

Projeto de Computação Gráfica - Fase 2

Diogo Braga A82547 João Silva A82005
Ricardo Caçador A81064 Ricardo Veloso A81919

25 de Março de 2019

Resumo

Neste relatório é apresentada a segunda fase dum projeto no qual a intenção é desenvolver um mecanismo baseado em gráficos 3D e fornecer exemplos de uso que mostrem o seu potencial. Este projeto é desenvolvido no âmbito da unidade curricular de Computação Gráfica.

Conteúdo

1	Introdução	3
2	Estrutura da Pasta do Projeto	3
3	Parser XML	4
3.1	Ficheiro	4
3.2	Funcionamento	5
4	Estruturas de Dados	7
4.1	Tree	7
4.2	Figure	7
4.3	Color	7
4.4	Scale	8
4.5	Rotation	8
4.6	Translation	8
4.7	Point	8
5	Render Scene	10
6	Generator	11
6.1	Cintura de Asteróides	11
6.1.1	Algoritmo	11
6.2	Torus	12
6.2.1	Algoritmo	13
7	Upgrades da Fase Anterior	14
8	Scenes	15
8.1	Primeiro Modelo	16
8.2	Segundo Modelo	17

9 Conclusão	18
10 Bibliografia	18

1 Introdução

Esta segunda fase tem como objetivo a realização de algumas etapas, nomeadamente:

1. A criação dum parser *XML* mais conciso;
2. A alteração/evolução da estrutura dos ficheiros *XML*;
3. A criação de estruturas capazes de armazenar as figuras necessárias;
4. A criação de funções que leiam as estruturas e as desenhem;
5. A criação dum modelo estático do sistema solar.

De seguida iremos apresentar todos os algoritmos e decisões relativos à realização destas etapas, bem como as respetivas explicações de cada passo. Serão ainda apresentadas figuras e esquemas que ilustrem os passos do processo desenvolvido.

2 Estrutura da Pasta do Projeto

Para um entendimento mais claro da estrutura do projeto, achamos por bem referenciar a estrutura da pasta do projeto. O projeto entregue contém para além do relatório, 4 pastas como é possível verificar na seguinte figura.

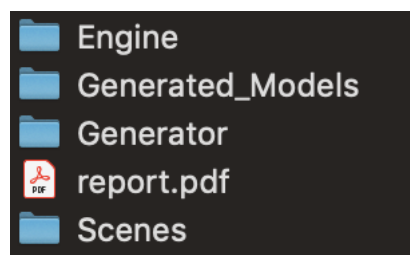


Figura 1: Estrutura da pasta.

Na pasta **Engine** residem os ficheiros relativos ao programa *engine*, bem como as bibliotecas (.h) criadas para o efeito. Contém ainda um ficheiro de configuração *cmake* e os ficheiros relativos à biblioteca *tinyxml2*.

Na pasta **Generator** residem os ficheiros relativos ao programa *generator*. Contém também um ficheiro de configuração *cmake*.

Na pasta **Generated_Models** residem os ficheiros que contêm os pontos gerados para cada figura, criados pelo programa *generator*.

Na pasta **Scenes** residem os ficheiros *XML* que contêm as estruturas dos sistemas solares desenvolvidos para esta fase do projeto. Construímos dois ficheiros deste tipo, um que contém os raios relativos de cada planeta e do sol baseados na realidade (grande disparidade entre as figuras criadas), e o outro que é mais elucidativo e perceptível.

3 Parser XML

3.1 Ficheiro

Devido aos requerimentos desta fase, foi necessário criar um novo ficheiro XML com o intuito de representar o sistema solar, que inclui: o sol, os planetas, e os satélites naturais destes mesmos.

Esta nova cena é encabeçada pelo grupo que faz referencia ao sol, e possui vários sub-grupos. Estes sub-grupos são referentes aos planetas, e possuem também sub-grupos. Estes sub-grupos dos planetas são referentes aos satélites naturais que cada um possui.

Todos os sub-grupos herdam as transformações geométricas presentes no grupo ao qual pertencem. Nesta fase, estas transformações são: translações, rotações e escalas.

Na figura 2 é possível visualizar um excerto do ficheiro XML por nós usado no qual constam o Sol, a Terra e a Lua.

```
<scene>
  <group> <!-- Sun -->
    <rotate axisX="0" axisY="1" axisZ="0" angle="90"/>
    <models>
      <model file="sun.3d" R="0.945" G="0.412" B="0.082" />
    </models>

    (...)

  <group> <!-- Earth -->
    <translate X="125" Y="0.0" Z="0.0" />
    <models>
      <model file="planet.3d" R="0.145" G="0.192" B="0.388" />
    </models>
    <group> <!-- Moon -->
      <translate X="0" Y="1.5" Z="2.0" />
      <models>
        <model file="moon.3d" R="0.412" G="0.388" B="0.376" />
      </models>
    </group>
  </group>

  (...)

```

Figura 2: Exemplo dum ficheiro usado no Parser.

De referenciar que para facilitar a parte de coloração de cada corpo do sistema solar, acrescentamos ao **model** 3 novos valores que se referem à cor que determinado objeto terá. Estes campos são nomeadamente **R**, **G** e **B**.

3.2 Funcionamento

O parser é inicializado na função **parser_XML**. Esta função tem a principal funcionalidade de atualizar a variável global **Tree t**, sendo esta a estrutura que irá armazenar a informação retirada do ficheiro XML. O algoritmo deste processo é demonstrado a seguir:

Função **parser_XML**:

1. Abre ficheiro \Rightarrow "scene.xml"
 - (a) Caso resulte em erro, termina execução.
2. Filtra primeiro elemento \Rightarrow scene
 - (a) Caso não seja obtido, termina execução.
3. A partir do nodo atual, procura o primeiro filho, \Rightarrow group
 - (a) Caso não seja obtido, termina execução.
4. O nodo (group) obtido no passo anterior é usado como parâmetro na função **parseGroup**.

Função **parseGroup**:

1. Iterar sobre todos os filhos do nodo passado como parâmetro, até algum ser null:
 - (a) Caso seja **translate**:
 - i. Obtém os valores e atribui-os à figura \Rightarrow X,Y,Z
 - (b) Caso seja **rotate**:
 - i. Obtém os valores e atribui-os à figura \Rightarrow angle,X,Y,Z
 - (c) Caso seja **scale**:
 - i. Obtém os valores e atribui-os à figura \Rightarrow X,Y,Z
 - (d) Caso seja **models**:
 - i. Iterar sobre os nodos filhos existentes:
 - A. Obtém o nome do ficheiro da figura e invoca com esse parâmetro a função \Rightarrow **loadFigure**
 - B. Obtém os valores das cores e atribui-os à figura \Rightarrow R,G,B

Função **loadFigure**:

- i. A função recorre aos ficheiros do **Generated Models**.
- ii. Obtém do ficheiro enviado como parâmetro os pontos dos triângulos que vão criar a figura.
- iii. Atribui esses valores à figura da **Tree**.

- (e) Caso seja group:
- i. Executa recursivamente a função `parseGroup` sendo o parâmetro inicial o nodo atualmente a ser iterado.
 - ii. As figuras possuem um vector de `trees`, que serão encaradas como `subtrees` \Rightarrow `children`
 - iii. Finalizada a recursão, o resultado é colocado na estrutura de dados referida no passo anterior.
 - iv. Desta forma, todas as transformações dos nodos filhos vão ser influenciadas pelas dos seus precedentes.
2. Retorna a `Tree` passada como parâmetro inicialmente, agora com as figuras e suas transformações já armazenadas.
3. Fim

4 Estruturas de Dados

Quanto à estrutura de dados, o grupo considerou que a mais adequado seria a de representar o sistema solar como uma árvore, sendo que, analogamente ao primeiro nodo da árvore estaria o Sol e os planetas seriam os *filhos* do Sol nessa árvore. Em consequência, as luas seriam *filhos* dos planetas e *netos* do Sol. Para esta representação ser efetiva foram criadas varias estruturas que serão apresentadas a seguir neste relatório.

4.1 Tree

A classe Tree será usada para representar uma Árvore. Uma instância de Árvore tem uma **Figure** principal e um **vector** com os *filhos* que tem.

```
class Tree{
public:
    Figure head_figure;
    vector<Tree> children;
};
```

Figura 3: Estrutura que define instâncias de uma Árvore

4.2 Figure

A classe Figure é composta por um inteiro com o número de triângulos que constitui essa figura, um vector com os pontos todos da figura. Possui também as 3 transformações geométricas que são feitas sobre a mesma - Translation, Rotation, Scale. Contém ainda uma Color, para definir a cor da figura.

```
class Figure{
public:
    int num_triangles;
    vector<Point> points;
    Translation translation;
    Rotation rotation;
    Scale scale;
    Color color;
};
```

Figura 4: Estrutura que define instâncias de uma Figura

4.3 Color

A classe Color é composta por 3 floats - r,g e b. Estes floats juntos definem a cor para usar numa figura.

```
class Color{
public:
    float r,g,b;
};
```

Figura 5: Estrutura que define instâncias de Cor.

4.4 Scale

A classe Scale contém 3 floats - x,y e z, que são os parâmetros de redimensionamento da figura que queremos. Contém ainda um bool para verificar se a transformação existe.

```
class Scale{
public:
    Scale() :empty(true) {}
    bool empty;
    float x,y,z;
};
```

Figura 6: Estrutura que define instâncias de Scale

4.5 Rotation

A classe Rotation é composta por 4 floats - as coordenadas dos eixos e o ângulo que queremos rodar a figura. Tem também um bool empty que apenas permite a leitura caso o seu valor seja true.

```
class Rotation{
public:
    Rotation() :empty(true) {}
    bool empty;
    float angle,x,y,z;
};
```

Figura 7: Estrutura que define instâncias de Rotation

4.6 Translation

A classe Translation é também composta pelas 3 coordenadas dos eixos e pela verificação de que a figura existe mesmo e queremos lê-la.

```
class Translation{
public:
    Translation() :empty(true) {}
    bool empty;
    float x,y,z;
};
```

Figura 8: Estrutura que define instâncias de Translation

4.7 Point

A classe Point, como o nome indica, é um ponto no sistema, ou seja, é composta por 3 floats - x, y e z.


```
class Point{  
    public:  
        float x,y,z;  
};
```

Figura 9: Estrutura que define instâncias de Point

5 *Render Scene*

Para o rendering do projeto, utilizamos uma pesquisa em profundidade na árvore representando todas as figuras e as suas translações, rotações e escalas.

Assim, a função `renderScene()`, chama duas funções. A função `drawAxis()` que representa os 3 eixos de coordenadas na cena e a função `drawTree` que desenha o Sistema Solar. Esta função `drawTree` recebe como parâmetro uma árvore (`Tree`) e o seu algoritmo pode ser explicado assim:

1. Efetua o push inicial \Rightarrow `glPushMatrix()`;
2. Verifica se a figura atual tem alguma translação;
(a) Caso tenha, efetua a translação com a `glTranslatef()`;
3. Verifica se a figura atual tem alguma rotação;
(a) Caso tenha, efetua a translação com a `glRotatef()`;
4. Verifica se a figura atual tem alguma escala;
(a) Caso tenha, efetua a translação com a `glScalef()`;
5. Após as transformações geométricas serem feitas, através da `drawFigure()`, é desenhada a figura do nodo atual.
6. Para o resto do `vector<Tree>`, é feito todo este procedimento.
7. Finalizando, é efetuado o `glPopMatrix()`;

A função `drawFigure()`, que tem como parâmetro uma `Figure`, é responsável por desenhando uma figura e o seu algoritmo pode ser explicado assim:

1. Colocar num `vector<Point>` todos os pontos que compõem a figura;
2. Colocar numa estrutura `Color`, a cor da figura;
3. Iniciar o desenho com `glBegin(GL_TRIANGLES)` e com `glColor3f(com os parâmetros que colocamos na estrutura Color0)`;
4. Iterando sobre o vector dos pontos:
(a) Colocar numa estrutura `Point`, o ponto actual da iteração (`current_point`);
(b) Através da `glVertex3f(current_point.x, current_point.y, current_point.z)` desenhar o ponto atual;
5. Terminada a iteração, termina o desenho da figura com `glEnd()`;

6 Generator

Nesta segunda fase do projeto usamos o programa criado na primeira fase (**Generator**) para criar todas as figuras com o intuito de alimentar os ficheiros *XML* e consequentemente o programa **Engine**.

6.1 Cintura de Asteróides

Para além da sua funcionalidade inicial, achamos que seria benéfico para o nosso modelo estático do sistema solar ter uma cintura de asteróides. Tal tarefa era bastante difícil de realizar, asteróide a asteróide, pelo que decidimos adicionar uma nova funcionalidade ao **Generator**. Esta funcionalidade passa por gerar um ficheiro *XML* com o formato definido na secção 3.1 que contém todos os asteróides pertencentes à cintura de asteróides entre Marte e Júpiter.

Depois de ter o ficheiro com a cintura de asteróides gerada, temos que copiar o conteúdo deste ficheiro e colocá-lo corretamente no ficheiro *scene.xml* que é onde reside todo o modelo do sistema solar.

Um exemplo de como invocar esta função é:

```
⇒ ./Generator asteroid_belt 1000 150 160 asteroids.xml
```

Neste caso vamos gerar *XML* que será escrito no ficheiro *asteroids.xml*, que irá conter **1000** asteróides a uma distância do sol compreendida entre **150** e **160**.

O algoritmo utilizado é descrito a seguir.

6.1.1 Algoritmo

A função *generateAsteroids* recebe como parâmetro três inteiros que representam respetivamente, o número de asteróides a criar, o raio mínimo e o raio máximo a que estão do sol. Por último recebe o nome do ficheiro onde irá guardar o *XML* criado.

De seguida apresentamos uma imagem duma cintura de asteróides gerada pela nossa aplicação e a sua explicação através do algoritmo criado pelo grupo.

1. Inicializa-se um gerador de números pseudo-aleatórios, usando 1 como *seed* ⇒ `srand(1)`.
2. Iterar sobre o número de asteróides:
 - (a) Enquanto não for gerado um ponto válido:
 - i. Calcula-se um número aleatório para X, baseando-se na fórmula:
$$X \Rightarrow \text{rand}() \% (2 \times \text{raio máximo}) - \text{raio máximo}$$
 - ii. Calcula-se um número aleatório para Z, baseando-se na fórmula:
$$Z \Rightarrow \text{rand}() \% (2 \times \text{raio máximo}) - \text{raio máximo}$$

iii. Calcula-se a distância ao centro, calculando a hipotenusa:

$$\text{Hipotenusa} \Rightarrow \sqrt{X^2 + Z^2}$$

iv. Testa-se se a hipotenusa está compreendida entre o raio mínimo e o raio máximo.

A. Se sim, escreve em ficheiro e passa à próxima iteração para um novo asteróide.

B. Se não, entra no ciclo até tentar ter um ponto válido.

3. Fim

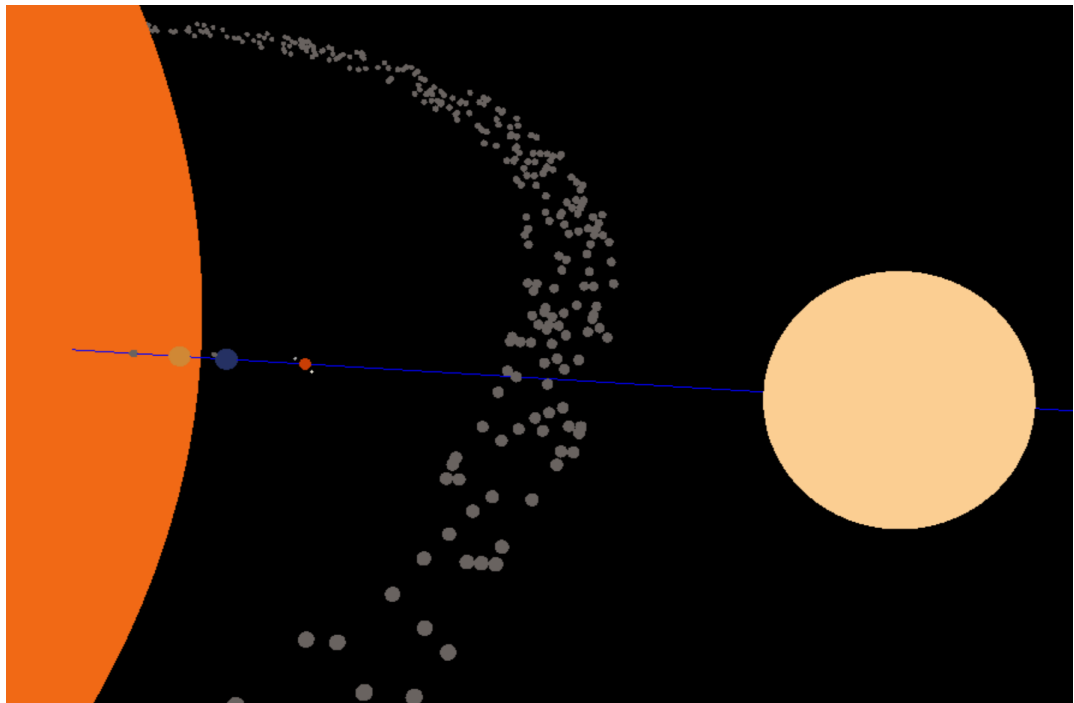


Figura 10: Imagem exemplificativa duma cintura de asteróides

6.2 Torus

Como Saturno na sua verdadeira natureza possui um anel, achamos por bem que o nosso modelo do sistema solar deveria conter algo que o representasse, uma vez que esta é uma característica bem conhecida deste planeta. Para tal achamos que uma *Torus* representaria bem o que pretendíamos.

Desta forma, criamos uma nova funcionalidade para o **Generator** que possibilita a criação de todos os pontos necessários para a criação duma *Torus*.

Um exemplo de como invocar esta função é:

\Rightarrow ./Generator torus 0.3 1.5 20 30 torus.3d

Neste caso iremos os pontos para a figura que serão guardados no ficheiro **torus.3d**, o raio de cada anel será de **0.3**, a distância (raio) do centro da figura até à figura será **1.5**, o número de lados de cada anel será **20** e finalmente o número de anéis que compõe a figura será **30**.

6.2.1 Algoritmo

7 Upgrades da Fase Anterior

Em relação à fase anterior houve alguns pontos que foram sujeitos a mudanças e otimizações pelo que estas serão relatadas nesta secção.

Uma das partes que foi sujeita a otimização foi o cálculo das coordenadas da câmara que na fase anterior continha demasiado código e algumas variáveis que não tinham nexo em ser usadas. Desta forma decidimos que esta parte deveria ser reescrita utilizando uma abordagem mais objetiva. O resultado é descrito na seguinte figura.

```
float alfa = 0.5f, beta = 0.5f, radius = 500.0f;
float camx, camy, camz, dx = 0.0, dy = 0.0, dz = 0.0;

void spherical2Cartesian() {
    camx = radius * cos(beta) * sin(alfa);
    camy = radius * sin(beta);
    camz = radius * cos(beta) * cos(alfa);
}
```

Figura 11: Código referente às coordenadas da câmara.

Não só a função que calcula as coordenadas para câmara mas também a própria função da câmara foi melhorada. Passou a poder mover-se consoante o ponto para o qual está a olhar. Desta forma possibilitou-nos a facilidade de podermos deslocarmo-nos para qualquer sítio que nos convenha. Tal alteração é descrita na seguinte figura.

```
gluLookAt(camx+dx, camy+dy, camz+dz,
          dx,dy,dz,
          0.0f,1.0f,0.0f);
```

Figura 12: Função que contém a matriz da câmara.

Outra das partes que sofreu algumas alterações foi a organização do código na pasta **Generator** que estava definido em ficheiros *.h* (plane, box, sphere e cone). Não devendo então código ser escrito em ficheiros deste género, excetuando apenas as definições e documentação, criamos um ficheiro *.h* que alberga todas as definições e documentação necessária e criamos ficheiros *.cpp* para cada figura. Desta forma todo o código ficou perfeitamente organizado.

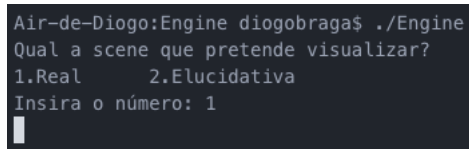
8 *Scenes*

Nesta fase do projeto é requerido um modelo estático do sistema solar, com o sol, os planetas e os satélites naturais do mesmo.

O grupo decidiu proceder à criação de duas cenas do sistema solar:

1. Modelo baseado nas diferenças reais entre os raios dos corpos presentes.
2. Modelo mais elucidativo e perceptível, que despreza parcialmente as diferenças reais entre os corpos presentes.

Na execução do programa *Engine* são disponibilizadas essas duas opções, e o utilizador deve escolher a que pretende visualizar. Deve, portanto, inserir o número do modelo que pretende. De seguida, na figura 13, é exemplificado um possível procedimento.



```
Air-de-Diogo:Engine diogobraga$ ./Engine
Qual a scene que pretende visualizar?
1.Real      2.Elucidativa
Insira o número: 1
```

Figura 13: Exemplo de um possível procedimento na interface.

8.1 Primeiro Modelo

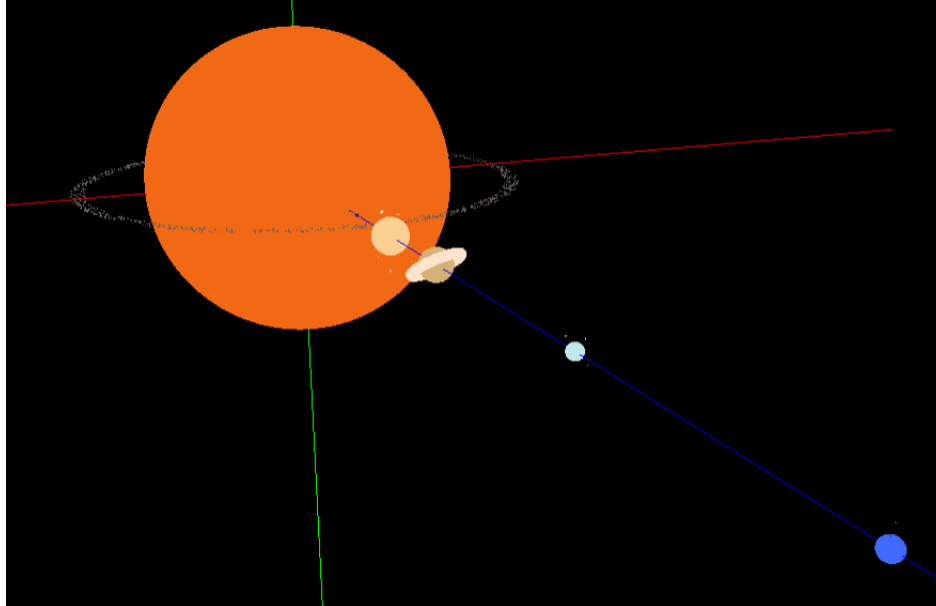


Figura 14: Imagem total do primeiro modelo.

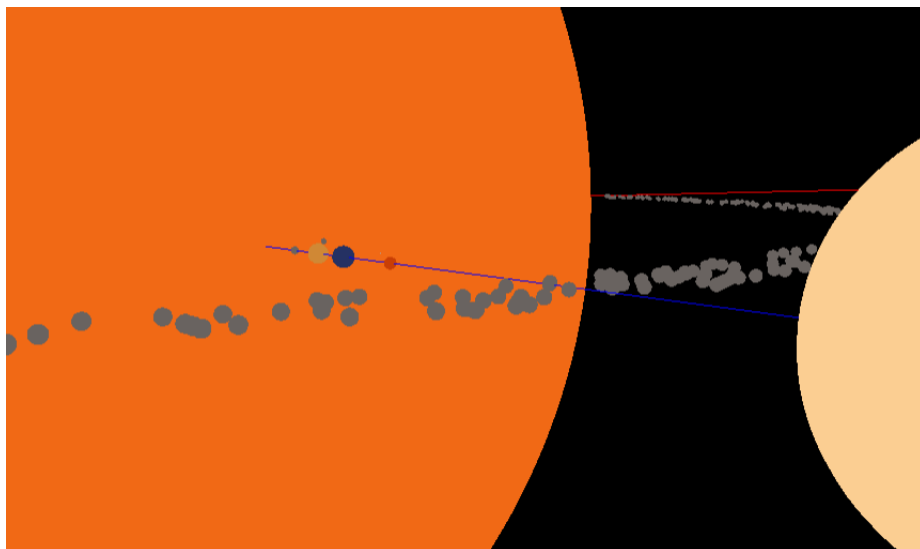


Figura 15: Imagem parcial do primeiro modelo.

8.2 Segundo Modelo

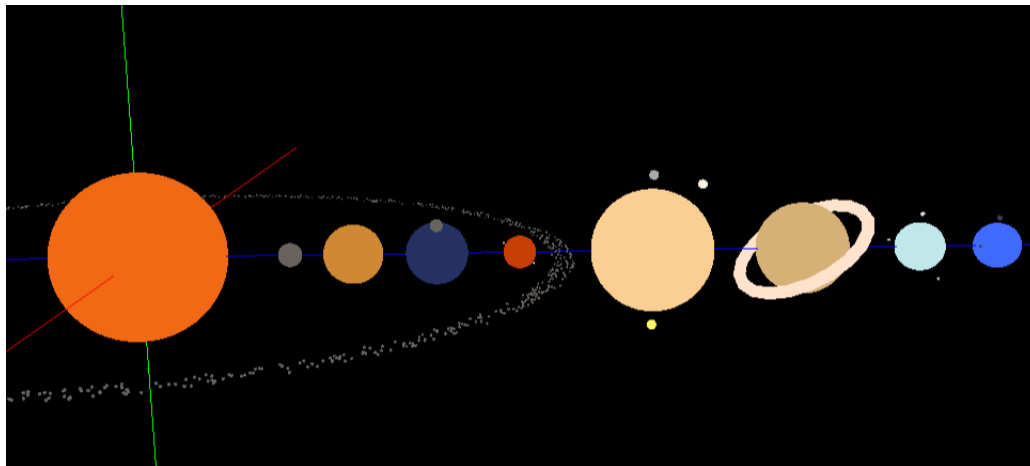


Figura 16: Imagem frontal do segundo modelo.

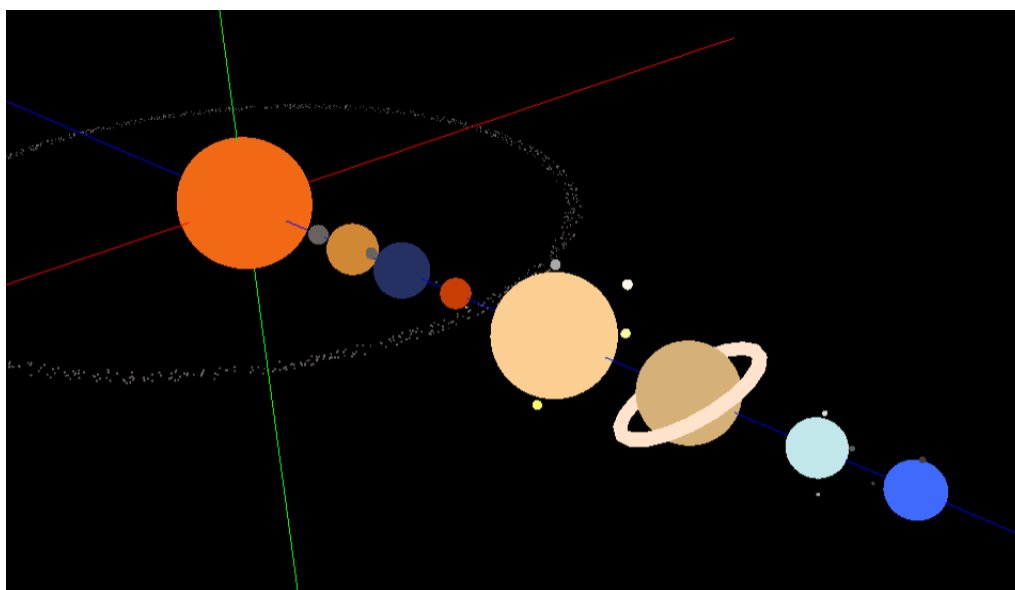


Figura 17: Imagem lateral do segundo modelo.

9 Conclusão

Após a conclusão desta fase do projeto, todos os membros do grupo se sentem mais à vontade com transformações geométricas e, conseqüentemente, com todos os conceitos apreendidos nas aulas que tiveram de ser colocados em prática nesta fase. Relativamente ao nível de execução desta fase o grupo no geral não sentiu grandes dificuldades, tendo apenas contado com um imprevisto, resolvido rapidamente que foi a dificuldade em *COMPLETAR COM A DIFICULDADE QUE TIVEMOS SÓ PARA ENCHER CHOURIÇOS E NÃO PARECER QUE SOMOS OS REIS DO CG*. Em jeito de conclusão e revisão final a esta fase, o grupo espera continuar num bom caminho para a realização de todas as fases deste projeto, que, para já, tem sido um projeto prazeroso.

10 Bibliografia

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! POR FAZER !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

- **Planetary Pixels Emporium :**

- https://www.suapesquisa.com/astronomia/distancia_sol_planetas.htm