Universidade do Minho

Mestrado Integrado em Engenharia Informática

Computação Gráfica

2ª Fase - Transformações Geométricas

Grupo 36

Gonçalo Esteves
.(A85731).

João Araújo
.(A84306).

Mário Real
.(A72620).

Rui Oliveira
.(A83610).

29 de Março de 2020

Conteúdo

Resumo

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

1 Introdução

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

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

2 Gerador

Nesta secção iremos explicar as alterações implementadas no nosso gerador, bem como as novas funcionalidades implementadas.

Para começar, adotamos uma nova estratégia, inserindo as funções geradoras dos pontos de cada figura num ficheiro à parte. Para além disto, criamos uma nova estrutura, a qual designamos de *Point*, e que utilizamos para guardar as informações relativas a um ponto. Deste modo, e recorrendo também a vetores de *Point*'s, podemos guardar todos pontos que constituem uma figura, podendo posteriormente guardá-los num novo ficheiro. Claro está que o gerador continuará a ser o responsável pela geração e armazenamento dos pontos dos triângulos que permitem construir as diversas figuras geométricas. Como já foi referido, adicionou-se também uma nova figura, o *Torus*.

2.1 *Torus*

Por forma a criar esta primitiva, adicionaram-se novas funcionalidades ao gerador.

Podemos ver esta nova figura como sendo um cilindro, onde as duas bases se unem. Como tal, definimos um raio interno, por forma a definir a espessura do tórus, sendo que a lateral do cilindro pode ser dividida em camadas e fatias.

Desta forma, começa-se por geral um anel, sendo este replicado até se completar o conjunto de fatias.

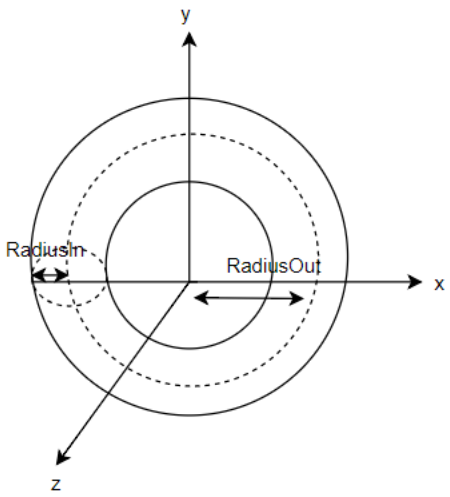

Figura 1: Idealização do Torus

Tal como foi observável anteriormente, o raio exterior define a abertura do *torus*, ou seja, a distância entre os anéis do *torus* e a origem do referencial. Consideramos também os ângulos *alpha* e *beta*, que variam entre 0 e 2π, por forma a abranger o *torus* na sua totalidade. Neste caso, o *alpha* é utilizado para a construção das camadas, enquanto que o *beta* é utilizado para a criação dos anéis interiores. De modo a gerar os pontos necessários à criação desta figura, utilizamos dois ciclos aninhados, sendo o primeiro referente às camadas e o segundo à criação das fatias.

```
vector<Point> generateTorus(float radiusIn, float radiusOut , int slices, int layers){
    float alpha, nextAlpha, beta, nextBeta;
    Point p1, p2, p3, p4;
    vector<Point> points;

    for (int i = 0; i < layers; i++) {
        beta = (float)i * (float)(2 * M_PI / layers);
        nextBeta = (float)(i + 1) * (float)(2 * M_PI / layers);

        for (int j = 0; j < slices; j++) {
            alpha = (float)j * (float)(2 * M_PI / slices);
            nextAlpha = (float)(j + 1) * (float)(2 * M_PI / slices);

            p1 = makePointTorus(radiusIn, radiusOut, nextBeta, alpha);
            p2 = makePointTorus(radiusIn, radiusOut, beta, alpha);
            p3 = makePointTorus(radiusIn, radiusOut, nextBeta, nextAlpha);
            p4 = makePointTorus(radiusIn, radiusOut, beta, nextAlpha);

            //primeiro triangulo
            points.push_back(p1);
            points.push_back(p2);
            points.push_back(p3);

            //segundo triangulo
            points.push_back(p3);
            points.push_back(p2);
            points.push_back(p4);
        }
    }

    return points;
}
```

Listing 1: Código para gerar o Torus

Recorremos também a uma função por nós criada, designada de *makePointTorus*, que gera os pontos de forma adequada a esta figura:

```
Point makePointTorus(float radiusIn,float radiusOut,float beta,float
    alpha){
    Point p;

    p.x = cos(alpha) * (radiusIn*cos(beta) + radiusOut);
    p.y = sin(alpha) * (radiusIn*cos(beta) + radiusOut);
    p.z = radiusIn   * sin(beta);

    return p;
}
```

Listing 2: Código para gerar um ponto do Torus

Desta forma, gerou-se o *torus*, que nos será útil noutras alturas ao longo deste trabalho.

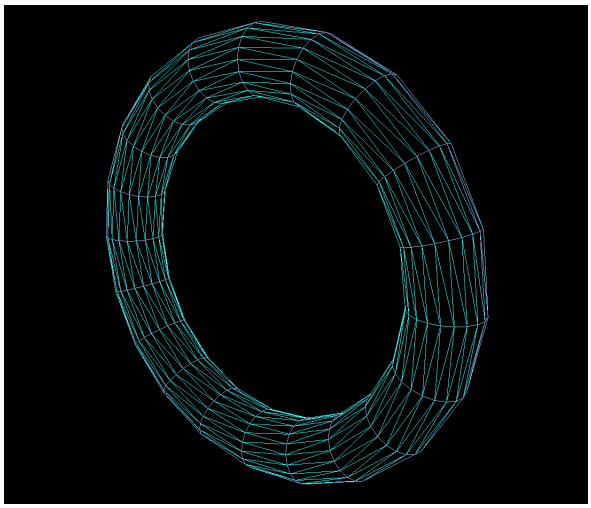

Figura 2: Torus

3 *Engine*

Com o intuito de implementar as novas funcionalidades necessárias nesta fase, optamos por alterar a estruturação do código já elaborado. Deste modo, criamos algumas classes, explicadas de seguida.

3.1 *Point*

Criamos uma classe designada de *Point*, que também usamos no gerador, por forma a guardar de forma mais adequada as coordenadas de um ponto.

```
class Point {
    private:
        float x, y, z;

    public:
        Point();
        Point(float xx, float yy, float zz);
        float getX();
        float getY();
        float getZ();
};
```

Listing 3: Classe que guarda as coordenadas de um ponto

3.2 *Shape*

Por forma a guardar todos os pontos correspondentes a uma figura geométrica, criamos a classe *Shape*, que suporta um vetor de *Point*'s.

```
class Shape {
    private:
        vector<Point*> points;

    public:
        Shape();
        Shape(vector<Point*> p);
        vector<Point*> getPoints();
};
```

Listing 4: Classe que guarda os pontos de uma figura geométrica

3.3 *Transformation*

De modo a armazenar todas as informações associadas a uma transformação geométrica, criamos a seguinte classe. As transformações podem ser rotações, translações e redimensionamentos. Usamos esta classe também para guardar as informações das cores a utilizar para cada figura.

Tendo isto em vista, consideramos necessário guardar o vetor aplicado na transformação e, no caso da rotação, o ângulo também.

```
1  class Transformation {
2
3      private:
4          string type;
5          float angle, x, y, z;
6
7      public:
8          Transformation();
9          Transformation(string t, float a, float xx, float yy, float
       zz);
10         string getType();
11         float getAngle();
12         float getX();
13         float getY();
14         float getZ();
15 };
```

Listing 5: Classe que guarda as informações de uma transformação geométrica

3.4 *Group*

Esta classe será responsável por armazenar toda a informação de um dado grupo, ou seja, todas as informações das transformações geométricas e das primitivas associadas, o que a torna numa das classes mais importantes do projeto. Esta classe irá armazenar:

- **Grupos** - usando um vetor com todos os grupos filhos;
- **Transformações** - usando um vetor com todas as transformações geométricas do grupo;
- **Figuras** - usando um vetor com pontos para desenhar cada uma das formas geométricas;

```
class Group {
    private:
        vector<Group*> groups;
        vector<Transformation*> trans;
        vector<Shape*> shapes;

    public:
        Group();
        void addGroup(Group* g);
        void addTransformation(Transformation* t);
        void setShapes(vector<Shape*> sh);
        vector<Group*> getGroups();
        vector<Transformation*> getTrans();
        vector<Shape*> getShapes();
};
```

Listing 6: Classe que guarda as informações de um dado grupo

3.5 *Camera*

À funcionalidade de movimentar a câmera com as setas, adicionou-se também a possibilidade de movê-la através do rato ou utilizando um menu com todos os planetas, de modo a focar no planeta escolhido. Com este objetivo em vista, idealizamos esta classe.

Inicialmente, a câmera encontra-se direcionada para a origem do referencial, sendo esta capaz de se movimentar numa superfície esférica imaginária. Assim, uma câmera necessita de guardar a sua posição atual, recorrendo a vareáveis *posX*, *posY* e *posZ*, que armazenam respetivamente as coordenadas x, y e z da câmera. Além disso, deve guardar também a variável *radius*, que armazena o raio da superfície esférica imaginária, e as variáveis *alpha* e *teta*, que correspondem ao ângulo que define a posição da câmera no plano xOz e xOy, respetivamente.

Com o intuito de implementar o menu que permite focar em determinado planeta, foi necessário garantir que o movimento da câmera acompanha a posição para a qual esta está a olhar. Como tal, definimos as variáveis *lookX*, *lookY* e *lookZ*, que indicam a posição para a qual a câmera está virada.

Deste modo, e atendendo às coordenadas esféricas, é possível definir a posição da câmera como:

```
    posX = lookX + radius*sin(alpha)*cos(teta);
    posY = lookX + radius*sin(teta);
    posZ = lookZ + radius*cos(alpha)*cos(teta);
```

3.5.1 Movimentação com as setas

Por forma a implementar esta funcionalidade, decidimos definir uma variável global designada por *speed*, que estabelece a velocidade com que se movimentará a câmera.

As setas para cima e para baixo permitem o movimento no sentido respetivo, sendo para tal necessário alterar o valor de *teta*, uma vez que este se refere à posição no plano xOz.

As setas para a esquerda e para a direita também permitem efetuar movimentos no respetivo sentido, sendo nete caso necessário alterar o valor de *alpha*, já que este se refere à posição no plano xOy.

Para além destas teclas, as teclas F1 e F2 permitem respetivamente uma aproximação ou afastamento da posição da câmera, sendo neste caso necessário alterar o valor do raio da superfície. É também possível recolocar a câmera na posição inicial utilizando a tecla F6.

3.5.2 Movimentação com o rato

Esta funcionalidade faz com que o utilizador seja capaz de movimentar a câmera, pressionando o botão esquerdo do rato e mexendo-o. Com este fim em vista, definimos uma variável *mousePressed* que indica se o botão esquerdo do rato está a ser pressionado.

A função *pressMouse* é responsável por armazenar as coordenadas x e y do rato, assim que o botão esquerdo deste é pressionado, e também por somar aos ângulos da câmera a variação efetuada com o movimento deste. Assim garantimos que a câmera fica na posição desejada quando o botão deixa de ser pressionado. Para isto, foi necessário definir também as variáveis $mousePosX$ e $mousePosY$ que indicam as coordenadas x e y do rato.

Por fim, sempre que o rato é movimentado, é necessário alterar os valores da posição da câmera. Para tal, antes de obtermos a sua posição final, somamos aos ângulos a variação do movimento do rato. Neste processo é necessário garantir que o ângulo teta não ultrapassa os 90 nem os -90 gruas.

```
class Camera {
    private:
        float radius, speed, alpha, teta, posX, posY, posZ, lookX,
    lookY, lookZ, mouseX, mouseY;
        bool mousePressed;

    public:
        Camera();
        float getPosX();
        float getPosY();
        float getPosZ();
        float getLookX();
        float getLookY();
        float getLookZ();
        void posIniCamera();
        void changeLook(float x, float y, float z);
        void specialKeysCamera(int key);
        void pressMouse(int button, int state, int x, int y);
        void motionMouse(int x, int y);
};
```

Listing 7: Classe que define a câmera

3.6 *Parser*

Nesta fase, foi necessário modificar a forma como é efetuado o *parsing* do XML, uma vez que agora também se torna necessário aplicar transformações geométricas às figuras geométricas obtidas através da leitura dos ficheiros .3d. As informações relativas a estas transformações geométricas encontram-se também registadas nos ficheiros XML, daí ser necessário alterar o *parser*.

```
<scene>
      <group>
            <!-- Sistema Solar -->
            <group>
                  <!-- Sol -->
                  <scale X="3.6" Y="3.6" Z="3.6" />
                  <colour R="1" G="0.55" B="0.0" />
                  <models>
                        <model file="sphere.3d" />
                  </models>
            </group>
            //...
            <group>
                  <!-- Terra -->
                  <translate X="-40" Z="10" />
                  <scale X="0.63" Y="0.63" Z="0.63" />
                  <colour R="0.39" G="0.58" B="0.93" />
                  <models>
                        <model file="sphere.3d" />
                  </models>
                  <group>
                        <!-- Lua -->
                        <scale X="0.3" Y="0.3" Z="0.3" />
                        <translate X="13" Y="10" Z="13" />
                        <colour R="0.2" G="0.2" B="0.2" />
                        <models>
                              <model file="sphere.3d" />
                        </models>
                  </group>
            </group>
            //...
</scene>
```

Listing 8: Parte do ficheiro XML solarsystem.xml

Tal como na primeira fase, recorremos à API do *tinyxml2*, por forma a realizar o *parsing* dos ficheiros, alterando, no entanto, o modo de leitura destes.

Inicialmente, carregamos todo o ficheiro para a memória, utilizando a função *loadXMLFile*, sendo esta também responsável por invocar, em caso de sucesso, a função *parseGroup*, que processa todos os nodos. Para tal, efetua-se um ciclo que irá percorrer todos os nodos, verificando individualmente qual o caso em que se encontra.

Desta feita, caso aconteça de este corresponder a uma transformação geométrica, invoca-se a função encarregue de recolher as informações necessárias e adicioná-las à estrutura. É de relevo referir que quando se recolhe as informações relativas a uma translação, é necessário armazenar também as informações relativas à orbita.

Tal como já foi referido, as informações relativas à cor com a qual as figuras deverão ser preenchidas vão ser também guardadas com o auxílio da classe *Transformation*, estando, por isso, estas também guardadas no XML, sendo necessário extraí-las juntamente com as informações das transformações geométricas. Assim, tal como acontece com as transformações geométricas, existe uma função responsável por extrair as informações necessárias e adicioná-las à estrutura.

Por outro lado, o nodo poderá conter a informação do ficheiro .3d que deverá ser lido. Nesse caso, o nodo estará marcado como sendo um *models*, devendo ser utilizada a função *parseModels*, que invoca a função *readPointsFile*, por forma a ler os pontos de cada ficheiro, armazenando-os num vetor de formas geométricas, que será armazenado na estrutura principal.

Por fim, também poderá ocorrer que existam grupos filhos e, para tal, invoca-se a função *parseGroup* recursivamente, por forma a que estes também sejam processados.

```
void parseGroup(Group *group, XMLElement *gElement, vector<Point*> *
    orbits, int d){
    XMLElement *element = gElement -> FirstChildElement();

    while (element){
        if (strcmp(element -> Name(),"translate") == 0)
            parseTranslate(group,element,orbits,d);

        else if (strcmp(element -> Name(),"scale") == 0)
            parseScale(group,element);

        else if (strcmp(element -> Name(),"rotate") == 0)
            parseRotate(group,element);

        else if (strcmp(element -> Name(),"models") == 0)
            parseModels(group, element);

        else if (strcmp(element -> Name(),"colour") == 0)
            parseColour(group, element);

        else if (strcmp(element -> Name(),"group") == 0){
            Group *child = new Group();
            group -> addGroup(child);
            parseGroup(child,element,orbits,d+1);
        }

        element = element -> NextSiblingElement();
    }
}
```

Listing 9: Função parseGroup

3.7 Representação do Sistema Solar

Tal como anteriormente, o *engine* será responsável por representar a cena, nomeadamente a função *drawScene*. No entanto, para além das figuras a desenhar, tem de se ter atenção às transformações a aplicar. Por isto, esta função irá receber como argumento uma variável *Group** chamada *scene*, que conterá toda a informação obtida através do *parsing* do ficheiro XML.

Antes de desenhar as primitivas, necessitamos de verificar não só as transformações existentes, por forma a realizar as translações/rotações/redimensionamentos necessários, mas também as colorações efetuadas, tendo em conta os parâmetros fornecidos para ambos os casos. Posto isto, desenham-se, então, as diferentes formas geométricas presentes em *scene*.

Para cada figura são representados os diferentes pontos, recorrendo à função *glVertex3f*, de forma semelhante ao efetuado na primeira fase do projeto. Após o desenho do grupo principal, é necessário realizar o mesmo processo para os seus filhos de modo recursivo. É relevante referir que acabou por se tornar fundamental guardar o estado inicial da matriz, repondo-o no fim, uma vez que as transformações aplicadas alteram as posições dos eixos. Isto é possível utilizando as funções *glPushMatrix* e *glPopMatrix*, por esta ordem.

```
void drawScene(Group *scene){
    const char* type;

    glPushMatrix();
    glColor3f(0.5f, 0.5f, 1.0f);

    for(Transformation *t : scene -> getTrans()){
        type = t -> getType().c_str();

        if(!strcmp(type, "translation"))
            glTranslatef(t -> getX(), t -> getY(), t -> getZ());

        else if(!strcmp(type, "rotation"))
            glRotatef(t -> getAngle(), t -> getX(), t -> getY(), t
   -> getZ());

        else if(!strcmp(type, "scale"))
            glScalef(t -> getX(), t -> getY(), t -> getZ());

        else if(!strcmp(type, "colour"))
            glColor3f(t -> getX(), t -> getY(), t -> getZ());
    }

    glBegin(GL_TRIANGLES);

    for(Shape *shape : scene -> getShapes()){
        for(Point *p : shape -> getPoints())
            glVertex3f(p -> getX(), p -> getY(), p -> getZ());
    }

    glEnd();

    for(Group *g : scene -> getGroups())
        drawScene(g);

    glPopMatrix();
}
```

Listing 10: Função drawScene

Achamos também por bem representar as órbitas dos planetas, encontrando-se estas armazenadas numa variável global chamada de *orbits* que é do tipo *vector<Point>*. Este vetor é construído tendo por base as coordenadas usadas para realizar as translações. É importante realçar que apenas são consideradas as translações sobre planetas principais, de forma a que este vetor apenas guarda as informações sobre aqueles que tinham uma profundidade inferior a 2.

Tendo isto em vista, para cada ponto do vetor é calculada a sua distância à origem, obtendo, assim, os diferentes pontos que nos permitem calcular a circunferência que deverá representar a órbita pretendida.

Assim, criamos uma função designada por *drawOrbits*, que é encarregue por desenhar todas as órbitas dos planetas.

```
void drawOrbits(){
    glColor3f(1.0f, 1.0f, 0.94f);

    for(auto const& p : orbits){
        glBegin(GL_POINTS);

        for(int j = 0; j < 200; j++){
            float x = p -> getX() * p -> getX();
            float y = p -> getY() * p -> getY();
            float z = p -> getZ() * p -> getZ();
            float radius = sqrtf(x + y + z);
            float alpha = (float)j * 2 * (float)(M_PI / 200);
            glVertex3f(radius * cos(alpha), 0, radius * sin(alpha));
        }

        glEnd();
    }
}
```

Listing 11: Função drawOrbits

4 Demonstração

Por forma a gerar os resultados pretendidos, torna-se necessário executar os seguintes comandos, a partir da pasta principal:

- **Gerador**

```
cd generator
mkdir build && cd build
cmake ..
make
./generator sphere 3 20 20 sphere.3d
./generator torus 0.5 3 20 20 torus.3d
```

Figura 3: Comandos para gerar os ficheiros .3d necessários

O projeto inclui uma pasta designada por *Files3d* onde serão inseridos todos os ficheiros .3d criados, para posteriormente serem lidos pelo *engine*.

- ***Engine***

```
cd engine
mkdir build && cd build
cmake ..
make
./engine solarsystem.xml
```

Figura 4: Comandos para gerar o Sistema Solar

Passando como parâmetro o nome do ficheiro XML que se pretende ler (este deverá estar guardado na pasta *XML's* incluída no projeto), o *engine* irá ler esse ficheiro, de modo a gerar o Sistema Solar.

4.1 Menus

Mantivemos o menu criado na primeira fase, atualizando-o agora com as novas funcionalidades implementadas. Este menu aparece quando o utilizador, ao executar o *engine*, passa como argumento *-h* em vez do nome de um ficheiro XML.

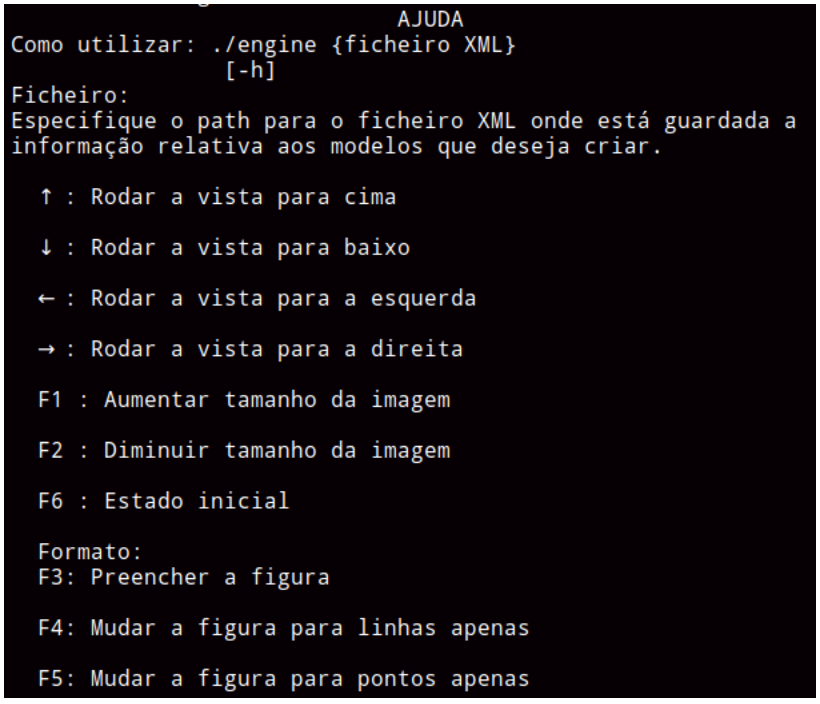

Figura 5: Menu Principal

Para além deste, criamos também um menu no próprio Sistema Solar, para facilitar a interação com o utilizador. Este menu aparece quando pressionamos o botão direito do rato. Os parâmetros de "Planeta"foram criados de forma *responsive*. Deste modo, e usando como exemplo a figura apresentada a seguir, uma vez que existem 9 planetas, são apresentadas 10 opções: uma para o sol e as restantes para cada um dos planetas. Ao selecionar um planeta, o programa foca nesse mesmo planeta.

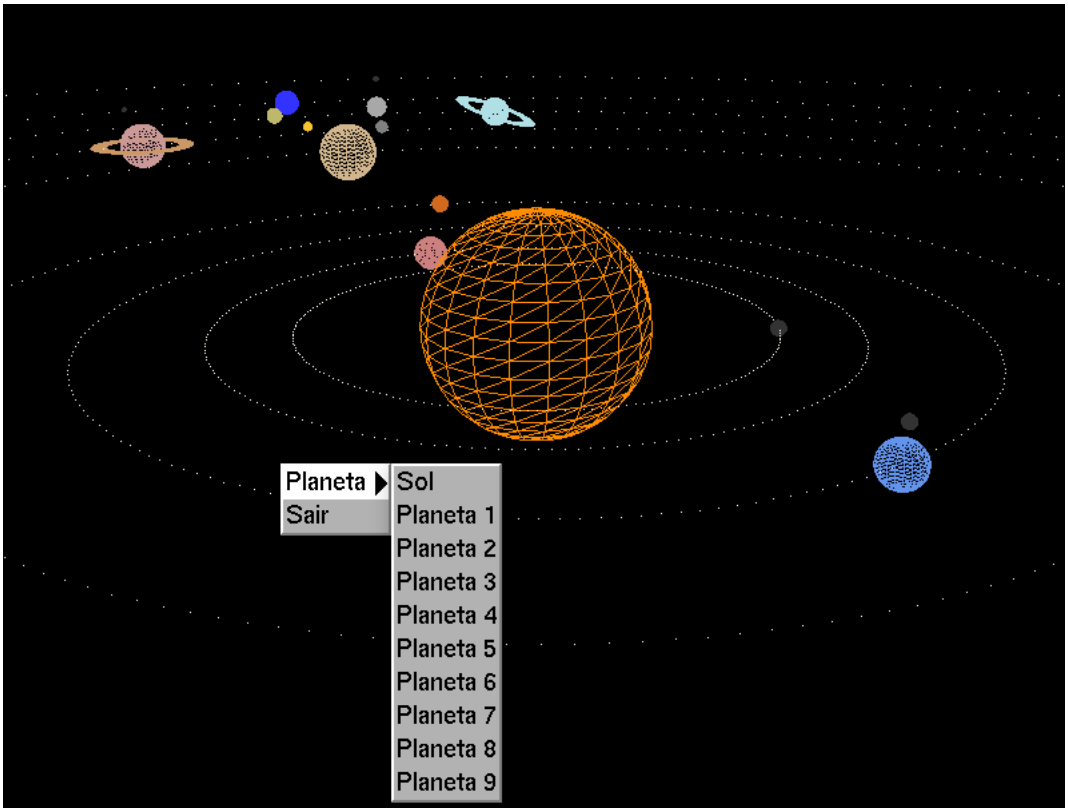

Figura 6: Menu dos Planetas

4.2 Sistema Solar

Tendo em vista a construção do Sistema Solar, tivemos em conta os planetas principais que o constituem (Mercúrio, Vénus, Terra, Marte, Júpiter, Saturno, Urano e Neptuno), a sua disposição, forma e cor. Optamos por não fazer o Sistema Solar à escala, isto é, não respeitamos totalmente as dimensões dos astros nem as distâncias que os separam, uma vez que, caso o fizessemos, ficaríamos com um cenário muito disperso e de difícil observação, acreditando nós não ser este o objetivo do projeto.

No entanto, tivemos em consideração a adição de alguns extras, como por exemplo alguns satélites naturais (a Lua e outros satélites de Júpiter e Saturno) bem como os anéis de Saturno e Urano (estes criados a partir do *torus* gerado). Acrescentamos também o planeta anão Plutão.

Tendo tudo isto em vista, formulamos um ficheiro XML que permita a construção do Sistema Solar, tendo em atenção tudo o que fora referido previamente. Deste modo, podemos obter o resultado seguidamente apresentado.

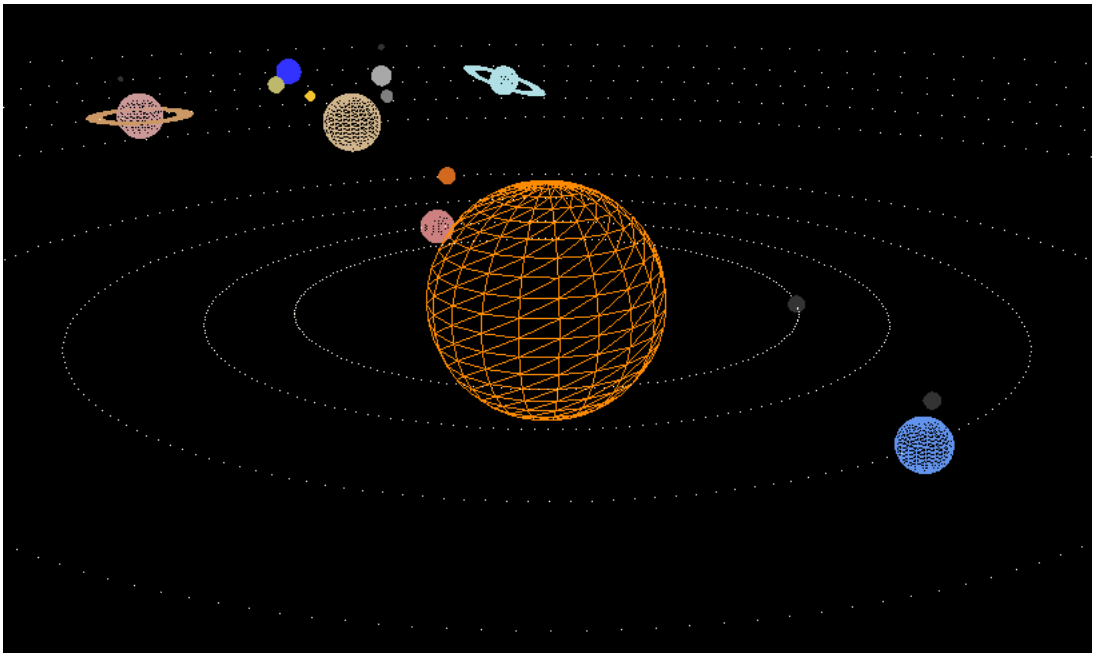

Figura 7: Sistema Solar

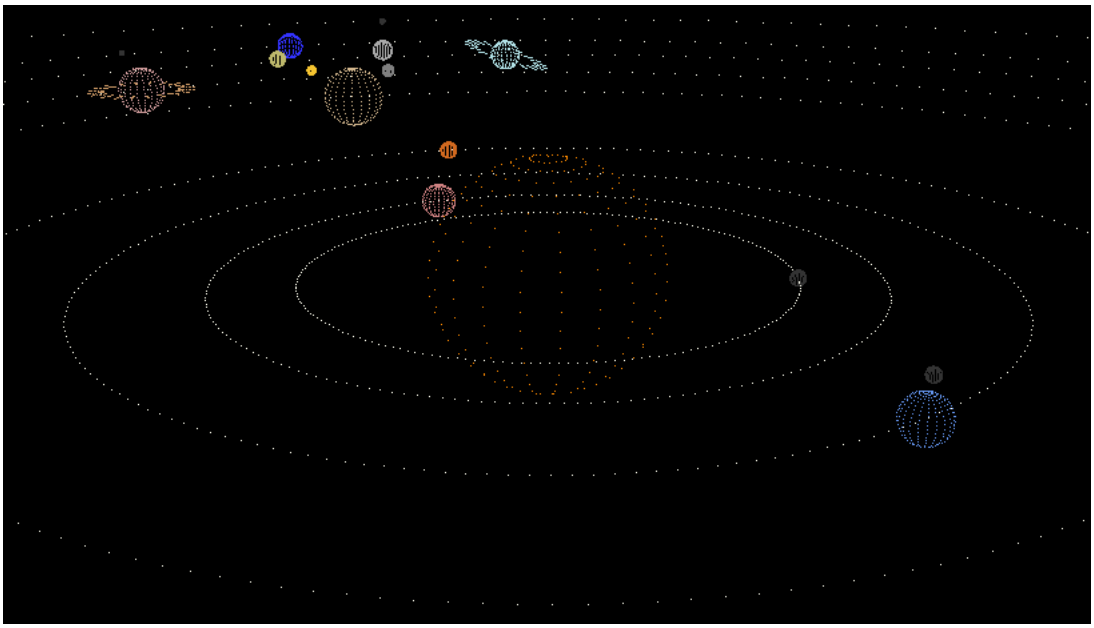

Figura 8: Sistema Solar com pontos

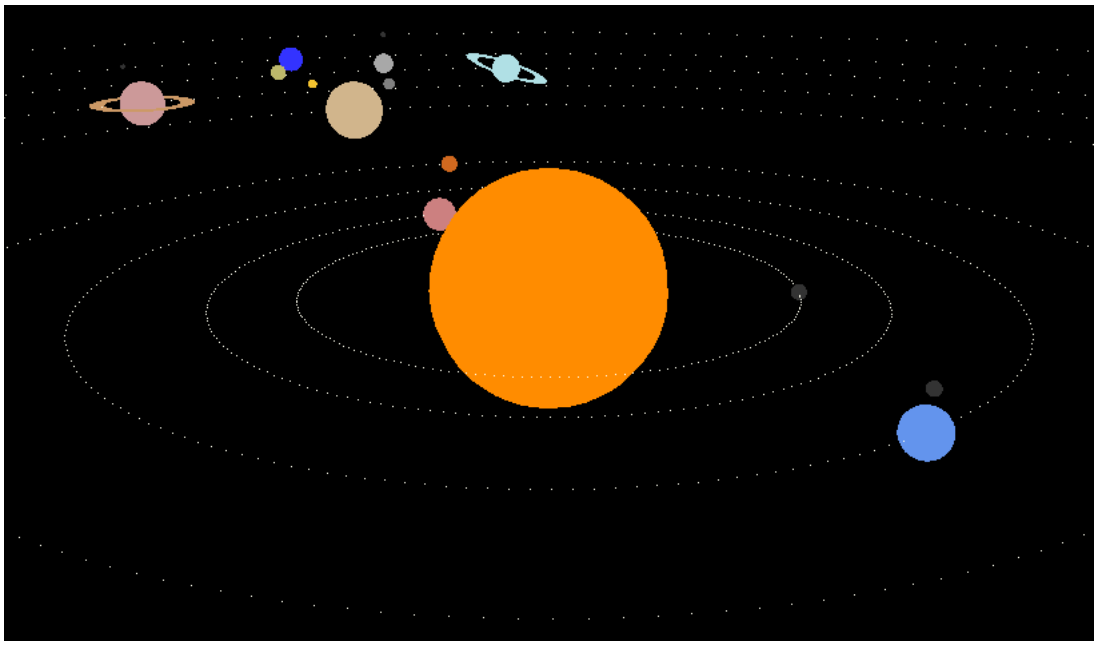

Figura 9: Sistema Solar preenchido

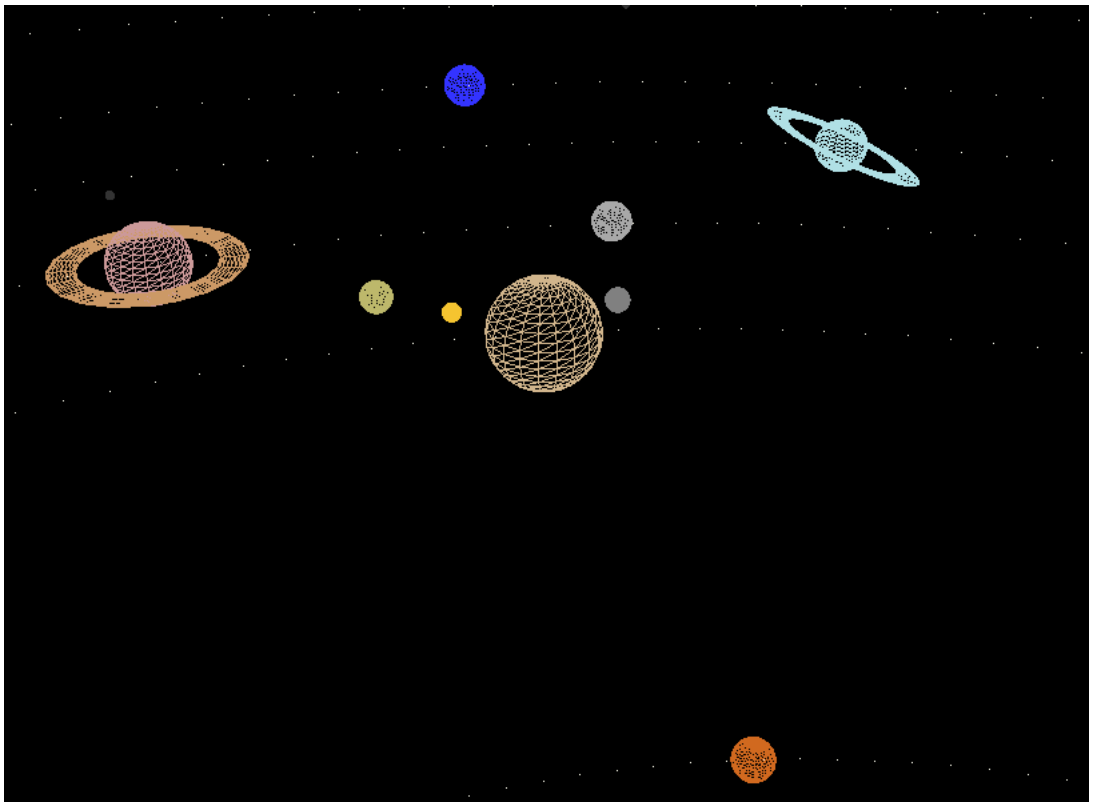

Figura 10: Sistema Solar na perspetiva de Júpiter

5 Conclusão

Para esta segunda fase podemos afirmar que cumprimos com os objetivos estabelecidos, uma vez que adicionamos as funcionalidades pretendidas, isto é, a capacidade de realizar transformações geométricas e de atribuir diferentes cores às figuras. Para além disto, também criamos um ficheiro XML capaz de suportar as informações necessárias à geração do Sistema Solar, bem como um *parser* que torna capaz o processamento do mesmo. Criamos novas classes que facilitam todo este processo e utilizamos também formas mais corretas de armazenamento dos diferentes pontos, recorrendo ao uso de vetores.