eixo do x entre si:

```
<group>
    <rotate angle="-30" axisX="0" axisY="1" axisZ="0"/>
    <color R="0.65" G="0.3" B="0.15"/>
    <translate X="3" Y="0" Z="0"/>
    <scale X="0.05" Y="0.05" Z="0.05"/>
    <models>
        <model file="sphere.3d"/>
    </models>
</group>
```

Mercúrio

```
<group>
    <rotate angle="12" axisX="0" axisY="1" axisZ="0"/>
    <color R="0.7" G="0.5" B="0.3"/>
    <translate X="3.5" Y="0" Z="0"/>
    <scale X="0.06" Y="0.06" Z="0.06"/>
    <models>
        <model file="sphere.3d"/>
    </models>
</group>
```

Vénus

```
<group>
    <rotate angle="234" axisX="0" axisY="1" axisZ="0"/>
    <color R="0" G="0" B="0.8"/>
    <translate X="4" Y="0" Z="0"/>
    <scale X="0.08" Y="0.08" Z="0.08"/>
    <models>
        <model file="sphere.3d"/>
    </models>
    <group>
        <color R="0.9" G="0.9" B="0.9"/>
        <translate X="0" Y="0" Z="3"/>
        <scale X="0.25" Y="0.25" Z="0.25"/>
        <models>
            <model file="sphere.3d"/>
        </models>
    </group>
</group>
```

Terra

Lua

```
<group>
    <rotate angle="332" axisX="0" axisY="1" axisZ="0"/>
    <color R="0.85" G="0.2" B="0.2"/>
    <translate X="4.5" Y="0" Z="0"/>
    <scale X="0.06" Y="0.06" Z="0.06"/>
    <models>
        <model file="sphere.3d"/>
    </models>
</group>
```

Marte

Figura 11 - Excerto do ficheiro XML (2)

- Desenho dos planetas gasosos (Júpiter, Saturno, Úrano e Neptuno) num ponto qualquer da sua orbita (rotação em torno do eixo y) com intervalo de 1.5 unidades no eixo do x entre si:

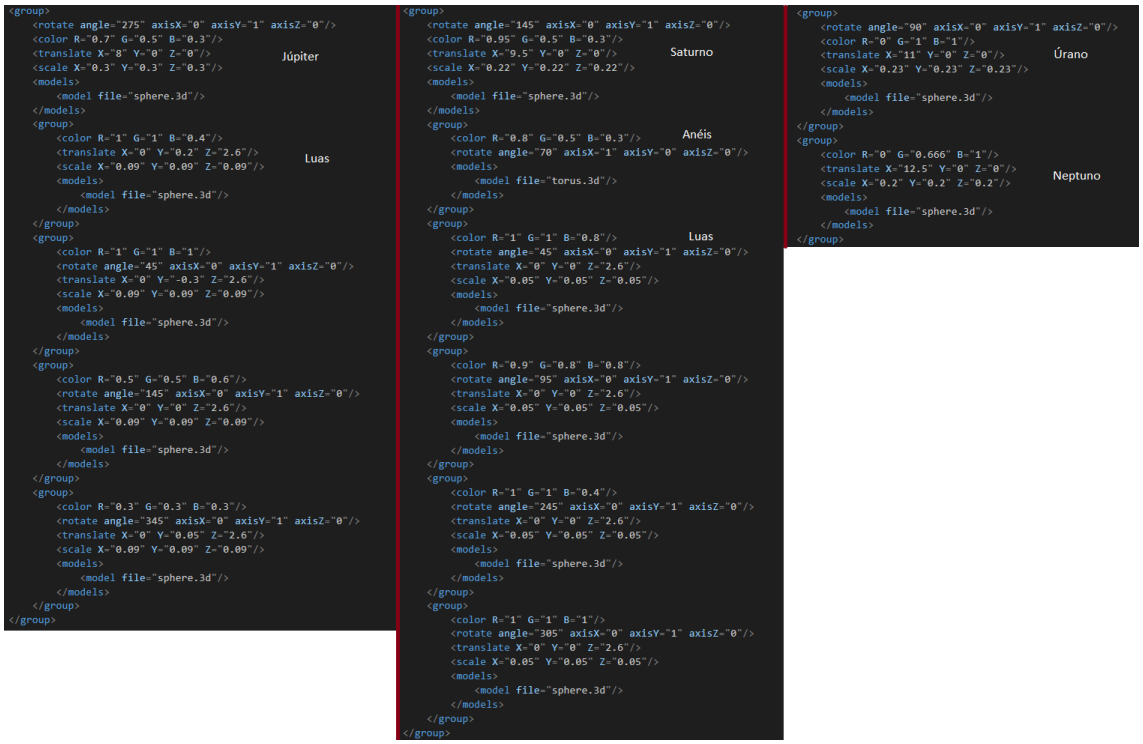

Figura 12 - Excerto do ficheiro XML (3)

Script extra

Para finalizar a escrita do ficheiro XML, de modo a facilitar a conceção da cintura de asteroides, foi criado um pequeno script em python que se localiza na pasta AsteroidScript.

Utilizando a função random.uniform da biblioteca random foram calculados vários valores aleatórios entre intervalos de modo a garantir a aleatoriedade das posições dos asteroides.

Deste modo foi calculado um angulo entre 0 e 360 para efetuar a rotação do asteroide em torno do eixo dos y e o raio foi definido entre 5.5 e 6.6 uma vez que são as posições que estão entre os dois planetas pretendidos (Marte e Júpiter). Para a altura foi determinada uma pequena variação entre -0.1 e 0.1 para que os asteroides não pertencessem ao mesmo plano e ainda um intervalo para fazer a escala da figura entre 0.15 e 0.25 de modo a terem tamanhos variáveis. Por fim, foi também estabelecido um pequeno intervalo nas cores 0.48 a 0.53 para alternar os tons de cinzento.

Output

Após a leitura do ficheiro XML final podem-se observar os seguintes *outputs*:

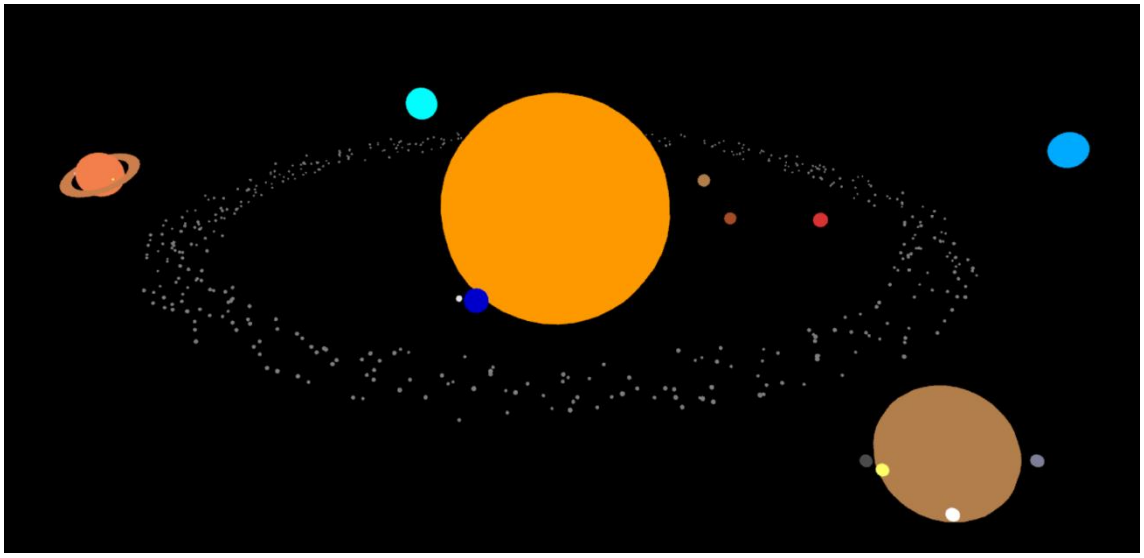

Figura 13 - Output do desenho do sistema solar

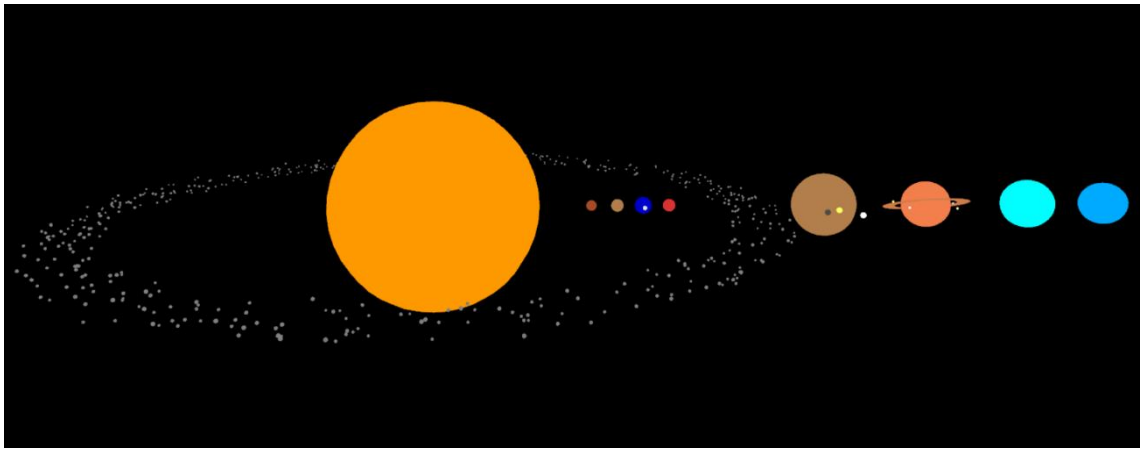

Figura 14 - Output do desenho do sistema solar com os planetas alinhados (sem rotações aplicadas aos planetas)

Apreciação Final

Melhorias

Nesta segunda fase do projeto foi revisto o código já implementado e foi necessário redefinir a função *createSphere* do gerador, que calcula todos os vértices necessários para o desenho de uma esfera.

Na versão anterior eram percorridos os valores dos ângulos correspondentes às *slices* e *stacks* respetivas o que originou um problema na condição do ciclo *for,* uma vez que os valores dos *floats* possuem incertezas após efetuar vários cálculos e havia cenários onde não eram desenhados todos os triângulos. Sendo assim, nesta fase foi corrigido o problema utilizando, em alternativa, o número de *slices* e *stacks* para percorrer os ciclos, onde sabendo o $\Delta 1$ e $\Delta 2$ é possível calcular os ângulos $\theta 1$ *e* $\theta 2$ correspondentes à iteração.

Todos os cálculos das coordenadas dos pontos foram mantidos.

$$\Delta 1 = \frac{\pi}{stacks}$$

$$\Delta 2 = \frac{2\pi}{stacks}$$

$$\theta 2 = j * \Delta 2$$

$$\theta 1 = \theta 1 + \Delta 1$$

Conclusão

Ao longo desta segunda fase, foi possível aplicar e consolidar vários conceitos abordados nas aulas teóricas e práticas, bem como aprofundar o nosso conhecimento em *OpenGL*.

O grupo considera que realizou com sucesso esta etapa, na medida em que o sistema solar gerado correspondeu totalmente às expectativas.

Assim, nas próximas fases do projeto, o objetivo consistirá em tornar o modelo do sistema solar cada vez mais realista e detalhado.