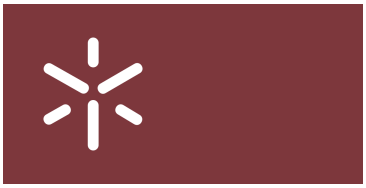

Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática
Mestrado Integrado em Engenharia Informática

Unidade Curricular de Computação Gráfica

Ano Letivo de 2024/2025

Transformações Geométricas - Fase 2

Afonso Pedreira
A104537

Fábio Magalhães
A104365

André Pinto
A104267

Março, 2025

CG

Índice

1. Introdução

Este relatório tem como objetivo explicar o trabalho desenvolvido na segunda fase desta UC, aplicando as técnicas apresentadas tanto nas aulas teóricas como nas aulas práticas. Nesta segunda fase, o trabalho consistiu no processamento e criação de cenas hierárquicas utilizando transformações geométricas.

O objetivo desta fase foi adaptar o **engine** de forma a conseguir processar este tipo de cenas, implementando o suporte para a definição e manipulação dessas estruturas hierárquicas. A cena demonstrativa exigida para esta fase consiste num modelo estático do sistema solar, incluindo o Sol, os planetas e as suas luas, todos definidos dentro de uma hierarquia coerente.

2. Estrutura do Projeto

O projeto está dividido em dois programas principais: ***generator*** e ***engine***, que trabalham em conjunto para gerar e renderizar figuras gráficas. A estrutura do projeto é organizada de forma a facilitar a modularidade e a reutilização de código.

A arquitetura do projeto é composta pelas seguintes pastas principais:

- **engine**: Esta pasta contém os componentes responsáveis pela renderização das cenas. Inclui elementos essenciais como:
  - **camera**: Define a perspetiva e a posição da visualização da cena.
  - **configuration**: Contém as configurações gerais do sistema, como a `window`, as definições da câmara e os modelos a serem desenhados.
  - **window**: Responsável pela gestão dos parâmetros da janela de visualização.
  - **draw**: Contém a lógica de desenho da cena na janela.
  - **parser**: Este módulo é responsável por fazer o `parsing` dos ficheiros de entrada (como `.xml`, `.obj` e `.3d`) e configurar os dados necessários para a renderização.
- **generator**: Esta pasta contém os módulos responsáveis pela geração das figuras geométricas. Cada figura é gerada a partir de parâmetros específicos e guarda num ficheiro `.3d`. Estes ficheiros, posteriormente, serão embutidos em ficheiros `.xml`, que a *engine* utiliza para renderizar as cenas.
- **common**: Esta pasta contém componentes compartilhados entre os dois programas principais. Inclui definições essenciais, como a definição de **Ponto** bem como funções auxiliares para tarefas comuns, como:
  - **salvar em ficheiro**: Funções para escrever os dados gerados em ficheiros de diferentes formatos.
  - **ler de ficheiro**: Funções para ler dados de ficheiros de entrada, como `.xml` e `.obj`.
  - **criar ficheiro 3D**: Funções que permitem a criação e manipulação de ficheiros no formato `.3d`.

A organização modular do projeto permite uma gestão eficiente de cada componente, garantindo a escalabilidade e uma boa base para as próximas fases.

2.1. *Engine*

Nesta segunda fase, foi necessário efetuar algumas melhorias e ajustes no `engine`, particularmente no `parsing` de ficheiros XML e na gestão dos modelos para a sua posterior renderização no ecrã. Estas alterações foram fundamentais para permitir a correta representação das transformações geométricas e garantir que a hierarquia da cena fosse respeitada.

2.1.1. *ModelGroup*

Para assegurar que toda a informação fosse devidamente organizada, foi criada a estrutura ***ModelGroup***. Esta estrutura permite armazenar não só as transformações aplicadas a um grupo de modelos, mas também manter a relação hierárquica entre subgrupos. Desta forma, assegura-se uma gestão mais eficiente dos modelos e a correta aplicação das transformações na cena. A respon-

sabilidade pelo armazenamento de vértices foi transferida da *Configuration* para o *ModelGroup*, ficando a *Configuration* apenas com acesso ao *ModelGroup* principal do ficheiro *XML*.

```
ModelGroup::ModelGroup(
    std::vector<std::string> models, std::vector<ModelGroup> subModelGroups,
    std::vector<Point> points,
    std::array<std::array<double, 4>, 4> transformationMatrix) {
  this->models = models;
  this->subModelGroups = subModelGroups;
  this->vertices = points;
  this->transformationMatrix = transformationMatrix;
}
```

Listagem 1: ModelGroup

2.1.2. Parsing de Ficheiros XML

Tal como mencionado na fase anterior, foi escolhida a biblioteca **RapidXML** devido à sua facilidade de utilização e à rapidez na aprendizagem dos seus conceitos. Para garantir que as transformações fossem aplicadas corretamente a cada modelo presente no ficheiro `XML`, foi necessário implementar uma nova estratégia de `parsing`, utilizando um método recursivo. Este método assegura que todas as transformações sejam aplicadas de forma hierárquica e estruturada. Em termos práticos, sempre que um novo grupo é detetado no ficheiro `XML`, ele é acrescentado ao grupo atual como um subgrupo, repetindo este processo até ao final do ficheiro.

```
ModelGroup parseModelGroup(rapidxml::xml_node<>* groupNode) {
  ModelGroup modelGroup;

  rapidxml::xml_node<>* transformNode = groupNode->first_node("transform");
  if (transformNode) {
    parseTransform(transformNode, group);
  }

  rapidxml::xml_node<>* modelsNode = groupNode->first_node("models");
  if (modelsNode) {
    parseModelGroup(modelsNode, group);
  }

  rapidxml::xml_node<>* subgroupsNode = groupNode->first_node("group");
  while (subgroupsNode) {
    ModelGroup subModelgroup = parseModelGroup(subgroupsNode);
    modelGroup.subModelGroups.push_back(subgroup);
    subgroupsNode = subgroupsNode->next_sibling("group");
  }

  return modelGroup;
}
```

Listagem 2: Método de parsing XML

3. Transformações (Escalas, Translações, Rotações)

3.1. Fórmulas Utilizadas

As fórmulas para calcular cada tipo de transformação são as seguintes:

- **Translação em (x, y, z)**:

$$\mathcal{T}(x, y, z) = \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

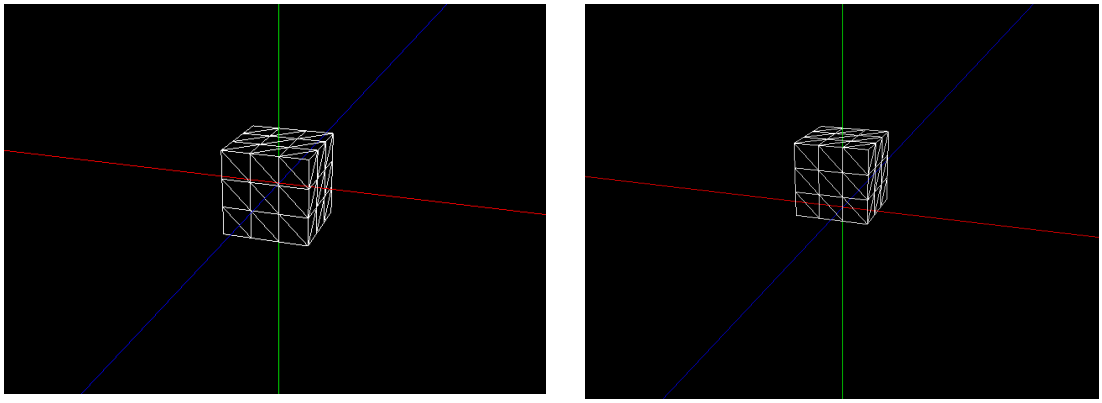

Figura 1: Antes e depois da aplicação de uma translação em Y=1

- **Escala em (x, y, z)**:

$$\mathcal{S}(x, y, z) = \begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

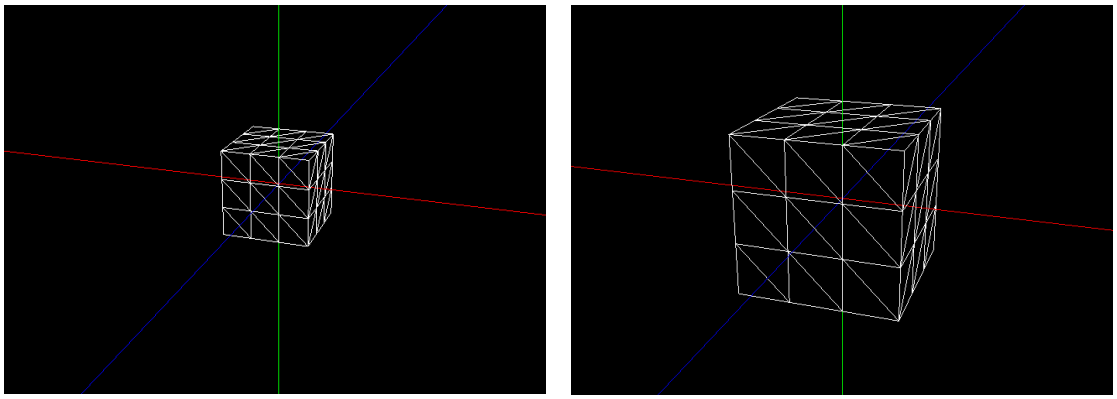

Figura 2: Antes e depois da aplicação de uma escala de 2 unidades em todos os eixos

- **Rotação em torno de um eixo arbitrário (x, y, z) por um ângulo α**:

$$\mathcal{R}(\alpha, x, y, z) = \begin{pmatrix} x^2 + (1 - x^2)\cos(\alpha) & xy(1 - \cos(\alpha)) - z\sin(\alpha) & xz(1 - \cos(\alpha)) + y\sin(\alpha) & 0 \\ xy(1 - \cos(\alpha)) + z\sin(\alpha) & y^2 + (1 - y^2)\cos(\alpha) & yz(1 - \cos(\alpha)) - x\sin(\alpha) & 0 \\ xz(1 - \cos(\alpha)) - y\sin(\alpha) & yz(1 - \cos(\alpha)) + x\sin(\alpha) & z^2 + (1 - z^2)\cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

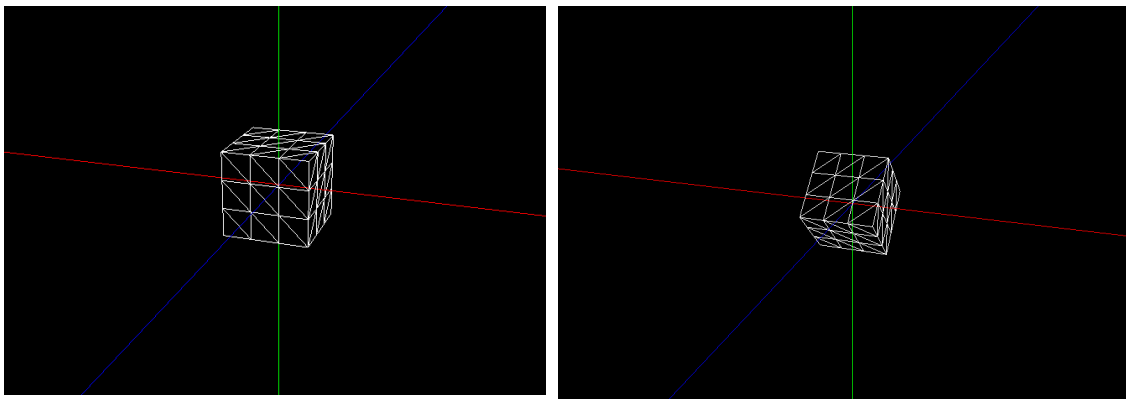

Figura 3: Antes e depois da aplicação de uma rotação de 45º em torno do eixo Y

3.2. Exemplo de transformações

Para que as transformações sejam aplicadas corretamente, cada `_ModelGroup_` deve possuir a sua própria matriz de transformações, como referido anteriormente. Desta forma, todas as transformações aplicadas ao grupo principal afetarão também os seus subgrupos, assegurando então que as alterações afetam tanto o grupo como os seus subgrupos de forma consistente, respeitando a hierarquia.

Assim, ao analisar o seguinte excerto *XML*, é possível perceber que:

```
<group>
<transform>
<translate x="0" y="1" z="0"/>
</transform>
<models>
<model file="box_2_3.3d"/>
<!--  generator box 2 3 box_2_3.3d  -->
</models>
<group>
<transform>
<translate x="0" y="1" z="0"/>
</transform>
<models>
<model file="cone_1_2_4_3.3d"/>
<!--  generator cone 1 2 4 3 cone_1_2_4_3.3d  -->
</models>
<group>
<transform>
<translate x="0" y="3" z="0"/>
</transform>
<models>
<model file="sphere_1_8_8.3d"/>
<!--  generator sphere 1 8 8 sphere_1_8_8.3d  -->
</models>
</group>
</group>
</group>
```

Listagem 3: Transformações ficheiro XML

O grupo principal inicia com uma translação de 1 unidade em y. Em seguida, um subgrupo sofre uma nova translação de 1 unidade em y, acumulando a transformação anterior. No nível seguinte, o próximo subgrupo recebe uma translação adicional de 3 unidades em y, continuando a acumulação das transformações hierárquicas.

Assim, utilizando as fórmulas apresentadas anteriormente, é trivial chegar ao resultado de aplicar as transformações a cada modelo:

Caixa

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Para a correta representação da caixa, é necessário aplicar a matriz resultante da multiplicação entre a matriz identidade e a matriz de translação em Y aos pontos da mesma.

Cone

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Para a correta representação do cone (que faz parte de um subgrupo da caixa), é necessário aplicar a matriz resultante da multiplicação entre a matriz de transformação do grupo pai e a matriz de translação em Y aos pontos do cone.

Esfera

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 5 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Para a correta representação da esfera (que faz parte do subgrupo do cone), é necessário aplicar a matriz resultante da multiplicação entre a matriz de transformação do grupo pai e a matriz de translação em Y por 3 unidades aos pontos da esfera.

Após a aplicação de todas as transformações descritas anteriormente, obtém-se o seguinte resultado:

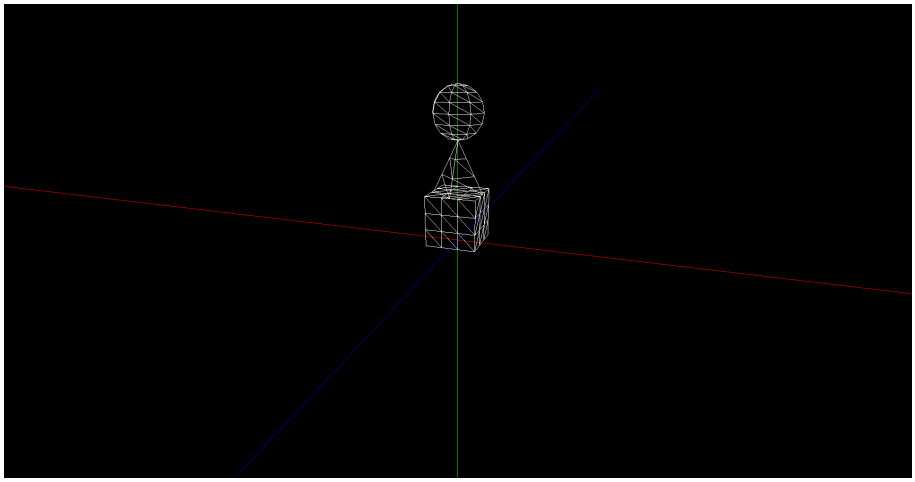

Figura 4: Resultado da aplicação das transformações

3.3. Renderização dos pontos de cada *ModelGroup* no ecrã

Tal como foi apresentado anteriormente no contexto do *parsing* do novo formato *XML*, para desenhar os pontos no ecrã é necessário utilizar também uma função recursiva. Esta função percorre todos os subgrupos do grupo principal, garantindo que a matriz de transformações seja corretamente propagada a cada subgrupo.

Para isso, recorremos à função `_glMultMatrix_()` do *OpenGL*, que multiplica a matriz de transformação atual pela matriz fornecida. Após a renderização de cada modelo dentro do grupo e a chamada recursiva para desenhar os subgrupos, utilizamos `_glPopMatrix_()` para reverter as alterações feitas, garantindo que as transformações dos subgrupos não afetem o restante desenho da cena.

```
// applying  the transformation to the current matrix
glMultMatrixf(transformationMatrix);

  glBegin(GL_TRIANGLES);
  glColor3f(1.0f, 1.0f, 1.0f);
  for (size_t idx = 0; idx < vertices.size(); idx += 3) {

    glVertex3f(vertices[idx].x, vertices[idx].y, vertices[idx].z);
    glVertex3f(vertices[idx + 1].x, vertices[idx + 1].y, vertices[idx + 1].z);
    glVertex3f(vertices[idx + 2].x, vertices[idx + 2].y, vertices[idx + 2].z);
  }
  glEnd();

  for (size_t idx = 0; idx < group.subModelGroups.size(); idx++) {
    drawModelGroups(group.subModelGroups[idx]);
  }

  // Restoring subgroups transformations
  glPopMatrix();
```

Listagem 4: Desenho do ModelGroup

3.4. Testes Realizados

Após as alterações necessárias no *engine*, especialmente na parte do *parsing* do novo formato, que agora inclui transformações e hierarquias, foi realizada uma série de testes disponibilizados pelos docentes para validar a correta interpretação e renderização dos modelos. Estes testes podem ser encontrados na pasta `scenes/testes`.

Em seguida, apresentam-se os resultados obtidos:

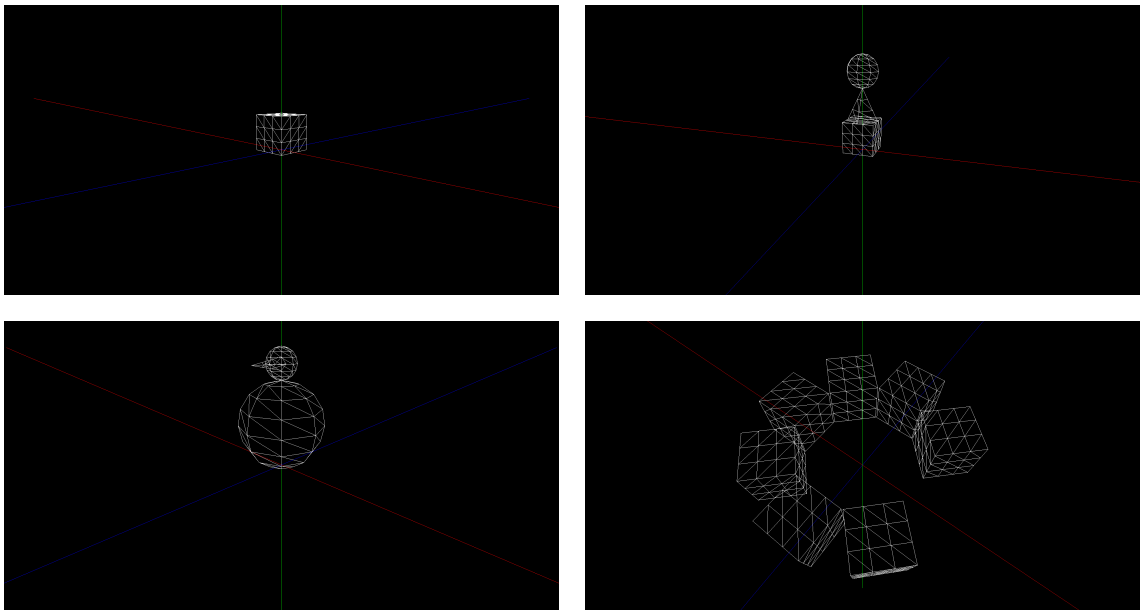

Figura 5: Resultados dos testes disponibilizados pelos docentes

4. Sistema Solar

Um dos requisitos para esta segunda fase foi a criação de uma cena do sistema solar, com o Sol, os planetas e algumas das suas luas, organizados numa hierarquia coerente. A cena exigida consiste num modelo estático, onde todos os elementos estão corretamente estruturados.

4.1. Construção do Sistema Solar

Para criar um modelo o mais realista possível, o grupo baseou-se em dados fornecidos pela *NASA* (*Eyes on the Solar System*) e pelo site *Devstronomy* . Estas fontes forneceram informações detalhadas, como diâmetros, ângulos de inclinação, distâncias ao Sol, períodos de rotação e translação, o ângulo atual da órbita de cada planeta e dados sobre todas as luas que orbitam os planetas do sistema solar.

Assim, tomando o Sol como referência central do Sistema Solar (representado por uma esfera de raio 1), torna-se trivial calcular tanto a posição de cada planeta na sua órbita como o seu tamanho relativo. Para isso, utilizou-se as seguintes fórmulas:

4.1.1. Tamanho

$$\text{planetSize} = \frac{\text{planetDiameter}}{\text{sunDiameter}}$$

Devido ao tamanho reduzido dos planetas em comparação com o Sol, aumentámos uma casa decimal em todos os valores, garantindo que ficassem visíveis na cena. Assim, os tamanhos foram arredondados para cima, mantendo uma escala proporcional, mas mais perceptível.

O mesmo método foi aplicado às luas de cada planeta, mas com uma fórmula diferente:

$$\text{sateliteSize} = \frac{\text{sateliteDiameter}}{\text{planetDiameter}}$$

4.1.2. Distâncias

Optámos por uniformizar as distâncias, atribuindo 4 unidades entre Mercúrio e o Sol. Para os outros planetas, ajustámos as distâncias reais proporcionalmente à nova escala, garantindo uma distribuição visível e coerente, ao mesmo tempo que mantivemos a escala o mais realista possível.

4.2. Anéis de Saturno

Para alcançar uma representação fidedigna do nosso sistema solar, implementámos os icónicos anéis de Saturno utilizando uma das primitivas geométricas desenvolvidas anteriormente - o **Torus** (ou *Donut*).

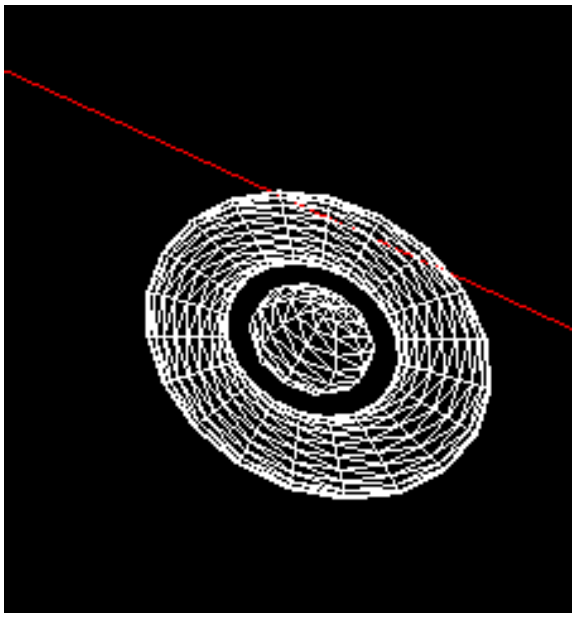

Figura 6: Saturno e os seus anéis

4.3. Cintura de Asteroides

Para além de representar todos os planetas do Sistema Solar e as respetivas luas, o grupo decidiu ir mais além e recriar a **cintura de asteróides**, que se encontra entre as órbitas de Marte e Júpiter. Para tal, foi desenvolvido um *script* em Python responsável por gerar aleatoriamente **1200 asteroides**, com tamanhos e ângulos de rotação variáveis, povoando o grupo ***Asteroids Belt*** de forma aleatória, com cada um dos asteroides posicionado de maneira única.

```
for i in range(1, num_asteroids + 1):
    angle = random.uniform(0, 360)
    distance = random.uniform(11, 12)
    scale = random.uniform(0.0001, 0.03)

    asteroid_data.append(f'    <group name="Asteroid{i}">\n')
    asteroid_data.append(f'        <models>\n')
    asteroid_data.append(f'            <model file="planet.3d"></model>\n')
    asteroid_data.append(f'        </models>\n')
    asteroid_data.append(f'        <transform>\n')
    asteroid_data.append(f'            <rotate angle="{angle}" x="0" y="1" z="0" />\n')
    asteroid_data.append(f'            <translate x="{distance}" y="0" z="0" />\n')
    asteroid_data.append(f'            <scale x="{scale}" y="{scale}" z="{scale}" />\n')
    asteroid_data.append(f'        </transform>\n')
    asteroid_data.append(f'    </group>\n')
```

Listagem 5: Script geração cintura de asteroides

4.4. Resultados Obtidos

Após uma minuciosa recolha e organização de todos os dados relativos aos planetas e respetivas luas, procedemos à sua estruturação num ficheiro *XML* devidamente formatado. Seguidamente, gerámos com sucesso a representação da cintura de asteroides, obtendo o seguinte resultado:

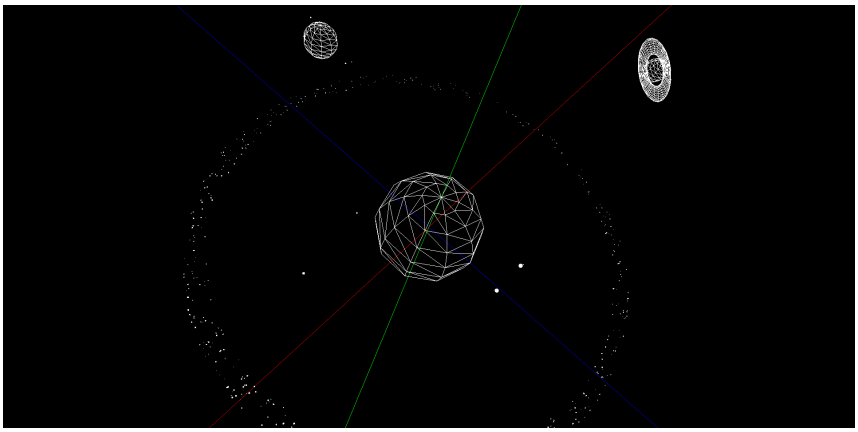

Figura 7: Representação do Sistema Solar (ficheiro `solar_system.xml`).

Para validar a consistência do nosso desenvolvimento, realizámos um teste adicional através de um *script* Python que alinhou todos os planetas segundo um mesmo ângulo orbital. Esta operação permitiu confirmar a correta parametrização do sistema, como evidenciado na imagem a seguir:

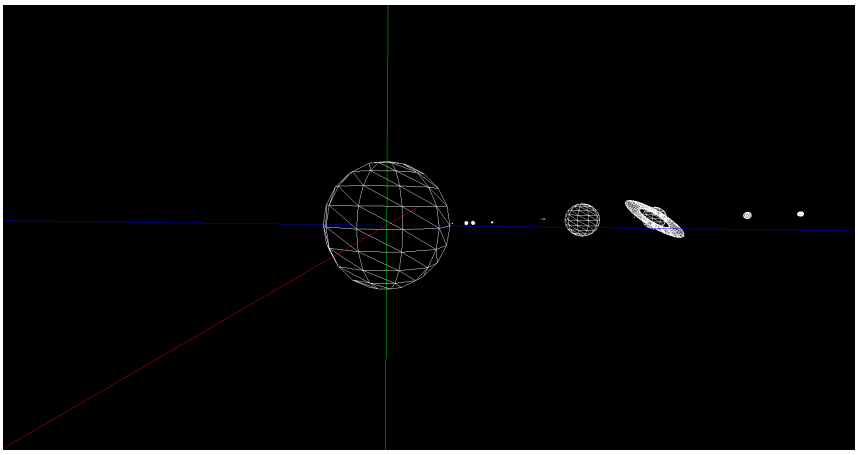

Figura 8: Sistema Solar com alinhamento orbital (ficheiro `solar_system_align.xml`).

5. Conclusões e Trabalho Futuro

O desenvolvimento desta fase do projeto revelou-se bastante satisfatório, deixando-nos extremamente satisfeitos com os resultados já alcançados. Conseguimos implementar com sucesso todas as funcionalidades planeadas, criando bases sólidas para a continuação do trabalho.

Para além de termos implementado com sucesso todas as funcionalidades básicas do sistema solar, já demos início a dois avanços significativos: a integração de *Vertex Buffer Objects (VBOs)* para otimização de desempenho e o desenvolvimento de uma interface gráfica (GUI) que tornará a interação com a aplicação mais intuitiva para os utilizadores.

Bibliografia

- [1] «vcpkg - Open source C/C++ dependency manager from Microsoft». Disponível em: https://vcpkg.io/en/
- [2] «CMake». Disponível em: https://cmake.org/
- [3] «OpenGL - The Industry Standard for High Performance Graphics». Disponível em: https://www.opengl.org/
- [4] «GitHub - Build software better, together». Disponível em: https://github.com/
- [5] Song Ho Ahn, «OpenGL Tutorials». Disponível em: https://www.songho.ca/opengl/
- [6] «Eyes on the Solar System». NASA. Disponível em: https://eyes.nasa.gov/apps/solar-system/#/home
- [7] «Devstronomy - Planetary Data Reference». Disponível em: https://devstronomy.martinovo.net/

Definição de Ponto

A estrutura `Ponto` representa uma coordenada num espaço tridimensional e é definida como:

```
typedef struct Point {
  float x;
  float y;
  float z;

  Point(float x_val = 0.0f, float y_val = 0.0f, float z_val = 0.0f)
      : x(x_val), y(y_val), z(z_val) {}

} Point;
```

ModelGroup

```
ModelGroup::ModelGroup(
    std::vector<std::string> models, std::vector<ModelGroup> subModelGroups,
    std::vector<Point> points,
    std::array<std::array<double, 4>, 4> transformationMatrix) {
  this->models = models;
  this->subModelGroups = subModelGroups;
  this->vertices = points;
  this->transformationMatrix = transformationMatrix;
}
```