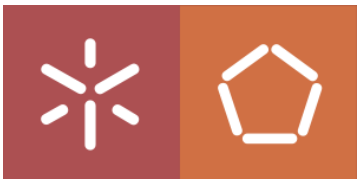

Universidade do Minho

Departamento de Informática

Computação Gráfica
Fase II – Transformações Geométricas

João Neves (a81366) Luís Manuel Pereira (a77667)
Rui Fernandes (a89138) Tiago Ribeiro (a76420)

Abril 2021

Resumo

O presente relatório descreve o trabalho prático realizado no âmbito da disciplina de *Computação Gráfica*, ao longo do segundo semestre do terceiro ano do Mestrado Integrado em Engenharia Informática da Universidade do Minho.

Assim, o principal objetivo deste trabalho é a implementação de transformações geométricas, aplicadas a um modelo estático do Sistema Solar que irá incluir o Sol e os diferentes planetas definidos numa hierarquia.

Neste documento descrevemos sucintamente a aplicação desenvolvida e discutimos as decisões tomadas durante a realização do trabalho prático.

Conteúdo

Lista de Figuras

1 Introdução

No seguimento da primeira fase do trabalho prático, esta fase tem como objetivo efetuar alterações no trabalho já desenvolvido de forma a acrescentar transformações geométricas, tais como rotações, translações e escalas.

Adicionou-se, também, uma nova primitiva gráfica ao *generator*, o *torus*, que permite a criação de anéis para alguns dos planetas.

Além disso, para permitir a correta renderização dos novos modelos, são necessárias alterações no *engine*, de forma a que este seja capaz de processar o novo formato dos ficheiros XML, e na estrutura de dados necessária para armazenar em memória as informações necessárias para a renderização das cenas.

Por fim, de forma a facilitar a compilação, foi criado um *script* `build.sh` que compila o código e gera os dois executáveis na diretoria `build` quando executado.

Ao longo deste relatório irão ser explicados a metodologia e raciocínio usados para a realização desta fase.

2 *Generator*

O *generator* é responsável pela escrita de ficheiros que contêm as coordenadas dos pontos necessários à triangulação de certas primitivas geométricas requisitadas. Para a elaboração de um modelo do Sistema Solar tornou-se necessária a implementação de uma nova primitiva gráfica, o *torus*, de forma a poder representar elementos como o anel de Saturno.

2.1 *Torus*

Um *torus* pode ser definido como o lugar geométrico tridimensional formado pela rotação de uma superfície circular plana de raio r, em torno de uma circunferência de raio R. Desta forma, para a sua construção são necessários quatro parâmetros – `inner_radius` (r), `outer_radius` (R), assim como o número de `slices` e `stacks`.

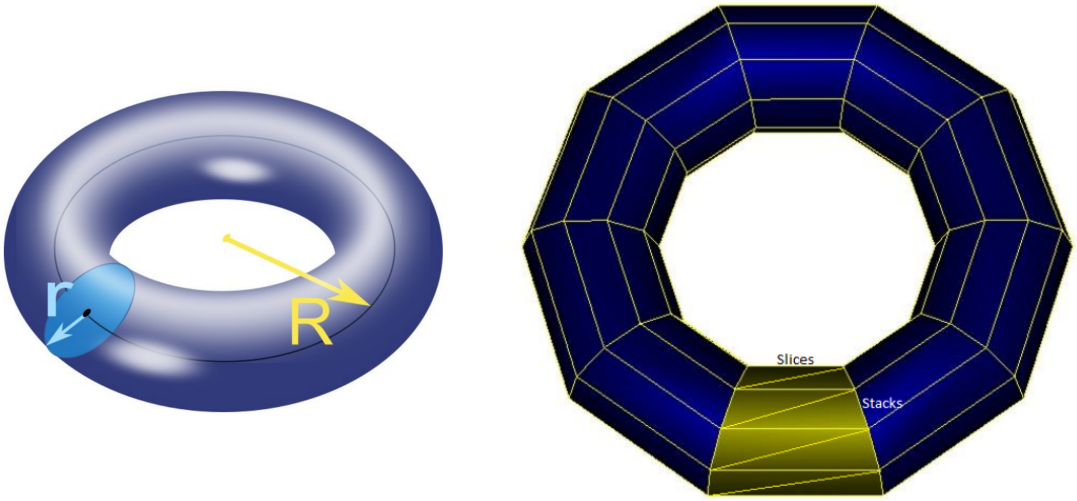

Figura 1: Geometria de um *torus*

Algoritmo

Começamos um raio interno, de forma a definir a espessura do *torus*, sendo que a lateral pode ser dividida em *stacks* e *slices*. Desta forma, começa-se por geral um anel, sendo este replicado até se completar o conjunto de *slices*. O raio exterior, por sua vez, define a abertura do *torus*, ou seja, a distância entre os anéis e a origem do referencial.

De maneira a percorrer as circunferências, são utilizados os parâmetros `slices` e `stacks` para dividir a execução em diferentes partes com amplitudes `theta_shift` e `phi_shift`.

```
float theta_shift = 2 * M_PI / slices;
float phi_shift = 2 * M_PI / stacks;
```

Assim, é possível percorrer tanto a circunferência interna ao adicionar `phi_shift`, como a externa ao adicionar `theta_shift`. Note-se que as variáveis `theta` e `phi` representam o ângulo interno e o externo, respetivamente, tal como ilustrado na Figura 2. Portanto, foi criado um ciclo para a iteração de cada circunferência. No final de cada iteração de construção de uma *stack*, incrementa-se `phi` de maneira a formar um anel. Mas também, no final de cada anel construído, incrementa-se o valor de `theta` de maneira a ser possível a passagem para a construção do anel imediatamente a seguir ao que foi gerado, até completar toda a circunferência externa e formar o *torus* pretendido.

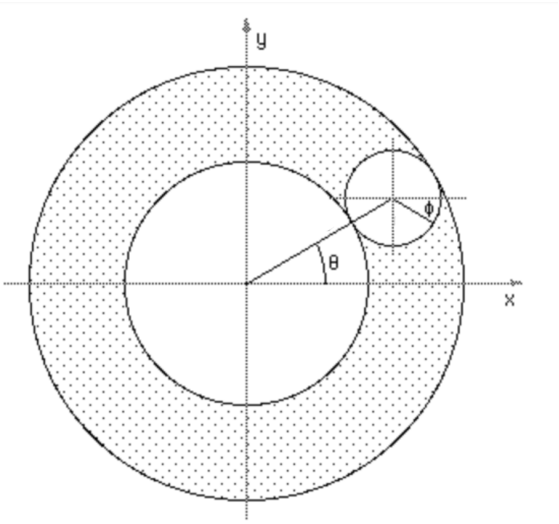

Figura 2: Esquema de um *Torus*

O resultado da renderização pode ser visto na figura seguinte:

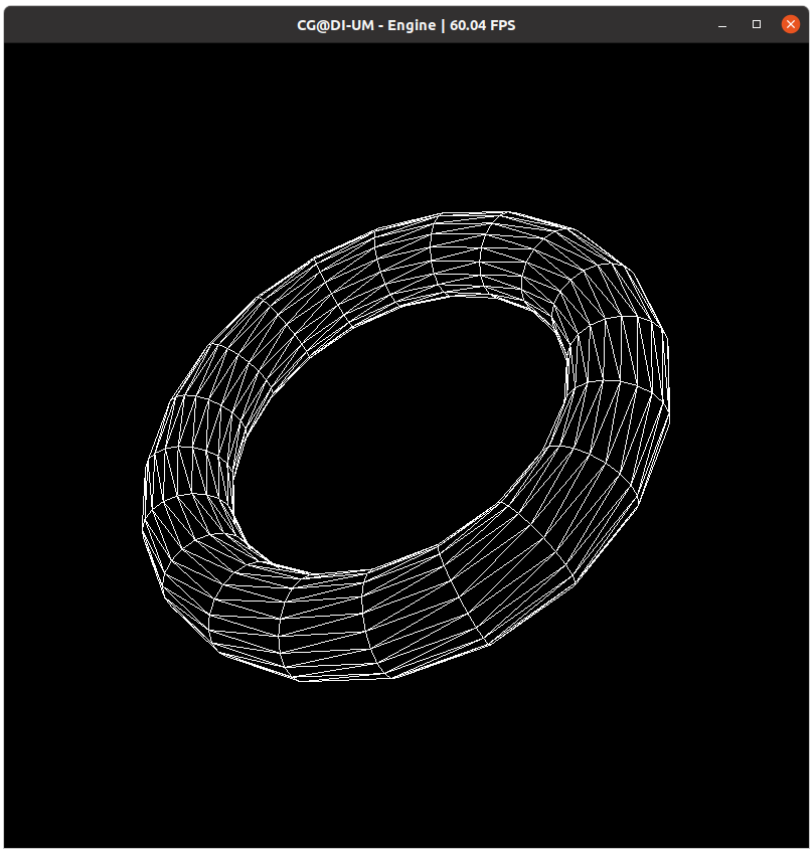

Figura 3: *Torus* com *inner radius* 1, *outer radius* 5, 20 *slices* e 20 *stacks*, renderizado no modo `GL_LINE`

3 *Engine*

Com o intuito de implementar as novas funcionalidades necessárias à realização desta fase do trabalho prático, optamos por alterar a estrutura do código já elaborado. Deste modo, criamos algumas classes, explicadas de seguida.

3.1 Classe *Transform*

De modo a armazenar todas as informações relativas a uma transformação geométrica, foi criada a seguinte classe `Transform`. As transformações geométricas podem ser rotações, translações e escalas. Neste sentido, é necessário guardar o vetor associado à transformação e, no caso de esta ser uma rotação, o ângulo.

```
class Transform {
private:
    std::string type;
    float angle, x, y, z;
    // (...)
};
```

3.2 Classe *Shape*

A classe `Shape` é responsável por armazenar as coordenadas dos os vértices necessários para representar uma determinada figura.

```
class Shape {
private:
    std::vector<Vertex *> vertices;
    // (...)
};
```

3.3 Classe *Group*

A classe `Group` é responsável por armazenar toda a informação de um determinado grupo, isto é, todas as informações relativas às primitivas geométricas e trasformações associadas, sendo, por isso, uma das classes mais importantes do projeto. Esta armazena todos os grupos filhos, assim como as transformações e as coordenadas pontos necessários para desenhar cada uma das formas geométricas.

```
class Group {
private:
    std::vector<Group *> groups;
    std::vector<Transform *> transforms;
    std::vector<Shape *> shapes;
    // (...)
};
```

3.4 Câmera

Inicialmente, a câmera encontra-se orientada para a origem do referencial, sendo esta capaz de se movimentar numa superfície esférica imaginária. Através da interação com o teclado, é possível alterar o valor do raio desta superfície esférica imaginária, assim como o valor dos ângulos α e β, que definem a posição da câmera. Note-se que é também possível recolocar a câmera na posição inicial utilizando a tecla `F3`.

De modo a implementar um menu que permite focar num determinado planeta, é necessário garantir que o movimento da câmera acompanha a posição para a qual esta está a olhar. Como tal, definimos as variáveis `look_x`, `look_y` e `look_z`, que indicam a posição para a qual a câmera está virada. Assim, fazendo uso de coordenadas esféricas, é possível definir a posição da câmera como:

```
cam_x = look_x + radius * cos(beta) * sin(alpha);
cam_y = look_y + radius * sin(beta);
cam_z = look_z + radius * cos(beta) * cos(alpha);
```

```
class Camera {
private:
    float alpha, beta, radius;
    float cam_x, cam_y, cam_z;
    float look_x, look_y, look_z;
    // (...)
};
```

4 *Parser* e Processamento de ficheiros XML

Nesta segunda fase do trabalho prático, existiu a necessidade de modificar a forma como é efetuado o *parsing* dos ficheiros XML uma vez que agora também é necessário aplicar transformações geométricas às figuras obtidas através da leitura dos ficheiros `.3d`. As informações relativas a estas transformações geométricas encontram-se registadas nos ficheiros XML, daí ser necessário alterar o *parser*. À semelhança da primeira fase, para a leitura dos ficheiros XML, foi utilizada a biblioteca `tinyxml2`, que disponibiliza um vasto conjunto de funções para o tratamento deste tipo de ficheiros.

Um ficheiro XML é composto pelos seguintes elementos:

- *Scene*;
- *Group*;
- *Models*;
- *Rotate*;
- *Scale*;
- *Translate*;
- *Colour*.

***Group*:** Determinados elementos e atributos podem ser agrupados através do elemento *group*.

***Models*:** A *tag models* aparece associada aos ficheiros `.3d`, que contêm as coordenadas dos vértices necessários à triangulação de certas primitivas geométricas, necessárias para renderizar uma cena.

***Rotate, Scale* e *Translate*:** As *tags translate*, *rotate* e *scale* estão associadas à translação, rotação e escala de um determinado objeto, respetivamente.

***Colour*:** De maneira a reconhecer os diferentes constituintes do modelo do Sistema Solar, foi também utilizado um atributo que especifica a cor dos mesmos.

A seguir apresenta-se um exemplo da forma como estas *tags* são utilizadas.

```
<group>
    <!-- Mercurio -->
    <translate X="-10" Z="20" />
    <scale X="0.24" Y="0.24" Z="0.24" />
    <colour R="0.2" G="0.2" B="0.2" />
    <models>
        <model file="sphere.3d" />
    </models>
</group>
```

Parsing

Inicialmente, o ficheiro XML é carregado para memória, utilizando a função `load_XML_file`, sendo esta responsável por invocar a função `parse_group` em caso de sucesso. Para tal, efetua-se um ciclo que percorre todos os nodos, processando individualmente cada um deles. Caso este corresponda a uma transformação geométrica, invoca-se a função responsável por recolher a informação necessária e adicioná-la a uma estrutura de dados adequada. Note-se que as informações relativas à cor com a qual as figuras deverão ser preenchidas vão também ser guardadas recorrendo à classe `Transform`, e, uma vez que estas estão também guardadas no ficheiro XML, irá ser necessário extraí-las juntamente com as informações das transformações geométricas.

Por outro lado, o nodo poderá conter a informação do ficheiro `.3d` que deverá ser lido. Nesse caso, é invocada a função `parse_models`, que, por sua vez, invoca a função `read_file` com o intuito de ler as coordenadas dos pontos de cada ficheiro, armazenando-os num vetor de formas geométricas, que será armazenado na estrutura de dados principal.

5 Renderização do Modelo

O *engine* está dividido essencialmente em duas partes: a leitura do ficheiro XML, e a interpretação e renderização do conteúdo do mesmo.

No que diz respeito à interpretação e renderização do conteúdo do ficheiro XML, na função `renderScene` irá ser chamada a função `drawScene`, tendo esta como único argumento o `Group` contendo toda a informação relativa ao modelo do Sistema Solar. Antes de desenhar as primitivas, é necessário verificar não só as transformações existentes, de forma a realizar as translações, rotações ou escalas necessárias, mas também as aplicar a cor necessária à figura, tendo em conta os parâmetros fornecidos para ambos os casos. Deste modo, fazendo uso das funções do openGl (`glTranslatef`, `glRotatef`, `glScalef`, etc), a função `drawScene` gera as respetivas figuras após aplicadas as devidas transformações geométricas.

Importa salientar que para o desenho em si, uma vez que serão efetuadas transformações geométricas, existe uma alteração na matriz de transformação e por isso deve ser guardado o estado inicial desta, e depois de todas as transformações serem aplicadas, este estado deve ser reposto. Para isso são usadas as funções `glPushMatrix` e `glPopMatrix` antes de aplicar a transformação e depois de esta ser aplicada, respetivamente.

Depois de realizadas todas as transformações, percorre-se o vetor que contém as coordenadas de todos os pontos necessários para desenhar as figuras e procede-se ao desenho dos triângulos que as compõem usando a função `glBegin(GL_TRIANGLES)`.

Por fim, os grupos filhos são processados através de uma chamada recursiva à função `drawScene`.

6 Sistema Solar

Para a segunda fase do trabalho prático, foi proposta a elaboração de um modelo estático do sistema solar, que contém o Sol, assim como os diferentes planetas e respetivas luas. De modo a reaproveitar o trabalho realizado na primeira fase, foi utilizado o ficheiro com o modelo da esfera para a representação de todos os elementos do Sistema Solar. Assim, a esfera utilizada tem raio 5, 20 *slices* e 20 *stacks*, e o *torus* tem *inner radius* 1, *outer radius* 5, 20 *slices* e 20 *stacks*.

Para cada um dos corpos celestes foi escolhida uma cor e foi definida uma rotação e translação de modo simular o movimento do planetas em torno do Sol. Foi também atribuída uma escala de forma a simular o tamanho dos planetas.

Assim, tendo em consideração aspetos como o raio dos diferentes planetas ao Sol e o seu raio, assim como a existência de anéis e satélites naturais, elaboramos um ficheiro XML que permite a construção do modelo do Sistema Solar.

No entanto, ao tentar estabelecer uma escala realista, houve alguma dificuldade uma vez que os últimos quatro planetas encontram-se muito distantes Sol, além de que os satélites de cada um dos planetas são pequenos. Posto isto, optamos por não fazer o Sistema Solar à escala, isto é, não respeitamos totalmente as dimensões dos diferentes corpos celestes nem as distâncias que os separam, uma vez que, caso o fizessemos, o resultado final seria um cenário muito disperso e de difícil observação.

Após construção do modelo do Sistema Solar, foram obtidos os resultados que se apresentam de seguida.

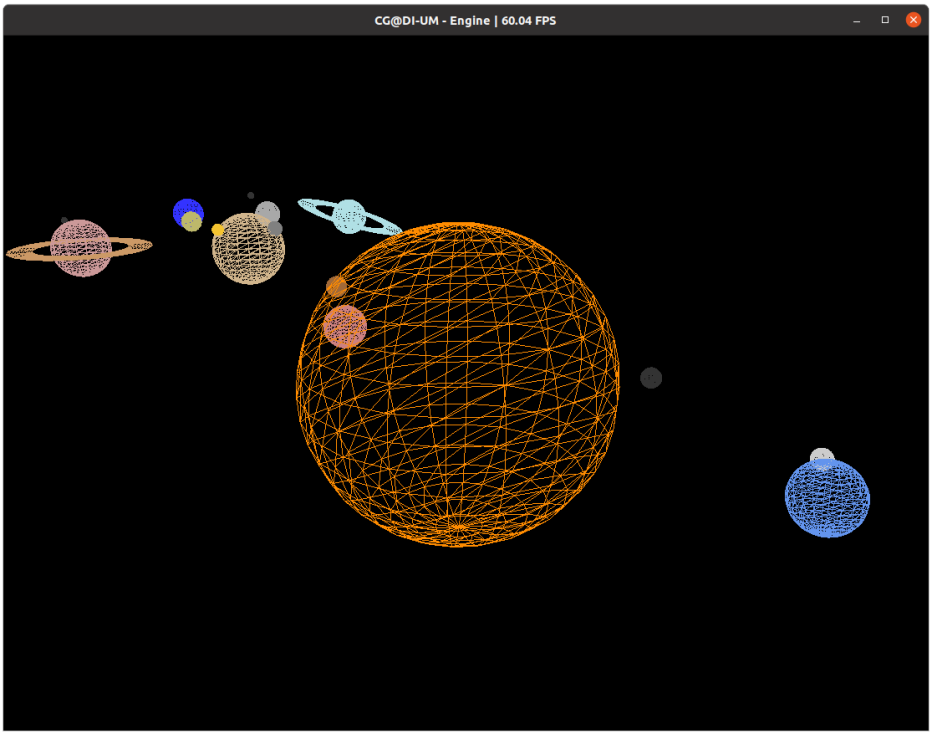

Figura 4: Modelo do Sistema Solar renderizado com linhas

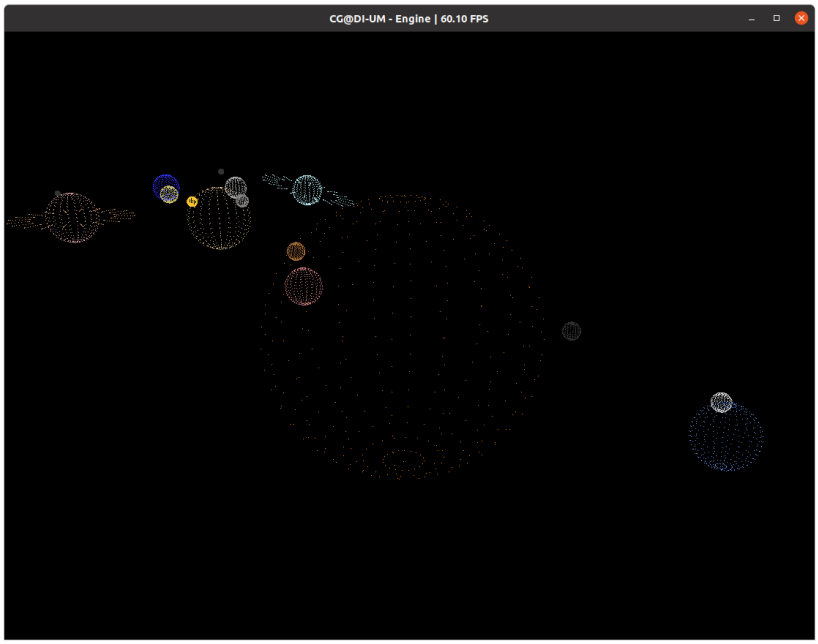

Figura 5: Modelo do Sistema Solar renderizado com pontos

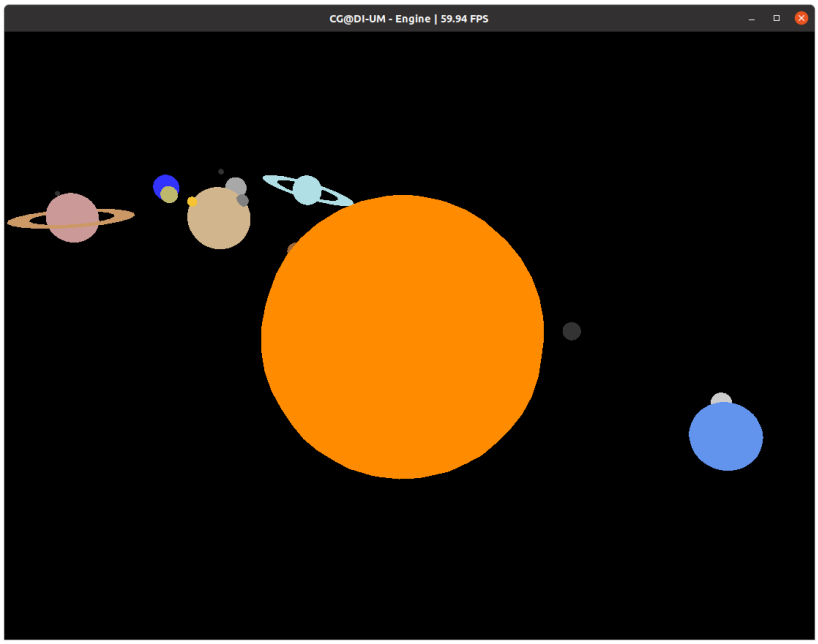

Figura 6: Modelo do Sistema Solar renderizado com as figuras preenchidas

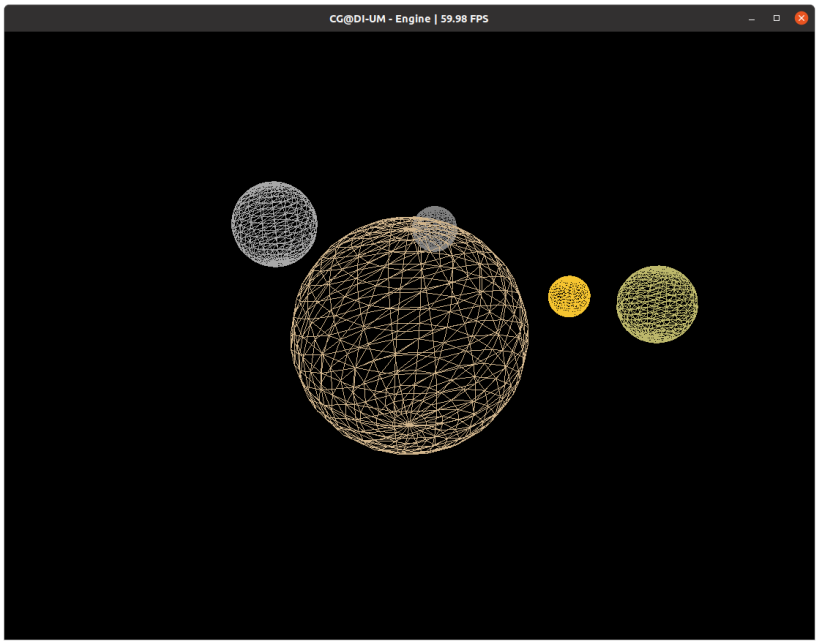

Figura 7: Júpiter e os satélites IO, Europa, Calisto e Ganímedes

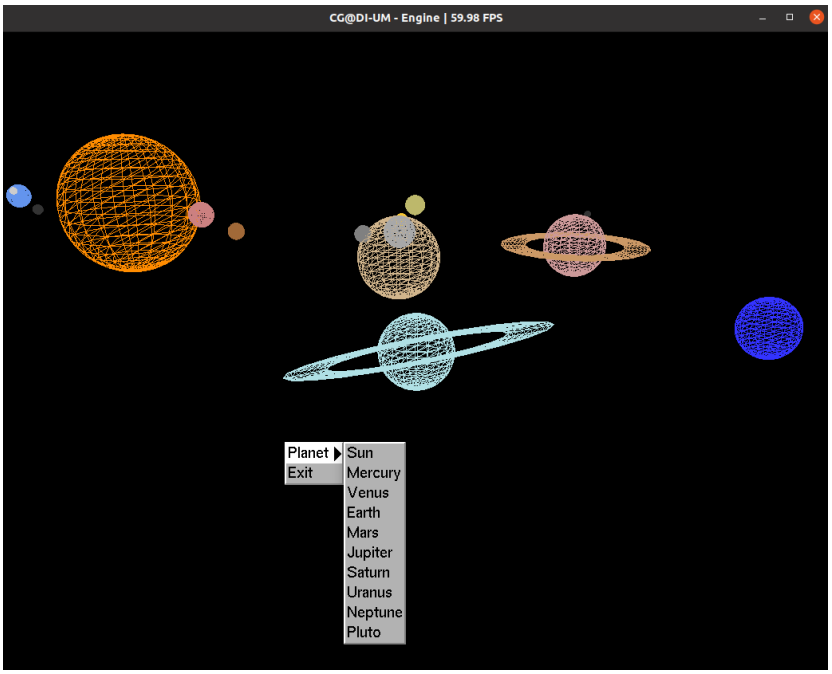

Figura 8: Menu dos Planetas

7 Conclusão

O desenvolvimento desta segunda fase do trabalho foi um pouco mais trabalhosa e demorada em relação à primeira fase. Isto deve-se ao sobretudo à alteração da estrutura dos ficheiros XML e consequentes alterações no *parsing* destas informações.

Nesta segunda fase do trabalho prático foi-nos possível aprofundar e por em prática conceitos relativos a transformações geométricas, o que nos permitiu ter uma melhor noção da importância da ordem e do funcionamento base das mesmas. No geral, consideramos que a maior parte dos resultados obtidos nesta fase correspondem aos esperado, uma vez que apresentamos um modelo do Sistema Solar, ainda bastante simples mas com o objetivo de nas fases que se seguem melhorar algumas questões. No entanto, ficaram por implementar alguns aspetos, como por exemplo a cintura de asteróides e de *Kuiper*.

Em conclusão, esperamos que nas restantes fases ultrapassemos estes obstáculos e consigamos melhorar cada vez mais este modelo, tornando-o mais realista e mais apelativo visualmente.