Universidade do Minho
Escola de Engenharia

Computação Gráfica

Licenciatura em Engenharia Informática

Grupo 41

2022/2023

2ª Fase

Alexandra Santos (A94523)
Inês Ferreira (A97040)
José Rafael Ferreira (A97642)
Marta Sá (A97158)

ÍNDICE

INTRODUÇÃO

Esta 2º fase do trabalho prático consistiu na continuação do trabalho realizado na 1º fase, agora usando algumas transformações geométricas essenciais para a implementação do Sistema Solar proposto.

Para realizar esta fase, foi necessário considerar que os ficheiros XML possuem uma estrutura hierárquica em árvore, na qual cada um dos nós incorpora um ficheiro .3d que especifica os vértices de um modelo e as transformações a ele aplicadas. Para além disso, estas transformações são obrigatoriamente aplicadas nos modelos presentes nos nós filhos.

Desta forma, o Sistema Solar desenvolvido nesta fase permite a visualização do Sol, dos planetas e das respectivas luas.

FICHEIRO XML

A nível da leitura de ficheiros XML ocorreram algumas mudanças nesta segunda fase, entre elas, a possibilidade de aplicar translações, rotações e escalas a objetos 3D especificados no ficheiro.

Assim, considerando a estrutura do ficheiro XML, dividiu-se a sua leitura em diferentes fases:

- Fase responsável pela obtenção dos dados necessários para configurar a janela de visualização e a câmera. Esta fase manteve-se inalterada comparativamente ao trabalho feito na primeira parte do trabalho.
- Fase responsável por obter os grupos e subgrupos definidos no ficheiro. Para implementar esta funcionalidade criamos a função *loadGroup* que recebe como parâmetro um nodo de forma a permitir fazer um *parse* recursivo sobre os subgrupos desse nodo. Para além disso, esta função é também responsável pelo *parse* de cada elemento presente no grupo (figuras e transformações).

```
// Vetor para armazenar os subgrupos do grupo
vector<Group> subGroups;

xml_node<>* subGroup = group_node->first_node("group");
while (subGroup) {
    Group g = loadGroup(subGroup);
    subGroups.push_back(g);
    subGroup = subGroup->next_sibling("group");
}

// Criação do grupo com os seus elementos
Group grupo(figures, subGroups, transformations);

return grupo;
```

Figura 1 - *parse* recursivo dos subgrupos.

```
    // Vetor para armazenar as figuras do grupo
    vector<Figure> figures;

    xml_node<>* models = group_node->first_node("models");
    if (models) {
        xml_node<>* model = models->first_node("model");

        while (model) {
            string figure = model->first_attribute("file")->value();

            // Abrir e guardar a informação do ficheiro
            string file_3d = "../../Generator/models/" + figure;

            figures = readFile(file_3d);
            if (figure.empty()) {
                cout << "Erro na leitura do ficheiro: " + figure << endl;
            }

            model = model->next_sibling();
        }
    }
```

```
// Função que faz o parser dos ficheiros .3d
vector<Figure> readFile(string file_3d) {
        vector<Point> points;
        vector<Figure> figures;
        int numVertices;
        float x, y, z;

        // Abre o ficheiro
        ifstream ficheiro(file_3d);
        if (!ficheiro.is_open()) {
            cout << "Erro ao abrir o ficheiro " << file_3d << endl;
            return figures;
        }

        // Armazena o número de vértices que a figura irá ter
        ficheiro >> numVertices;
        for (int i = 0; i < numVertices; i++) {
            // Lê do ficheiro as coordenadas
            ficheiro >> x >> y >> z;

            // Cria um objeto ponto com as coordenadas lidas
            Point p(x,y,z);

            // Armazena cada ponto no vetor points
            points.push_back(p);
        }
        // Cria uma figura através de um vetor de pontos
        Figure fig(points);

        // Armazena a figura criada num vetor de figuras
        figures.push_back(fig);

        ficheiro.close();

        return figures;
```

Figura 2 - *parse* dos modelos.

```
//
Group loadGroup(xml_node<>* group_node) {
    // Vetor para armazenar as transformações do grupo
    vector<Transformation> transformations;

    xml_node<>* transform_node = group_node->first_node("transform");
    while (transform_node) {
        // Analisa cada nodo dentro do bloco transform
        xml_node<>* node = transform_node->first_node();
        while (node) {
            // Transformação
            Transformation transform;

            // Vetor para armazenar os parâmetros da transformação
            vector <float> params;

            // Parse do elemneto translate
            if (strcmp(node->name(), "translate") == 0) {
                params.push_back(atof(node->first_attribute("x")->value()));
                params.push_back(atof(node->first_attribute("y")->value()));
                params.push_back(atof(node->first_attribute("z")->value()));
                transform.setTipo("translate");
                transform.setParams(params);
                transformations.push_back(transform);
            }

            // Parse do elemento rotate
            if (strcmp(node->name(), "rotate") == 0) {
                params.push_back(atof(node->first_attribute("angle")->value()));
                params.push_back(atof(node->first_attribute("x")->value()));
                params.push_back(atof(node->first_attribute("y")->value()));
                params.push_back(atof(node->first_attribute("z")->value()));
                transform.setTipo("rotate");
                transform.setParams(params);
                transformations.push_back(transform);
            }

            // Parse do elemento scale
            if (strcmp(node->name(), "scale") == 0) {
                params.push_back(atof(node->first_attribute("x")->value()));
                params.push_back(atof(node->first_attribute("y")->value()));
                params.push_back(atof(node->first_attribute("z")->value()));
                transform.setTipo("scale");
                transform.setParams(params);
                transformations.push_back(transform);
            }
```

```
            // Parse do elemento color
            if (strcmp(node->name(), "color") == 0) {
                params.push_back(atof(node->first_attribute("x")->value()));
                params.push_back(atof(node->first_attribute("y")->value()));
                params.push_back(atof(node->first_attribute("z")->value()));
                transform.setTipo("color");
                transform.setParams(params);
                transformations.push_back(transform);
            }
            node = node->next_sibling();
        }
        transform_node = transform_node->next_sibling("transform");
    }
```

Figura 3 - *parse* das transformações.

ESTRUTURAS DE DADOS

Para armazenar em memória os dados lidos aquando a leitura do ficheiro XML, criou-se diferentes estruturas representadas da seguinte forma:

```
class Point {

private:
    float x;
    float y;
    float z;
```

```
class Figure {

private:
    vector <Point> vertices;
```

Figura 4 - classes *Point* e *Figure*.

Estas classes, responsáveis pelo armazenamento de coordenadas formando pontos e de pontos formando figuras, mantêm-se inalteradas relativamente à primeira fase.

```
class Transformation {

private:
    string tipo;
    vector <float> params;
```

```
class Group {

public:
    vector<Transformation> transformations;
    vector<Figure> figures;
    vector<Group> subGroups;
```

Figura 5 - classes *Transformation* e *Group*.

Em contrapartida, as classes *Transformation* e *Group* foram criadas nesta fase com o objetivo de armazenar e organizar os dados lidos.

A classe *Transformation* é constituída por um tipo podendo este ser uma rotação, uma translação, uma escala ou uma cor e por um vetor de *floats* que correspondem aos parâmetros de cada transformação. A transformação "cor" foi adicionada nesta fase de forma a melhorar a visualização do modelo. Para além disso, nesta classe são definidas as funções necessárias para aceder e definir cada variável.

Já a classe *Group* é constituída por um vetor de transformações, de modelos e de subgrupos.

Desta forma, facilitamos o acesso a cada elemento do ficheiro lido uma vez que a informação é armazenada, ao longo do programa, numa variável global da classe *group*.

TORUS

Para esta fase foi necessário criar um modelo de um sistema solar com o Sol, os planetas e as respectivas luas. No entanto, uma vez que não havia nenhuma primitiva que servisse para desenhar os anéis de Saturno, decidimos adicionar ao *generator* a primitiva *Torus*. Esta primitiva recebe como argumentos o raio interno da circunferência, a distância entre o centro do torus e o centro da circunferência interior - denominado por raio externo, o número de *slices*, o número de *stacks* e o ficheiro .3d de *output* onde serão guardados os pontos da figura.

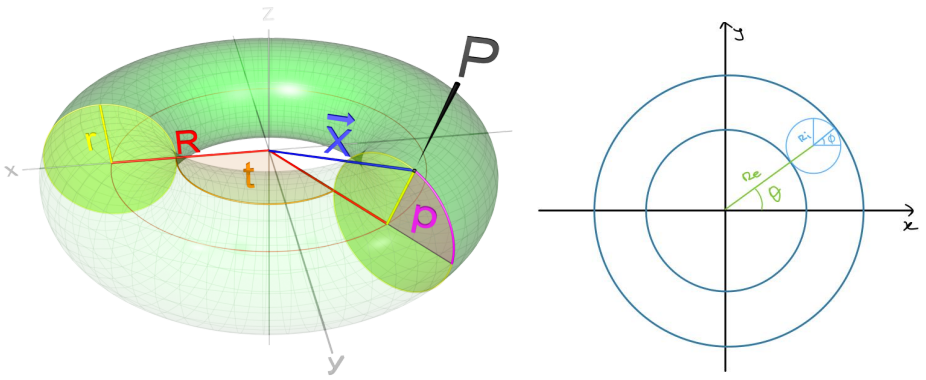


Figura 6 - Topologia do *torus*.

O *torus* é gerado de acordo com as seguintes coordenadas, sendo r (ri) o raio interno e R (re) o raio externo:

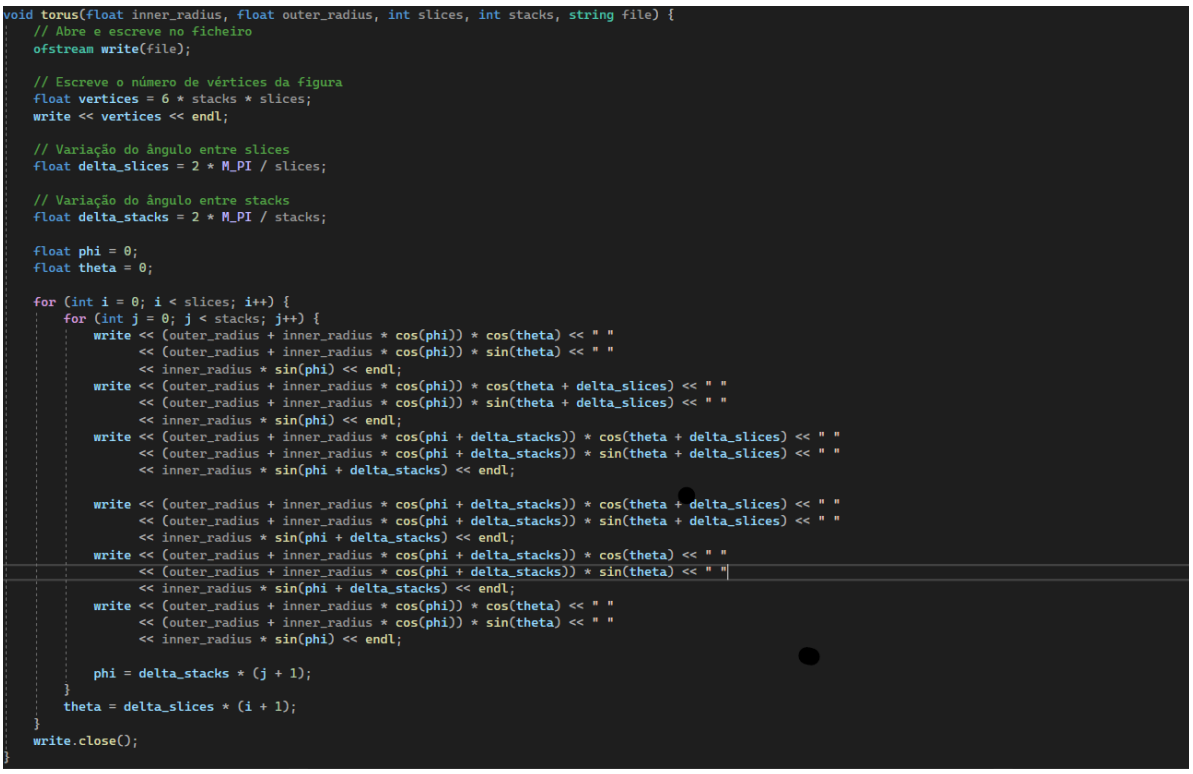
$$x = (R + r\,cos\Phi)\,cos\theta$$
$$y = (R + r\,cos\Phi)\,sen\theta$$
$$z = r\,sen\Phi$$

em que,

Φ, θ pertencem ao intervalo [0, 2π],

R (re) é a distância do centro do tubo ao centro do *torus* - raio externo,

r (ri) é o raio do tubo - raio interno.



```
void torus(float inner_radius, float outer_radius, int slices, int stacks, string file) {
    // Abre e escreve no ficheiro
    ofstream write(file);

    // Escreve o número de vértices da figura
    float vertices = 6 * stacks * slices;
    write << vertices << endl;

    // Variação do ângulo entre slices
    float delta_slices = 2 * M_PI / slices;

    // Variação do ângulo entre stacks
    float delta_stacks = 2 * M_PI / stacks;

    float phi = 0;
    float theta = 0;

    for (int i = 0; i < slices; i++) {
        for (int j = 0; j < stacks; j++) {
            write << (outer_radius + inner_radius * cos(phi)) * cos(theta) << " "
                  << (outer_radius + inner_radius * cos(phi)) * sin(theta) << " "
                  << inner_radius * sin(phi) << endl;
            write << (outer_radius + inner_radius * cos(phi)) * cos(theta + delta_slices) << " "
                  << (outer_radius + inner_radius * cos(phi)) * sin(theta + delta_slices) << " "
                  << inner_radius * sin(phi) << endl;
            write << (outer_radius + inner_radius * cos(phi + delta_stacks)) * cos(theta + delta_slices) << " "
                  << (outer_radius + inner_radius * cos(phi + delta_stacks)) * sin(theta + delta_slices) << " "
                  << inner_radius * sin(phi + delta_stacks) << endl;

            write << (outer_radius + inner_radius * cos(phi + delta_stacks)) * cos(theta + delta_slices) << " "
                  << (outer_radius + inner_radius * cos(phi + delta_stacks)) * sin(theta + delta_slices) << " "
                  << inner_radius * sin(phi + delta_stacks) << endl;
            write << (outer_radius + inner_radius * cos(phi + delta_stacks)) * cos(theta) << " "
                  << (outer_radius + inner_radius * cos(phi + delta_stacks)) * sin(theta) << " "
                  << inner_radius * sin(phi + delta_stacks) << endl;
            write << (outer_radius + inner_radius * cos(phi)) * cos(theta) << " "
                  << (outer_radius + inner_radius * cos(phi)) * sin(theta) << " "
                  << inner_radius * sin(phi) << endl;

            phi = delta_stacks * (j + 1);
        }
        theta = delta_slices * (i + 1);
    }
    write.close();
}
```

Figura 7 - programação do *torus* em *C++*.

É possível observar que foram realizados dois ciclos for, no qual o primeiro percorre as *slices* e o segundo percorre as *stacks*. A cada iteração são desenhados dois triângulos que correspondem ao plano delimitado pela *stack* e pela *slice* correspondente.

Tal como referido em cima, os ângulos θ *e* Φ pertencem ao intervalo [0 , 2π], sendo que a cada iteração pelas *stacks* Φ = *delta_stacks* $*$ (j + 1) e a cada iteração pelas *slices* θ = *delta_slices* $*$ (i + 1).

DESENHO DOS MODELOS

Para desenhar os triângulos no ecrã, adaptou-se a função *drawFigures* criada na primeira fase de forma a aplicar a cada figura as correspondentes transformações. Nesta função, temos que ter em conta que cada transformação aplica-se apenas aos modelos presentes no próprio grupo, sendo que deve ser revertida quando analisamos o próximo grupo sequencial. Para isso, recorremos à função *glPushMatrix*, chamada no início da análise do grupo, e *glPopMatrix*, chamada no final da análise do grupo, para colocar e retirar, respetivamente, a matriz da *stack*.

Deste modo, podemos visualizar o algoritmo a partir da seguinte figura:

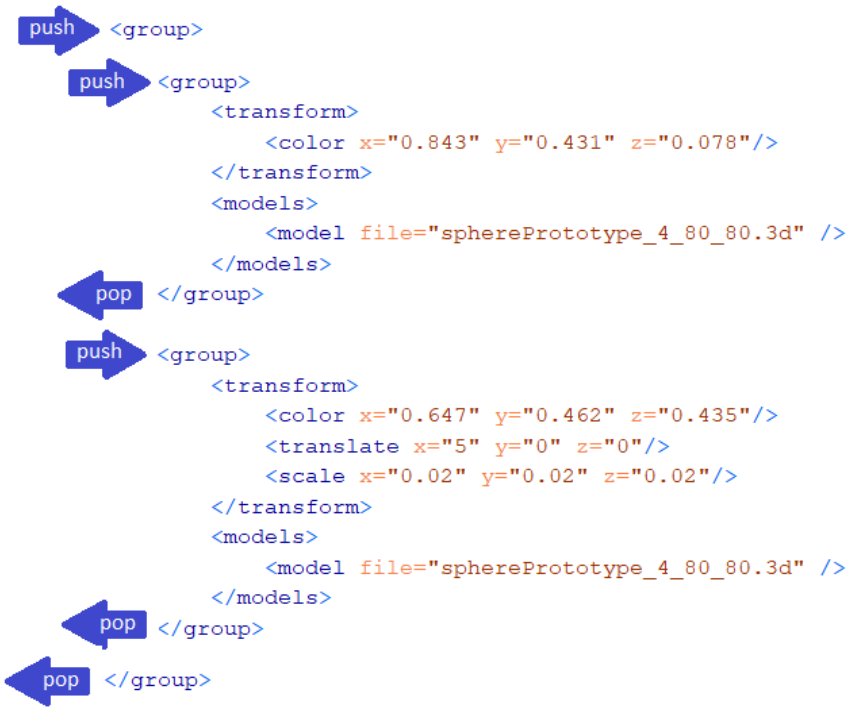


Figura 8 - visualização do algoritmo.

O algoritmo começa com o *push* da matriz original para a *stack*. De seguida, analisamos os elementos do grupo e aplicamos as transformações necessárias, alterando a matriz. Posteriormente, desenhamos as figuras pertencentes ao grupo e verificamos se existem ou não subgrupos. Caso exista algum subgrupo, aplicamos-lhe, de forma recursiva, a função *drawFigure*. Por fim, fazemos *pop* da matriz voltando ao estado anterior.

```
// Função que desenha todas a figuras armazenadas previamente num vetor
void drawFigures(Group g) {
    // Guarda o estado atual da matriz
    glPushMatrix();

    // Percorremos e aplicamos as transformações
    for (int k = 0; k < g.transformations.size(); k++) {
        Transformation transf = g.transformations[k];
        if (transf.getTipo() == "translate") {
            float x = transf.getParams()[0];
            float y = transf.getParams()[1];
            float z = transf.getParams()[2];
            glTranslatef(x, y, z);
        }
        else if (transf.getTipo() == "scale") {
            float x = transf.getParams()[0];
            float y = transf.getParams()[1];
            float z = transf.getParams()[2];
            glScalef(x, y, z);
        }
        else if (transf.getTipo() == "rotate") {
            float angle = transf.getParams()[0];
            float x = transf.getParams()[1];
            float y = transf.getParams()[2];
            float z = transf.getParams()[3];
            glRotatef(angle, x, y, z);
        }
        else if (transf.getTipo() == "color") {
            float red = transf.getParams()[0];
            float green = transf.getParams()[1];
            float blue = transf.getParams()[2];
            glColor3f(red, green, blue);
        }
    }
```

```
    // Percorremos e desenhamos as figuras
    glBegin(GL_TRIANGLES);
    for (int i = 0; i < g.figures.size(); i++) {
        Figure fig = g.figures[i];
        for (int j = 0; j < fig.getNrVertices(); j++) {
            Point point = fig.getPoint(j);
            glVertex3f(point.getX(), point.getY(), point.getZ());
        }
    }
    glEnd();

    // Percorremos e analisamos os subgrupos
    if (!g.subGroups.empty()) {
        for (int i = 0; i < g.subGroups.size(); i++) {
            drawFigures(g.subGroups[i]);
        }
    }

    // Voltamos ao estado anterior
    glPopMatrix();
    glClearColor;
}
```

Figura 9 - Função *drawFigures* em *C++*.

SISTEMA SOLAR

De modo a cumprir o objetivo desta segunda fase do projeto, definiu-se um ficheiro XML, a ser processado pelo *engine*, que permite desenhar o Sistema Solar.

Relativamente à formatação do ficheiro XML, começamos por recorrer ao *generator* para gerar o ficheiro *spherePrototype_4_80_80.3d* (esfera com raio 4, 80 *stacks* e 80 *slices*) que servirá de base para a construção do Sol, dos planetas e das luas. Assim, somos capazes de definir grupos que contêm as transformações necessárias a serem aplicadas a cada modelo representado pelo referido ficheiro .3d. Na figura abaixo podemos verificar uma parte do ficheiro XML utilizado para construir o Sistema Solar, no caso, com a definição do Sol, da Terra, da Lua e de Neptuno:

```
<group>
    <!-- SOL -->
    <group>
        <transform>
            <color x="0.843" y="0.431" z="0.078"/>
            <scale x="1.5" y="1.5" z="1.5"/>
        </transform>
        <models>
            <model file="spherePrototype_4_80_80.3d" />
        </models>
    </group>

    <!-- TERRA -->
    <group>
        <transform>
            <color x="0.058" y="0.156" z="0.423"/>
            <rotate angle="90" x="0" y ="1" z = "0"/>
            <translate x="9" y="0" z="0"/>
            <scale x="0.05" y="0.05" z="0.05"/>
        </transform>
        <models>
            <model file="spherePrototype_4_80_80.3d" />
        </models>

        <!-- LUA -->
        <group>
            <transform>
                <color x="1" y="1" z="1"/>
                <rotate angle="45" x="0" y ="0" z = "1"/>
                <translate x="12" y="0" z="0"/>
                <scale x="0.25" y="0.25" z="0.25"/>
            </transform>
            <models>
                <model file="spherePrototype_4_80_80.3d" />
            </models>
        </group>
    </group>
```

```
        <!-- NEPTUNO -->
        <group>
            <transform>
                <color x="0.203" y="0.133" z="0.749"/>
                <rotate angle="315" x="0" y ="1" z = "0"/>
                <translate x="70" y="0" z="0"/>
                <scale x="0.175" y="0.175" z="0.175"/>
            </transform>
            <models>
                <model file="spherePrototype_4_80_80.3d" />
            </models>
        </group>

    </group>
```

Figura 10 - excerto do ficheiro XML.

Para definir Saturno foi necessário construir o seu anel a partir do ficheiro *rings_0.7_5_80_3.3d* onde se encontram os pontos calculados às custas de um *Torus* (com raio interno 0.7, raio externo 5, 80 *slices* e 3 *stacks*) definido no *generator*.

De modo a tornar o modelo o mais realista e preciso possível, utilizou-se as distâncias até ao Sol e os tamanhos reais dos astros para definir as escalas dos planetas e das luas e as translações dos planetas (normalizadas com valores entre 7 e 70). Os valores calculados e aplicados no ficheiro XML estão representados nas tabelas abaixo:

Estrela	Diâmetro (km)	Raio (km)	Raio Escala (cm)
Sol	1.390.000	695.000	4

Tabela 1 - informações do Sol.

Planetas	Diâmetro (km)	Raio (km)	Raio Escala (cm)	Distância média ao Sol (km)	Distância ao Sol Escala (cm)
Mercúrio	4.879,4	2.439,7	0.0125	57.910.000	7
Vénus	12.103,6	6.051,8	0.0375	108.200.000	8
Terra	12.756,28	6.378,14	0.05	149.600.000	9
Marte	6.794,4	3.378,2	0.025	227.940.000	10
Júpiter	142.984	71.492	0.5125	778.330.000	19
Saturno	120.536	60.268	0.4375	1.429.400.000	30
Urano	51.118	25.559	0.1875	2.870.990.000	53
Neptuno	49.538	24.746	0.175	4.504.300.000	70

Tabela 2 - informações dos planetas.

As escalas das luas foram atribuídas em relação ao tamanho do planeta a que pertencem:

Luas	Diâmetro (km)	Raio (km)	Raio Escala (cm)
Lua (Terra)	3.500	1.728,1	0.25
Europa (Jupiter)	3.121,6	1.560,8	0.20
Ganimedes (Júpiter)	5.262,4	2.634,1	0.04
Io (Júpiter)	3.643,2	1.821,6	0.03
Calisto (Júpiter)	4.820,6	2.410,3	0.03
Encélado (Saturno)	504,2	252,1	0.004
Titã (Saturno)	5.151	2.574,4	0.04

Tabela 3 - informações de algumas luas.

Para além disso, desenhou-se a cintura de asteroides entre Marte e Júpiter a partir da função *drawAsteroids*:

```
// Função que desenha o anel de asteroides
void drawAsteroids() {
    glColor3f(0.478f, 0.458f, 0.458f);

    srand(100000);
    for (int i = 0; i < 500; i++) {
        float x = 30 * (rand() / (float)RAND_MAX) - 15;  // x coordinate between -15 and 15
        float y = 4 * (rand() / (float)RAND_MAX) - 2;    // y coordinate between -2 and 2
        float z = 30 * (rand() / (float)RAND_MAX) - 15;  // z coordinate between -15 and 15

        if (169 < x*x + z*z && x*x + z*z <= 225) {
            glPushMatrix();
            glTranslatef(x, y, z);
            glScalef(0.01, 0.01, 0.01);
            glutSolidSphere(4, 80, 80);
            glPopMatrix();
        }
    }
    glClearColor;
}
```

Figura 11 - função *drawAsteroids*.

RESULTADO FINAL

A execução do programa gera o seguinte *output*:

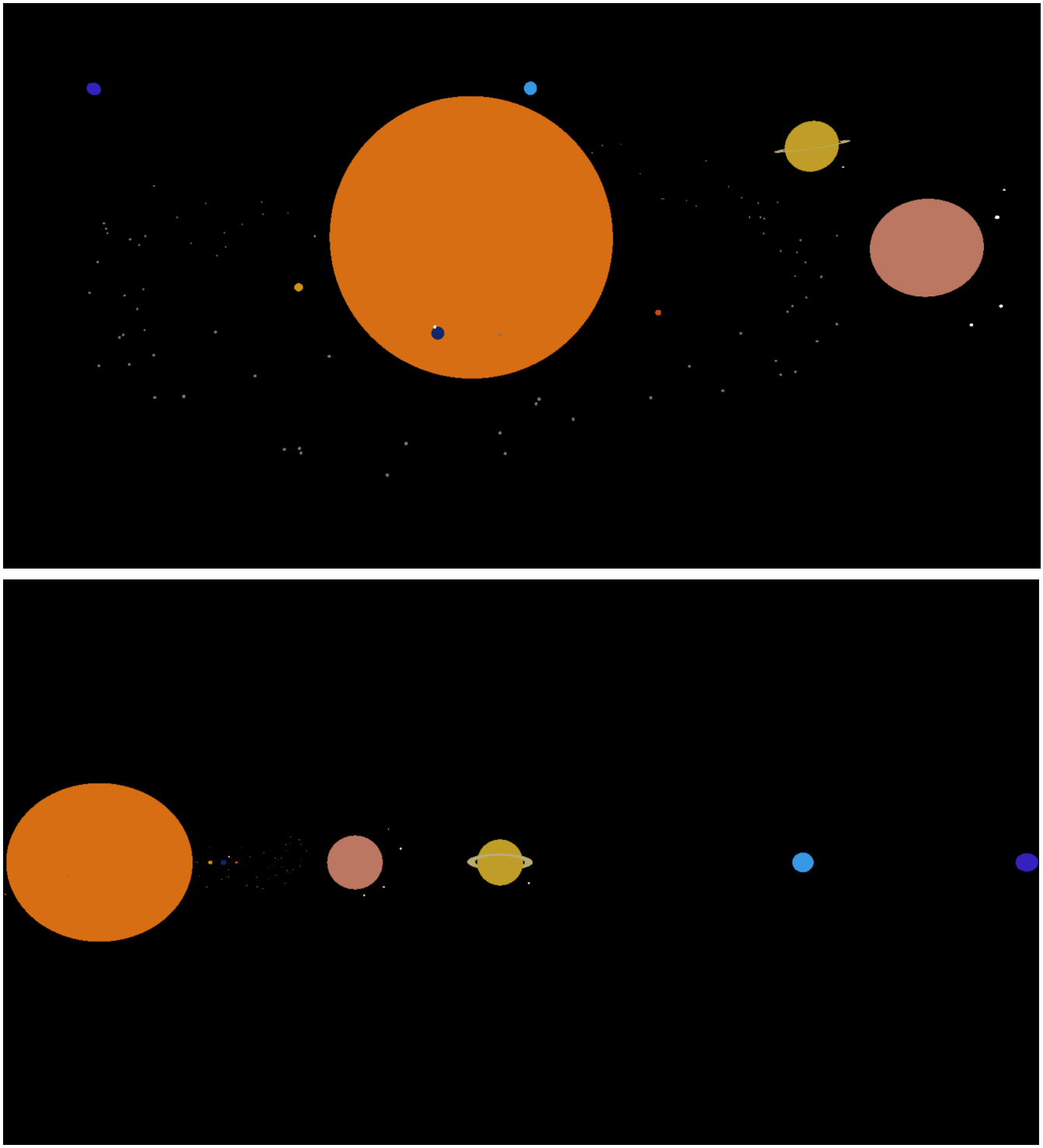


Figura 12 - Sistema Solar.

FUNCIONALIDADES

Durante o desenvolvimento desta fase foram implementados comandos que permitem ao usuário a interação com a renderização obtida. Assim, a seguinte tabela especifica as funcionalidades, especificadas pelas respectivas teclas, de forma a modificar a visualização do modelo completo:

Tecla	Funcionalidade
w	Mover os objetos no sentido positivo do eixo do x.
s	Mover os objetos no sentido negativo do eixo do x.
d	Mover os objetos no sentido positivo do eixo do z.
a	Mover os objetos no sentido negativo do eixo do z.
1	Mudar o modo de desenho dos objetos para GL_LINE.
2	Mudar o modo de desenho dos objetos para GL_POINT.
3	Mudar o modo de desenho dos objetos para GL_FILL.
e	Mostrar/esconder os eixos de coordenadas.
Esc	Fechar a janela de visualização.
GLUT_KEY_UP	Rotação da câmera para cima.
GLUT_KEY_DOWN	Rotação da câmera para baixo.
GLUT_KEY_RIGHT	Rotação da câmera para a direita.
GLUT_KEY_LEFT	Rotação da câmera para a esquerda.
Rato + GLUT_DOWN	Movimentar a câmera em todas as direções.

Tabela 4 - funcionalidades do sistema.

CONCLUSÃO

Concluída esta segunda fase do projeto, foi possível aplicar e consolidar vários conceitos abordados nas aulas teóricas e práticas, bem como aprofundar o nosso conhecimento em OpenGL.

Por outro lado, também existiram algumas dificuldades, tais como definir as escalas a utilizar, sendo este um sistema com distâncias tão grandes que a escolha seria sempre algo importante. Como tal, a nossa representação foi feita da forma que consideramos mais eficiente embora não tenha uma escala idêntica em todos os seus elementos.

Por fim, consideramos que houve um balanço positivo do trabalho realizado dado que as dificuldades sentidas foram superadas e foram cumpridos todos os requisitos.