Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Computação Gráfica

Ano Letivo de 2022/2023

Parte 2 - Transformações Geométricas

Filipa Gomes (A96556) Miguel Gomes (A93294) Pedro Pacheco (A61042)
Rita Lino (A93196)

14 de abril de 2023

CG

Resumo

Neste relatório explicitaremos as nossas decisões na realização da segunda fase do trabalho da Unidade Curricular de Computação Gráfica. Visando, deste modo, explicar como foram implementadas as Transformações Geométricas no *engine*, que lê a configuração dos ficheiros e renderiza os modelos.

Área de Aplicação: Computação Gráfica.

Palavras-Chave: OpenGL, C++, XML, Transformações geométricas, Primitivas gráficas.

Índice

Lista de Figuras

1 Introdução

O objetivo desta tarefa prática é desenvolver um motor 3D baseado em grafos de cena e fornecer exemplos de uso que demonstrem o seu potencial. A tarefa está dividida em quatro fases, sendo esta a segunda. Para cada fase, serão fornecidos um conjunto de ficheiros XML de configuração para fins de teste e avaliação. Cada configuração é acompanhada pela respetiva saída visual.

Neste relatório fazemos uma descrição das funcionalidades pedidas para a segunda fase do projeto. Assim sendo, esta fase envolve a implementação de transformações geométricas base no motor que por sua vez lê um ficheiro de configuração XML e exibe os modelos num ambiente de apresentação gráfica.

Esta fase requer a implementação de três Transformações Geométricas: a translação (*translate*), a rotação sobre um eixo (*rotate*) e a escala (*scale*). Foi também desenvolvido um grafo de cena para representação do Sistema Solar, sendo esta desenvolvida recorrendo a um *script* em *Python*.

Em resumo, a segunda fase deste projeto expande as bases criadas para o motor 3D baseado em grafos de cena, recorrendo a Transformações Geométricas com o objetivo final de exibir os modelos de forma visualmente atraente.

2 Engine

2.1 Transformation

Uma vez que a ordem das operações é relevante para o resultado final, estas são armazenadas num vetor pela ordem que aparecem no XML. Para tal esta classe ***Transformation*** baseia-se simplesmente na aplicação das transformações geométricas solicitadas no cenário.

A definição da classe ***Transformation*** é feita da seguinte forma.

```
1
2  enum class TransformationType {
3      None,
4      Translate,
5      Rotate,
6      Scale
7  };
8
9  class Transformation {
10     public:
11         TransformationType type;
12         int angle;
13         float x;
14         float y;
15         float z;
16
17         Transformation(TransformationType type, float x, float y, float z, int
       angle);
18         Transformation();
19         auto show() -> string;
20         void apply();
21 };
```

Após alguma reflexão chegamos a conclusão que esta implementação não seria a mais correta, tendo como objetivo a correção da mesma para a próxima fase do projeto.

2.2 Scene

Para facilitar o armazenamento da cena e dos seus componentes foram desenvolvidas a classe ***Scene*** e suas classes descendentes ***Group***.

```
1  class Scene {
2
3      public:
4      Camera camera;
5      Group group;
6
```

```
    Scene(Camera camera, Group group);
    Scene();

    auto render(bool picker = false) -> void;
    auto load_models() -> void;
};

class Group {

    public:
        vector<Group> subgroups;
        vector<Model> models;
        vector<Transformation> transformations;

        Group(vector<Group> subgroups, vector<Model> models, vector<
    Transformation> transformations);
        Group();

        //(...)

        auto applyTransformations();
        auto render(bool picker = false) -> void;
        auto load_models() -> void;
};
```

2.3 *XML Parsing*

Uma vez que o ***Parser*** já estava desenvolvido para a realização da primeira fase deste projeto foi apenas necessário a *atualização* de como os campos eram interpretados. Com esse objetivo o ***Parser*** original foi redesenhado com o intuito de o tornar mais modular e para suportar *parsing* de subgrupos e das respetivas transformações.

O método responsável pelo *parsing* de um Grupo foi definido da seguinte forma:

```
auto Parser::parse_group(XMLElement* element_group) -> Group {
    Group group;
    XMLElement* child_element = element_group->FirstChildElement();
    while (child_element != nullptr){
        if (strcmp(child_element->Name(), "group") == 0){
            group.addSubgroup(parse_group(child_element));
        }
        else if (strcmp(child_element->Name(), "models") == 0){
            group.models = parse_models(child_element);
        }
        else if (strcmp(child_element->Name(), "transform") == 0){
            group.transformations = parse_transformations(child_element);
        }
        child_element = child_element->NextSiblingElement();
    }
    return group;
}
```

Este é dependente de outros 2 métodos: *parse_models* e *parse_transformations*, mas também de si mesma para suportar subgrupos.

```
auto Parser::parse_transformations(XMLElement* element_transform) -> vector<
    Transformation>{
    vector<Transformation> transformations;
    TransformationType type;
    float x, y, z;
    int angle;
    XMLElement* child_element = element_transform->FirstChildElement();
    while (child_element != nullptr){
        if (strcmp(child_element->Name(), "translate") == 0){
            type = TransformationType::Translate;
            x = atof(child_element->Attribute("x"));
            y = atof(child_element->Attribute("y"));
            z = atof(child_element->Attribute("z"));
        }
        else if (strcmp(child_element->Name(), "rotate") == 0){
            type = TransformationType::Rotate;
            angle = atoi(child_element->Attribute("angle"));
            x = atof(child_element->Attribute("x"));
            y = atof(child_element->Attribute("y"));
            z = atof(child_element->Attribute("z"));
        }
        else if (strcmp(child_element->Name(), "scale") == 0){
            type = TransformationType::Scale;
            x = atof(child_element->Attribute("x"));
            y = atof(child_element->Attribute("y"));
            z = atof(child_element->Attribute("z"));
        }
        transformations.push_back(Transformation(type, x, y, z, angle));
        child_element = child_element->NextSiblingElement();
    }
    return transformations;
}

auto Parser::parse_models(XMLElement* models_element) -> vector<Model>{
    vector<Model> models;
    XMLElement* model_element = models_element->FirstChildElement();
    while (model_element != nullptr){
        string file = model_element->Attribute("file");
        string name = model_element->Attribute("name") ? model_element->
    Attribute("name") : file;
        models.push_back(Model(file, name));
        model_element = model_element->NextSiblingElement();
    }
    return models;
}
```

3 Sistema Solar

3.1 Script

Com o auxílio de algumas fontes incluídas nas ***Referências*** deste relatório relativas às dimensões do Sol, Planetas e os seus Satélites naturais correspondentes, e das distâncias entre si, foi produzido um *script* que permite gerar um ficheiro XML cuja interpretação feita pelo *engine* resulta no Sistema Solar.

De maneira que as distâncias e dimensões dos astros e planetas fossem mais realísticas, foi necessário realizar uma escala possibilitando a sua visualização.

A indentação típica de um ficheiro XML foi possível devido ao uso de várias bibliotecas da linguagem de Python, a leitura dos dados dos astros presentes em ficheiros ***.csv*** e a aleatoriedade da colocação dos asteroides presentes na cintura de asteroides.

Um ficheiro XML foi gerado com o intuito de armazenar todos os planetas e respetivos satélites naturais, bem como todos os (500) asteroides que constituem a cintura de asteroides e o anel de Saturno *(toro)*.

É utilizado um ficheiro ***.3d*** para elementos de menor dimensão, sendo este uma esfera com baixo número de stacks e slices. Visando criar um elemento Sol realista, utilizou-se um ficheiro ***.3d*** de uma esfera com um maior número de stacks e slices de modo que seja o mais parecido com uma esfera verdadeira.

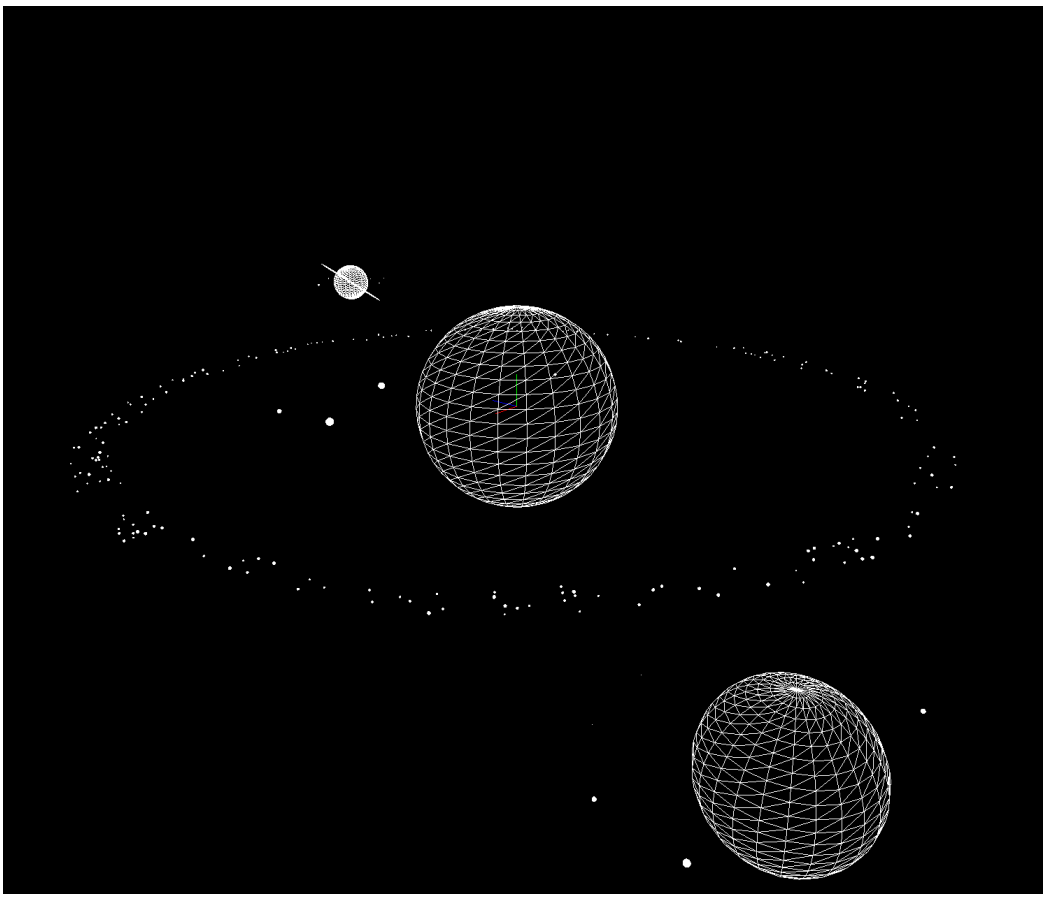

Figura 3.1: Sistema Solar - Wireframe

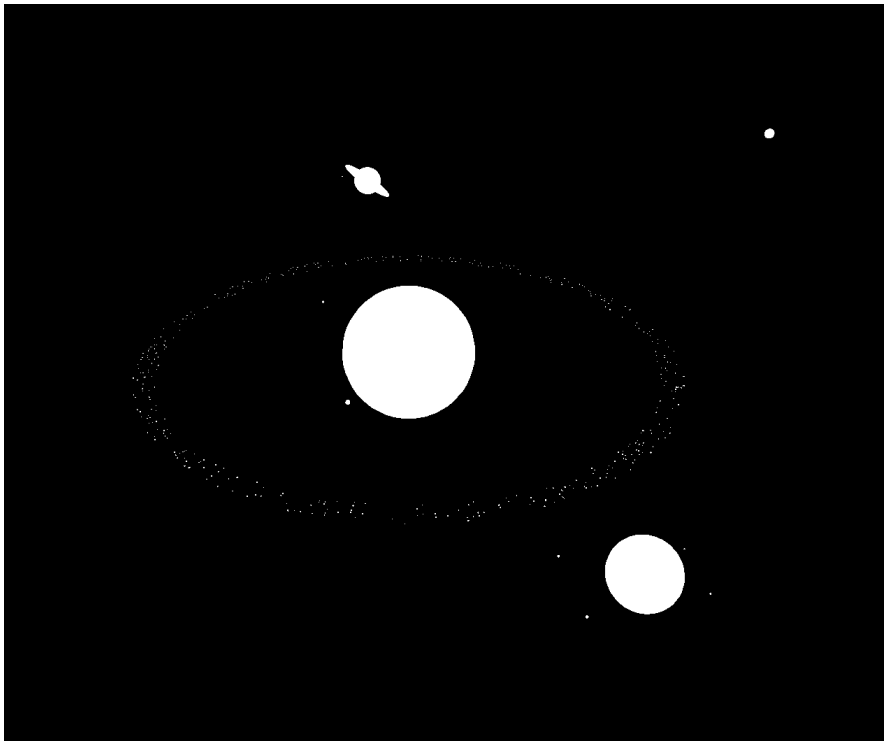

Figura 3.2: Sistema Solar - Solid

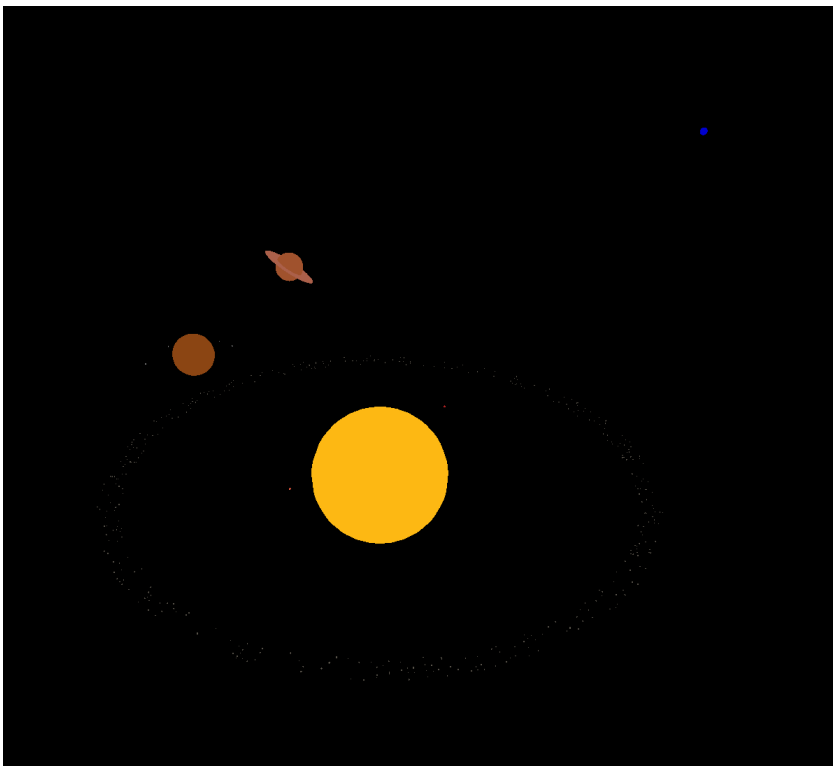

Figura 3.3: Sistema Solar - Color

3.2 Cores

Para ser obtida uma representação fiável e mais percetível do Sistema Solar foram utilizadas cores para cada astro, sendo que, as cores dos satélites naturais e dos asteroides são genéricas, enquanto que as cores dos planetas e do anel de Saturno são as suas cores reais.

A sua realização foi feita com o auxílio da biblioteca ***Pillow*** de Python, que permite a conversão de expressões hexadecimais para RGB. Estas expressões estão contidas no ficheiro *.csv* tendo cada astro a sua cor.

O programa suporta modelos sem qualquer cor atribuída, sendo definida uma cor branca ***default (#ffffff)***, este suporta também um atributo no modelo com a cor em hexadecimal em vez de uma *tag* inteira para os campos da cor do modelo.

```
<group name="Earth">
  <!-- Earth -->
  <models>
    <model file="../models_generated/sphere.3d">
      <color r="65" g="105" b="225" />
    </model>
  </models>
</group>

<group name="Earth">
  <!-- Earth -->
  <models>
    <model file="../models_generated/sphere.3d" color="#4169E1">
  </models>
</group>
```

4 Testes

Por meio de ficheiros disponibilizados é possível fazer a validação da implementação das ***Transformações Geométricas*** usando os cenários de teste. Seguem os cenários resultantes dos mesmos:

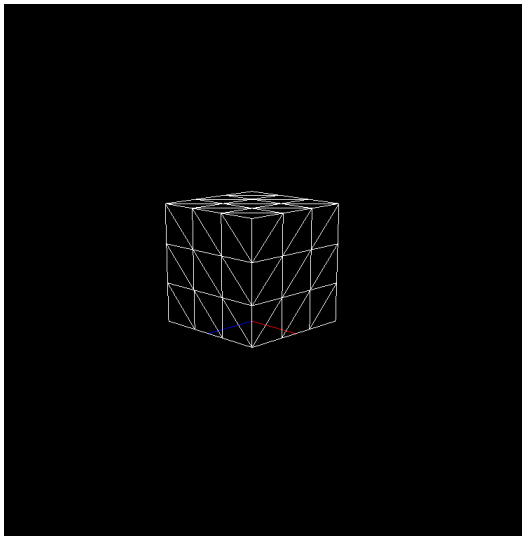

Figura 4.1: Figura de teste_2_1

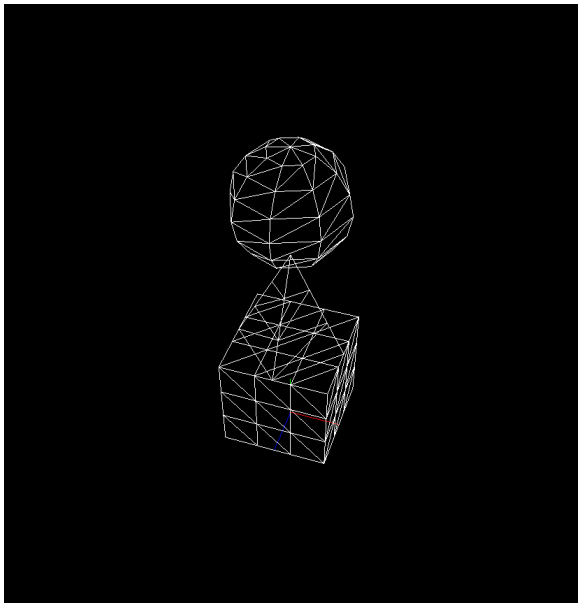

Figura 4.2: Figura de teste_2_2

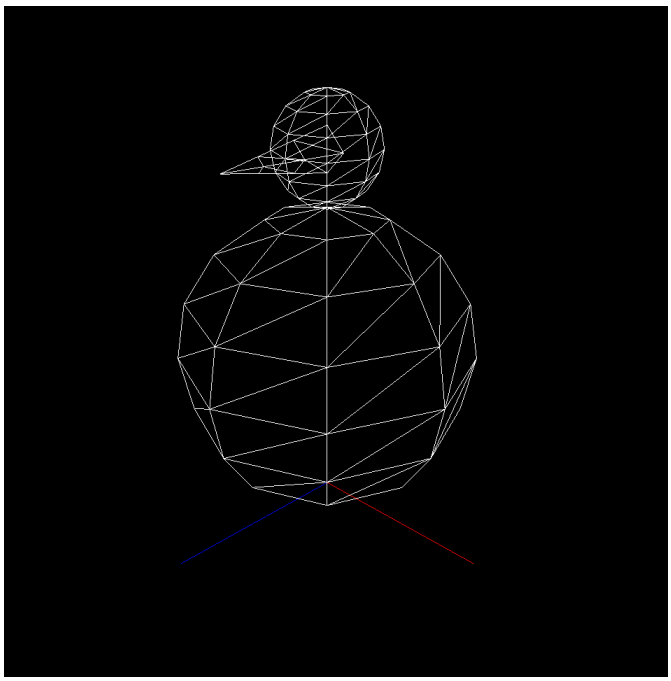

Figura 4.3: Figura de teste_2_3

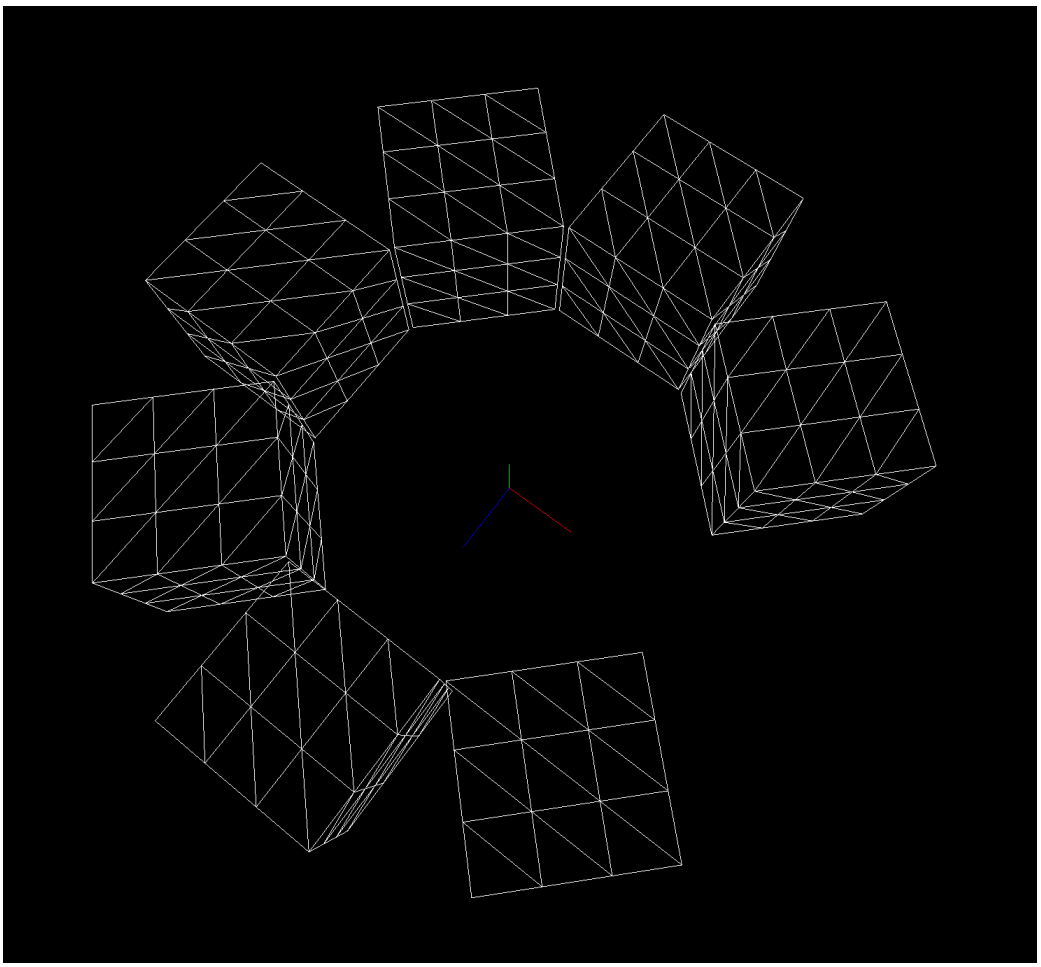

Figura 4.4: Figura de teste_2_4

5 Conclusões e trabalhos futuros

A realização desta fase do projeto permitiu-nos consolidar os conhecimentos adquiridos durante as aulas acerca de transformações de modelos 3D, matrizes e câmaras. Com isto, fomos capazes de construir um modelo inicial do sistema solar, que vai ser necessário nas próximas fases deste trabalho prático.

Considera-se que foi efetuado um bom trabalho, e que o programa implementa uma boa variedade de funcionalidades. No entanto, no futuro, será melhorado relativamente ao seu aspeto estético, utilizando texturas e iluminação e no aspeto funcional com a implementação de otimizações e funcionalidades extra.

5.1 Otimizações

O cenário do *Sistema Solar* contem centenas de astros, o que leva ao aparecimento de *bottlenecks* que não estiveram em consideração durante a implementação do *engine* na primeira fase e na expansão do mesmo nesta fase. O maior *bottleneck* encontrado foi a leitura dos ficheiros *model* uma vez que o mesmo ficheiro era lido centenas de vezes. Esse problema produz também uma elevada utilização de memória. Como implementação para a próxima fase pretendemos corrigir esses *bottlenecks* e quais queres outros que apareçam durante o desenvolvimento. Além disso, pretendemos trocar o vetor de pontos dos modelos por buffers armazenados na placa gráfica (*VBOs*).

5.2 Extras

O *engine* contém algumas funcionalidades extra, como, por exemplo, suporte de cores por *model*, como foi possível visualizar na figura ***Sistema Solar - Color***

As funcionalidades extra ainda em desenvolvimento são *Pick and Click* para identificação de vários modelos com o nome desse grupo/modelo com utilização do rato e a criação de uma câmara *FPV*. Pretendem se que estas funcionalidades estejam implementadas completamente até a próxima fase deste projeto.

6 Referências

1. **Dados sobre planetas e luas (NASA)**. (04/04/2022).
2. **Distancia relativa entre planetas e luas (National Geographic)**. (04/04/2022).
3. **Referência para o *'script'* gerador do *Sistema Solar* (Github)**. (12/04/2023).