

Universidade do Minho

Licenciatura em Engenharia Informática

Computação Gráfica

Phase 2 - Geometric Transforms

Grupo 12

Ana Murta (A93284)
Ana Henriques (A93268)
Leonardo Freitas (A93281)
Rui Coelho (A58898)

abril, 2022

Conteúdo

1	Introdução	4
2	Engine	5
3	Ficheiro XML	9
4	Extras	11
4.1	Elipsóide - Extra	11
4.2	Semiesfera - Extra	13
4.3	Câmera	15
4.4	Cintura de asteróides	16
5	Sistema Solar	18
6	Conclusão	20

Lista de Figuras

4.1	Parâmetros do elipsóide.	11
4.2	Equações paramétricas.	11
4.3	ellipsoid 2 1 10 ellipsoid.3d	12
4.4	ellipsoid 1 2 10 ellipsoid.3d	13
4.5	Estratégia de desenho da base da semiesfera	14
4.6	hsphere 1 10 10 hsphere.3d	14
4.7	<i>Zoom in</i> do Sistema Solar.	16
5.1	Sistema Solar	18
5.2	Sistema Solar gerado através do ficheiro <i>XML</i> .	19

Capítulo 1

Introdução

Na segunda fase do trabalho prático, realizado no âmbito da disciplina de Computação Gráfica, foi desenvolvida uma primeira versão do cenário gráfico 3D representativo do Sistema Solar. O cenário apresentado recorre a algumas das primitivas gráficas anteriormente desenvolvidas na primeira fase do projeto em curso – como o caso da representação das esferas e do toro.

A construção deste modelo do Sistema Solar assenta na leitura e interpretação de um ficheiro *XML* que indica os modelos que devem ser representados e as respetivas transformações a que estes são submetidos. Como tal, foi imperativa a atualização de alguns componentes previamente desenvolvidos na fase anterior – e.g., a componente *Engine* previamente apresentada sofreu algumas alterações. Adicionalmente, foram incorporadas funcionalidades adicionais como o desenho de elipsóides e a rotação da câmera.

Ao longo do presente relatório, é dado a conhecer o trabalho desenvolvido, assim como as estratégias adotadas para o efeito.

Capítulo 2

Engine

Nesta fase do projeto, foi necessário atualizar o programa `Engine` de maneira que este consiga ler e processar (apenas uma vez) o novo ficheiro XML. Para isto, criou-se uma classe `Group` que facilitou o armazenamento da informação do ficheiro. Isto é, no ficheiro `main.cpp`, implementou-se um vetor em que cada elemento é da classe `Group`, permitindo guardar nesta estrutura os valores das transformações geométricas e as primitivas, uma vez que esta classe é composta por vetores de outras classes, nomeadamente da `Trans` e `Primitive`.

O ficheiro XML é constituído por grupos que contém transformações, primitivas e, em alguns casos, subgrupos, como irá ser explicado mais à frente. Como referido em cima, é possível armazenar, de forma organizada, os dados presentes no ficheiro através do uso de um vetor com elementos da classe `Group`. A informação contida no dito ficheiro resume-se às transformações, às primitivas e, opcionalmente, aos subgrupos e, como tal, a classe `Group` utiliza 3 vetores: o `trans`, com elementos da classe `Trans`, o `primitives` com elementos da classe `Primitive` e, finalmente, o `groups`, que contém elementos da classe `Group` caso existam subgrupos.

Através classe `Trans` conseguimos distinguir a transformação de que se trata e as suas características, uma vez que esta classe implementa uma variável do tipo `string` que identifica o tipo da transformação: `translate`, `scale`, `rotate` ou `color`. Para além desta variável, esta classe ainda implementa 4 variáveis do tipo `float` que guardam, então, as características da transformação.

A class `Primitive`, usada na primeira fase do projeto, implementa um vetor em que cada elemento é da classe `Point`, permitindo armazenar todos os pontos existentes nos ficheiros `.3d`.

Leitura do Ficheiro XML

Nesta segunda fase, considerando o novo formato do ficheiro XML, foi preciso modificar o modo como este era lido. Deste modo, utilizou-se as funções presentes na biblioteca `tinyXML2`.

A partir da função `parseDocument()` abrimos o ficheiro XML que pretendemos ler, neste caso o `solarSystem.xml`. Ainda dentro desta função, temos a `FirstChildElement()`, que obtém o primeiro nodo definido com a string `world`. Seguidamente, lê-mos, da mesma forma, do XML, o nodo `camera` e `group`, invocando a seguir as funções `parseCamera()` e `parseGroup()`, que iriam ocupar-se da informação presente dentro de cada um destes nodos. Quanto à função `parseGroup()`, para além de se passar como argumento o nodo `group`, houve a necessidade, devido à recursividade da função, de passar como argumento um identificador que distinguisse o grupo pai do grupo filho. Para ambas as funções, é aplicado a mesma estratégia para ler e guardar o conteúdo do ficheiro.

```

bool parseDocument() {
    (...)

    XMLElement* world = doc.FirstChildElement();
    if (world == nullptr) {
        cout << "ERRO";
        return false;
    }

    XMLElement* camera = world->FirstChildElement("camera");
    parseCamera(camera);
    XMLElement* group = world->FirstChildElement("group");
    parseGroup(group, 0);
    return true;
}

```

A função `parseCamera()` contem um ciclo que permite, de acordo com o elemento em questão, guardar os valores nas variáveis corretas para depois serem usados nas funções `gluLookAt()` e `gluPerspective()`.

```

void parseCamera(XMLElement* camera) {
    XMLElement* element = camera->FirstChildElement();
    while (element != nullptr) {
        if (position.compare(element->Name()) == 0)
            (...)

        else if (lookAt.compare(element->Name()) == 0)
            (...)

        else if (up.compare(element->Name()) == 0)
            (...)

        else if (projection.compare(element->Name()) == 0)
            (...)

        element = element->NextSiblingElement();
    }
}

```

A função `parseGroup()` contem um ciclo exterior que itera os vários grupos do ficheiro XML através de um apontador. Por cada iteração, e para os elementos de cada grupo, é averiguado o tipo de nodo a tratar. Se for um nodo *models*, o apontador guarda a primitiva e o nome do ficheiro .3d. Caso se trate de um nodo *transform*, ele armazena, na estrutura, a transformação geométrica pretendida juntamente com as suas características. Na situação do nodo encontrado ser um subgrupo, a função será chamada recursivamente. No entanto, o valor passado como identificador (father) será 1 visto se tratar de um subgrupo. No fim da leitura do grupo, caso estejamos perante um grupo pai, é guardado esse grupo no vetor com elementos da classe *Group*.

```

Group parseGroup(XMLElement* group, int father) {
    do {
        g = Group();
        XMLElement* element = group->FirstChildElement();

        while (element != nullptr) {
            if (models.compare(element->Name()) == 0) {
                XMLElement* file = element->FirstChildElement("model");
                while (file != nullptr) {

```

```

        (...)

        g.addPrimitives(primitive);
        g.setNameFile(namefile);
        file = file->NextSiblingElement();
    }

} else if (transform.compare(element->Name()) == 0){
    XMLElement* transformation = element->FirstChildElement();
    while(transformation != nullptr){
        if (scale.compare(transformation->Name()) == 0) {
            if (transformation != nullptr) {
                (...)

                Trans t = Trans("scale", x, y, z, 0);
                g.addTrans(t);
            }
        } else if(translate.compare(transformation->Name()) == 0) {
            if (transformation != nullptr) {
                (...)

                Trans t = Trans("translate", x, y, z, 0);
                g.addTrans(t);
            }
        } else if (rotate.compare(transformation->Name()) == 0) {
            if (transformation != nullptr) {
                (...)

                Trans t = Trans("rotate", x, y, z, angle);
                g.addTrans(t);
            }
        } else if (color.compare(transformation->Name()) == 0) {
            (...)

            Trans t = Trans("color", red, green, blue, 0);
            g.addTrans(t);
        }
        transformation = transformation->NextSiblingElement();
    }

} else if (grupo.compare(element->Name()) == 0) {
    Group gr = parseGroup(element, 1);
    g.addGroups(gr);
}

element = element->NextSiblingElement();
if (element == NULL && father == 1) return g;
}

if (father == 0) groups.push_back(g);
group = group->NextSiblingElement();
} while (group != nullptr);
return g;
}

```

Construção do Ficheiro XML

Outro aspecto modificado foi a função `drawPrimitive()`, previamente desenvolvida na primeira fase, para ser possível desenhar as transformações geométricas, as primitivas e os subgrupos.

Recebendo como argumento um grupo e tendo em consideração os dados armazenados, a `drawPrimitive()` desenha as primitivas e executa as transformações geométricas pedidas à medida que percorre o vetor que recebe. Adicionalmente, como queremos que as transformações de cada grupo principal sejam apenas desse grupo, é usada a `glPushMatrix()` para guardar a matriz antes de ocorrer transformações sobre os eixos. É, também, usada a `glPopMatrix()` para que as primitivas e transformações dos elementos seguintes sejam desenhadas sobre os eixos inciais. Quanto ao desenho dos subgrupos, a função será chamada recursivamente.

Para além disto, a `drawPrimitive()` é, ainda, responsável por desenhar a cintura de asteroides. Como a primitiva que identifica um asteróide é a esfera, revelou-se necessário distinguir o processo de desenho da cintura de asteroides do de um planeta ou de sol. Assim sendo, se o nome do ficheiro .3d for `asteroids.3d`, a função calculará os valores passados como argumento à função `glTranslated()`. O processo de desenho da cintura de asteroides será, no entanto, melhor explicado mais à frente no relatório.

```
void drawPrimitives(Group groups) {

    for (int j = 0; j < groups.getNrTrans(); j++) {

        Trans t = groups.getTrans(j);
        if (translate.compare(t.getName()) == 0) {
            glTranslatef(t.getX(), t.getY(), t.getZ());
        } else if (scale.compare(t.getName()) == 0) {
            glScalef(t.getX(), t.getY(), t.getZ());
        } else if (rotate.compare(t.getName()) == 0) {
            glRotated(t.getAngle(), t.getX(), t.getY(), t.getZ());
        } else if (color.compare(t.getName()) == 0){
            glColor3f(t.getX() / 255.f, t.getY() / 255.f, t.getZ() / 255.f);
        }
    }
    for (int z = 0; z < groups.getNrPrimitives(); z++) {
        Primitive p = groups.getPrimitives(z);
        if (groups.getNameFile().compare("asteroids.3d") == 0) {
            ...
        } else {
            glBegin(GL_TRIANGLES);
            for (int c = 0; c < p.getNrVertices(); c++) {
                Point point = p.getPoint(c);
                glVertex3f(point.getX(), point.getY(), point.getZ());
            }
            glEnd();
        }
    }
    for (int z = 0; z < groups.getNrGroups(); z++) {
        glPushMatrix();
        drawPrimitives(groups.getGroup(z));
        glPopMatrix();
    }
}
```

Capítulo 3

Ficheiro XML

Nesta segunda fase do projeto, desenvolveu-se um novo ficheiro XML, desta vez com o formato proposto no enunciado, a partir do qual, e com o auxílio das primitivas e transformações geométricas, foi possível representar o Sistema Solar.

No ficheiro XML, definimos um *scene* como uma árvore onde cada nó contém um conjunto de transformações geométricas, nomeadamente *translate*, *rotate* e *scale*, e, opcionalmente, um conjunto de modelos. Além disto, cada nó pode também ter nós filhos, herdando estes as transformações geométricas dos pais.

Em relação às transformações geométricas, tal como supramencionado, foram apenas aplicadas três transformações: a *translate*, a *rotate* e a *scale*. Para além destas, também foi aplicado uma transformação *color* que permite alterar a cor dos modelos.

Conforme referido anteriormente, o formato utilizado no ficheiro XML foi adaptado para o formato proposto no enunciado. Deste modo, temos um nodo que corresponde a um grupo (*group*) e um nodo que corresponde à camera (*camera*). Ambos os nodos são definidos por um conjunto de elementos: dentro de um nodo *group*, podemos encontrar um nodo *transform* e um nodo *models*, que são constituídos pelas transformações geométricas e o ficheiro .3d, respetivamente. Adicionalmente, podemos ter um ou mais nodos *group* dentro deste, designados por subgrupos. No que diz respeito ao conteúdo do nodo *camera*, podemos encontrar valores parâmetros para as funções *gluLookAt()* e *gluPerspective()*.

```
<camera>
    <position x="5.0" y="5.0" z="5.0"/>
    <lookAt x="0.0" y="0.0" z="0.0" />
    <up x="0.0" y="1.0" z="0.0"/>
    <projection fov="45" near="1" far="1000"/>
</camera>

<!-- Terra -->
<group>
    <transform>
        <rotate angle="-35" x="0" y="1" z="0"/>
        <translate x="1.75" y="0" z="0"/>
        <scale x="0.10" y="0.10" z="0.10"/>
        <color x="0" y="0" z="205"/>
```

```
</transform>
<models >
    <model file="sphere.3d"/>
</models >

<!-- LUA -->
<group>
    <transform>
        <translate x="1.3" y="1" z="1"/>
        <scale x="0.3" y="0.3" z="0.3"/>
        <color x="150" y="150" z="150"/>
    </transform>
    <models >
        <model file="sphere.3d"/>
    </models >
</group>
</group>
```

Capítulo 4

Extras

A presente seção do relatório pretende dar a conhecer eventuais atualizações efetuadas ao trabalho desenvolvido até ao momento. Algumas atualizações surgem como uma necessidade associada à expansão natural do trabalho. Por outro lado, outras alterações efetuadas surgem na perspetiva de proporcionar um maior leque de ferramentas que possam estar à disposição aquando da modelação do Sistema Solar.

De modo a obter mais ferramentas de desenho disponíveis para serem usadas na cena do Sistema Solar, foram incorporadas primitivas para a determinação dos pontos de um elipsoide e de uma semiesfera.

4.1 Elipsóide - Extra

A função *generateEllipsoid* permite a construção de um sólido elipsóide. A função recebe como parâmetros o raio e altura do elipsóide, o número total de *slices* e de *stacks*, assim como o nome do ficheiro .3d onde serão armazenados os pontos calculados. A Figura 4.1 apresenta a conceção adotada de para os parâmetros de altura e de raio do elipsóide.

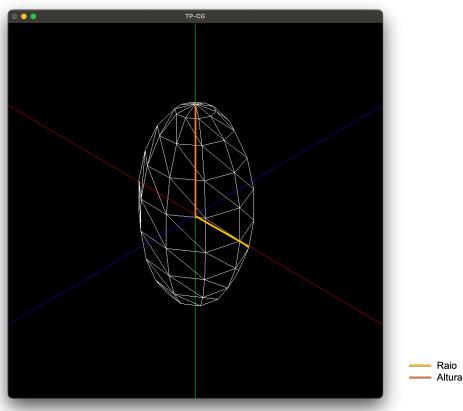


Figura 4.1: Parâmetros do elipsóide.

$$\begin{aligned}x(\theta, \phi) &= a * \sin(\theta) * \cos(\phi) \\y(\theta, \phi) &= c * \cos(\theta) \\z(\theta, \phi) &= a * \sin(\theta) * \sin(\phi)\end{aligned}$$

Figura 4.2: Equações paramétricas.

A noção de *slices* e de *stacks* segue os mesmos princípios apresentados para a esfera, previamente descrita na fase de entrega anterior. Assim sendo, tal como para o caso da esfera, a determinação dos pontos de um elipsóide foi efetuada com base em equações paramétricas, apresentadas na Figura 4.2. Observando as equações, é possível verificar a enorme similaridade das equações paramétricas apresentadas com as equações usadas para a determinação dos pontos da esfera. De facto, a esfera representa um caso particular em que o elipsoide possui os parâmetros altura e raio iguais¹.

As Figuras 4.3 e 4.4 resultam da representação gráfica após a execução dos comandos:

```
./GENERATE ellipsoid 2 1 10 ellipsoid.3d
```

```
./GENERATE ellipsoid 1 2 10 ellipsoid.3d
```

pode ser observado, a forma do elipsóide varia conforme a relação existente entre os parâmetros raio e altura: quando o raio é maior do que a altura, Figura 4.3, o elipsóide assume uma forma mais achatada. No caso de se verificar que o valor da altura é superior ao valor do raio, Figura 4.4, o elipsóide possui uma forma ligeiramente esticada.

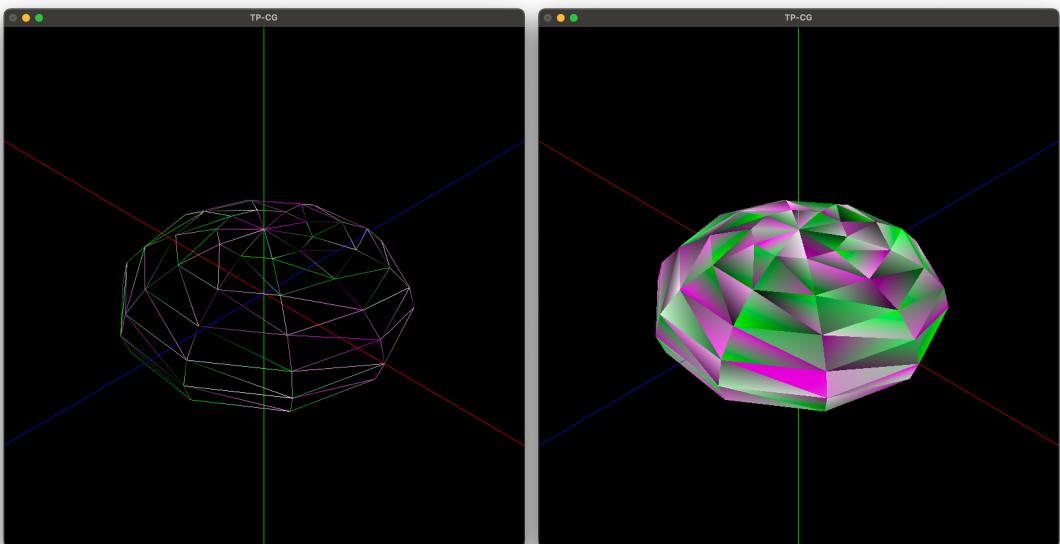


Figura 4.3: ellipsoid 2 1 10 ellipsoid.3d

¹Deve notar-se que, tipicamente, as equações do elipsóide distinguem três parâmetros para cada uma das equações apresentadas. Para efeitos do presente trabalho, esta conceção foi simplificada, obrigando a que o parâmetro de variação para o eixo dos xx e dos zz tivesse o mesmo valor.

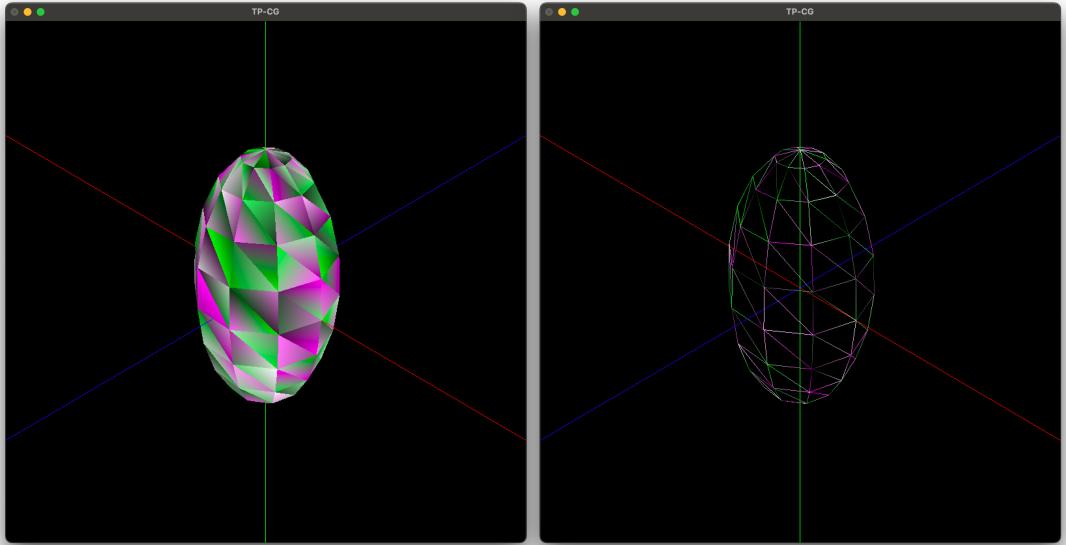


Figura 4.4: ellipsoid 1 2 10 ellipsoid.3d

4.2 Semiesfera - Extra

A função *generateHalfSphere* permite a construção de uma semiesfera. A função recebe como parâmetros o raio da semiesfera, o número total de *slices*, o número total de *stacks* e, ainda, o nome do ficheiro .3d onde serão armazenados os pontos calculados. A construção desta primitiva, tal como para o caso anterior, deriva do código desenvolvido anteriormente para a geração de esferas, tendo sido adaptado em dois níveis:

- Restrição do ângulo de desenho dos pontos;
- Desenho da base da semiesfera.

A redução do ângulo de rotação, aquando do desenho dos pontos, para π permitiu determinar os pontos de interesse para o desenho da superfície da semiesfera. Uma vez determinados, resta determinar os pontos necessários para representar a base da semiesfera. O cálculo destes pontos centrou-se na determinação dos pontos do círculo, em duas dimensões: a base da esfera foi dividida em diversas seções, onde, em cada seriam calculadas as coordenadas dos três pontos necessários para determinar o triângulo da secção referente. A Figura 4.5 apresenta uma seção, aleatória, de um círculo com raio 2, centrado na origem.

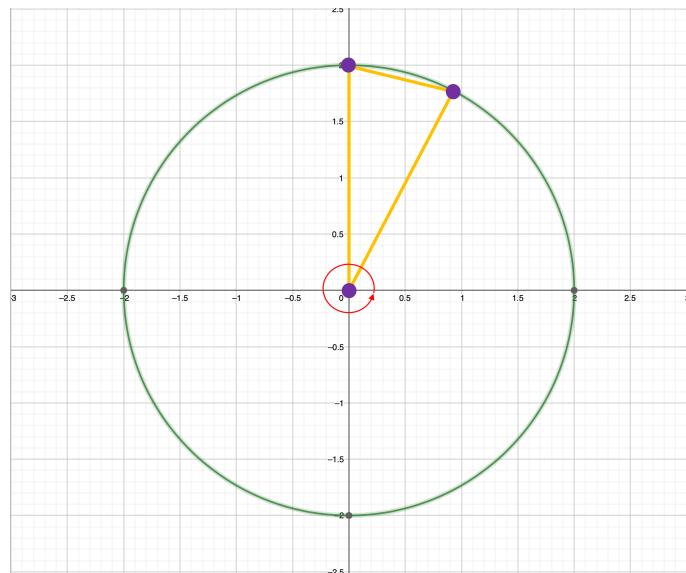


Figura 4.5: Estratégia de desenho da base da semiesfera

Através de um processo iterativo, são percorridas as diversas seções do círculo, tendo sido fixado o número total de 30 seções, e são determinados os pontos com base no valor dos senos e cossenos para o ângulo que corresponde à seção atual – deve notar-se que a “transcrição” deste raciocínio para o domínio tridimensional passou por fixar o valor da coordenada em z como sendo 0.

Após o cálculo dos pontos necessários, através da execução do comando `./GENERATE hsphere 1 10 10 hsphere.3d`, foi possível desenhar a semiesfera apresentada na Figura 4.6.

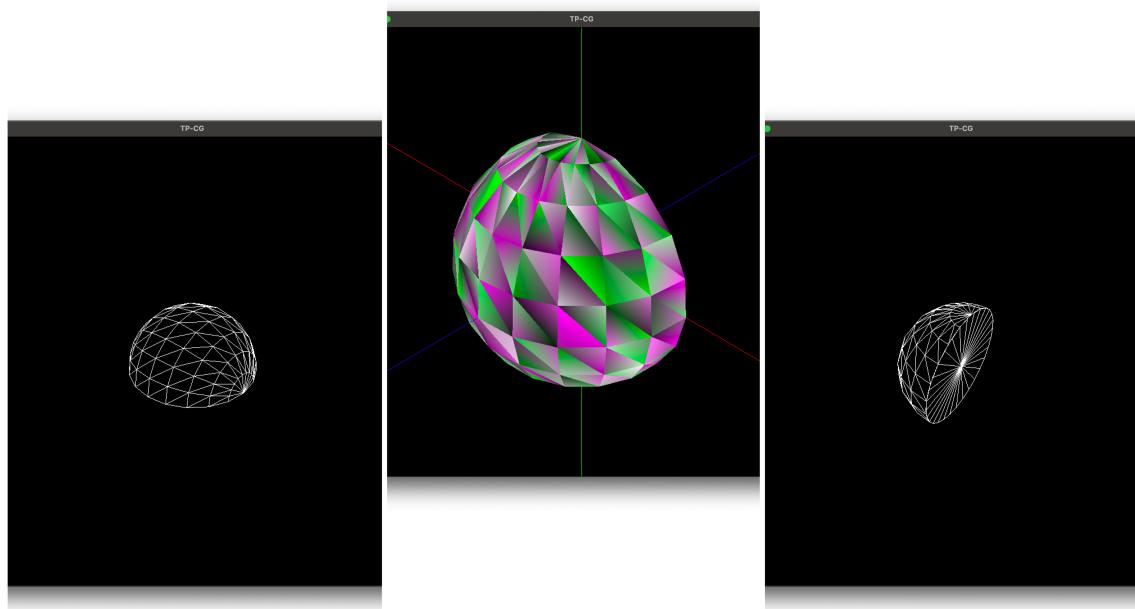


Figura 4.6: `hsphere 1 10 10 hsphere.3d`

4.3 Câmera

Com vista a tirar melhor partido da construção do modelo do sistema solar, foram integradas algumas funcionalidades que permitem o manuseamento da câmera. Estas funcionalidades representam, de um modo simplista, a reação a eventos do teclado, que provocam alterações na visualização da cena. Como tal, foi criada a função *rodar*, que permite efetuar operações simples como a rotação do ângulo de visualização – quer seja uma rotação horizontal, ou vertical – e ainda a possibilidade de efetuar *zoom in* ou *zoom out*. A função criada, com as respetivas teclas, encontra-se abaixo apresentada.

```
void rodar(int key_code, int x, int y) {
    float shift = M_PI / 2;
    switch (key_code) {
        case GLUT_KEY_LEFT:
            angle -= shift;
            break;
        case GLUT_KEY_RIGHT:
            angle += shift;
            break;
        case GLUT_KEY_UP:
            angle2 += shift;
            break;
        case GLUT_KEY_DOWN:
            angle2 -= shift;
            break;
        case GLUT_KEY_F1:
            zoomFactor = std::max(zoomFactor - 0.25f, min_zoom);
            break;
        case GLUT_KEY_F2:
            zoomFactor = std::min(zoomFactor + 0.25f, max_zoom);
            break;
    }
    glutPostRedisplay();
}
```

Para que estas alterações fossem visíveis, algumas alterações foram efetuadas à estrutura inicial do código e, em particular à função *renderScene*. Foram criadas variáveis globais, como *zoomFactor*, *min_zoom*, *angle*, que ajudam a determinar os parâmetros de visualização de uma cena – como pode ser observado no excerto de código apresentado da função *renderScene*. Estes valores podem sofrer alterações com as reações despoletadas pelos eventos acima mostrados: por exemplo, pressionar a tecla *F1* provocará um *zoom in* – ver Figura 4.7.

```
void renderScene(void) {
    (...)

    gluLookAt(eyeX * zoomFactor, eyeY * zoomFactor, eyeZ * zoomFactor,
              centerX, centerY, centerZ,
              upX, upY, upZ);

    glTranslatef(x, 0.0, z);
    glRotatef(angle, 0.0, 1.0, 0.0);
```

```

    glRotatef(angle2, 1.0, 0.0, 0.0);

    (...)

}

```

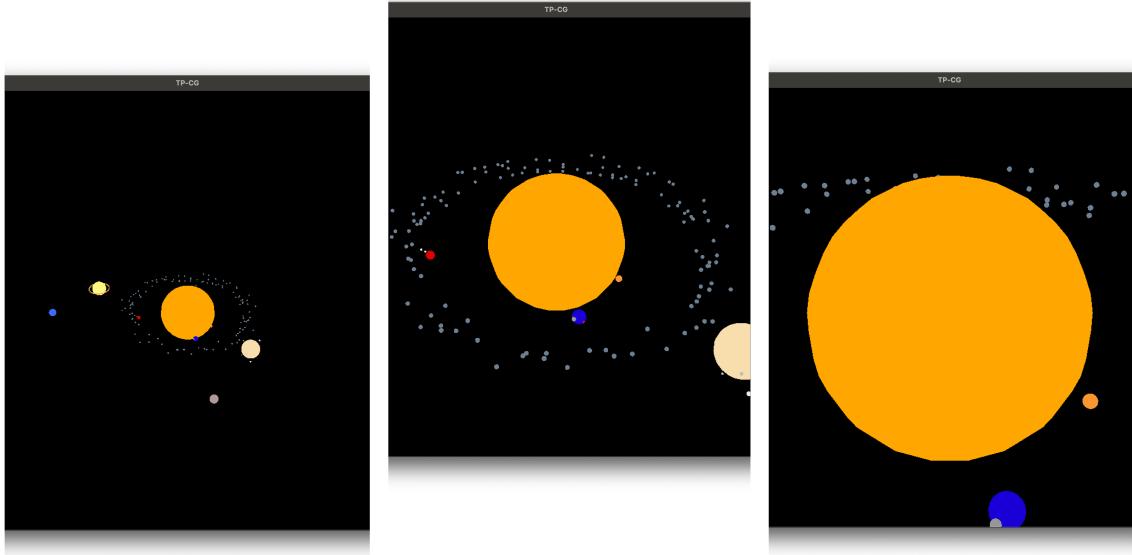


Figura 4.7: *Zoom in* do Sistema Solar.

4.4 Cintura de asteróides

A cintura de asteróides corresponde a uma região circular em torno do Sol, formada por diversos corpos irregulares. Para efeitos de representação, e tal como foi previamente referido, os asteróides são representados como esferas, sendo o seu desenho efetuado com recurso à função *drawPrimitives*.

O desenho deste conjunto de corpos celestes exigiu um pouco de criatividade no recurso à sua representação, uma vez que foi procurada uma tentativa de representação que segui-se a distribuição irregular que estes apresentam. Denote-se que esta estrutura difere, por exemplo, dos anéis de Saturno – que são representados através de um toro. Por seu lado, os asteróides seguem uma distribuição irregular e aparentemente aleatória: a uma dada distância do Sol, os asteróides orbitam num limiar próximo do “centro” da sua cintura. A necessidade de refletir esta distribuição aleatória, mas fixa num certo intervalo de distância, foi conseguida através da determinação aleatória de rotações e translações para várias esferas: são usados os pontos padrão de uma esfera para gerar um vasto número de esferas que serão dispostas de forma semi aleatória em torno do centro da cintura de asteróides – como pode ser observado no excerto de código abaixo apresentado.

```

void drawPrimitives(Group groups) {
    (...)

    if (groups.getNameFile().compare("asteroids.3d") == 0) {
        int arrx[180];
        int arry[180];

```

```

    srand(1);
    for (int i = 0; i < 100; i++) {
        arrx[i] = rand() % 18 + (-5);
    }
    for (int i = 0; i < 100; i++) {
        arry[i] = rand() % 18 + (-5);
    }
    for (int i = 0; i < 180; i++) {
        glPushMatrix();
        glRotated(6 * i, 0, 1, 0);
        glTranslated(80 + arrx[i], arry[i], 0);
        glBegin(GL_TRIANGLES);
        for (int c = 0; c < p.getNrVertices(); c++) {
            Point point = p.getPoint(c);
            glVertex3f(point.getX(), point.getY(), point.getZ());
        }
        glEnd();
        glPopMatrix();
    }
}
(...)
```

Capítulo 5

Sistema Solar

Como pedido para esta fase do projeto, foi desenvolvido um modelo do Sistema Solar a partir de um ficheiro XML. Para a criação do mesmo, foi tido em consideração as cores, as dimensões de cada planeta (e respetivas luas) e as distâncias entre planetas de maneira a tornar a sua representação o mais realista possível. De facto, este objetivo foi tão ambicionado que até foram desenhados os anéis de Saturno, com recurso à primitiva torus e a cintura de asteroides entre Marte e Júpiter, como previamente mencionado. Os planetas, as luas e o Sol foram construídos recorrendo à primitiva da esfera, tal como para a construção dos asteróides.

Relativamente aos valores aplicados para as transformações geométricas, nomeadamente para o *translate*, *rotate*, *scale* e *color*, foi tido em conta o seguinte: as distâncias entre os planetas para o *translate*; o posicionamento dos planetas na sua respetiva órbita em torno do Sol (tal como na Figura 5.1) para o *rotate*; as dimensões de cada planeta e luas para o *scale*; e, finalmente as cores (segundo o código RGB, tendo por base fontes na *internet*) para o *color*.

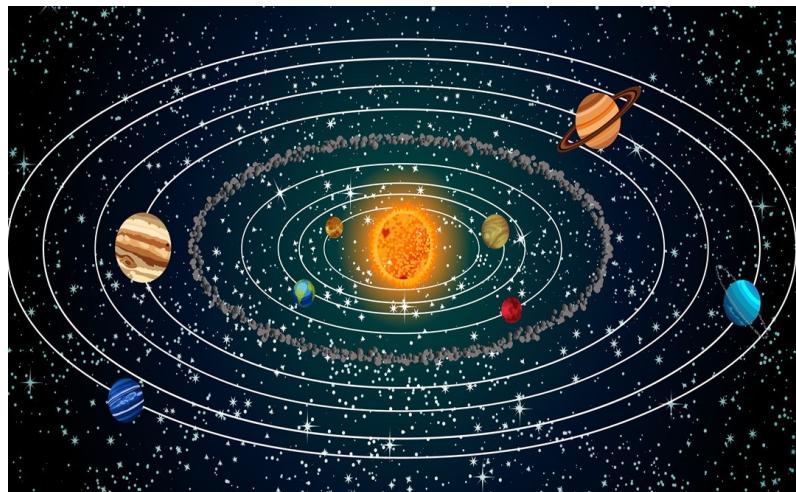


Figura 5.1: Sistema Solar

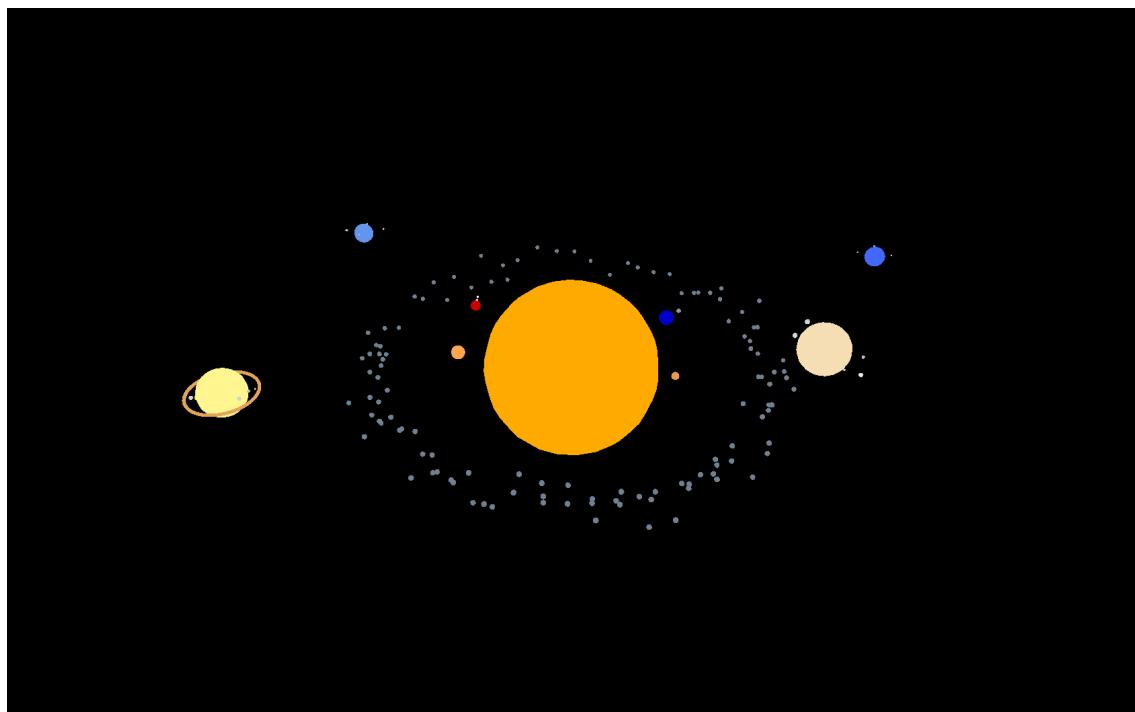


Figura 5.2: Sistema Solar gerado através do ficheiro *XML*.

Capítulo 6

Conclusão

A elaboração da presente fase do trabalho permitiu consolidar conhecimentos no domínio da linguagem *C++*. Adicionalmente, permitiu sedimentar os conceitos lecionados nas aulas relativos ao conhecimento em ferramentas de Computação Gráfica; particularmente, foram usadas transformações para operar sobre os diversos modelos criados.

O trabalho em torno do ficheiro *XML* revelou-se desafiante, em parte pelo desejo de uma representação fiel do Sistema Solar, mas também pela necessidade de desenvolvimento de um interpretador capaz de reconhecer e operar corretamente sobre o ficheiro *XML* que incorpora a representação do Sistema Solar. No decurso desta fase de desenvolvimento, algumas das dificuldades sentidas centraram-se precisamente na criação deste *parser* de *XML*.

Posto isto, considera-se que os objetivos estipulados foram cumpridos com sucesso, tendo o grupo desenvolvido funcionalidades adicionais que poderão ser proveitosa em fases futuras – como, por exemplo, seria o caso de incluir as novas primitivas de sólidos para o desenho de novas cenas.