

Universidade do Minho
Escola de Engenharia

Computação Gráfica

TRABALHO PRÁTICO

Fase final

Mestrado Integrado em Engenharia Informática

Grupo 42

Fábio Quintas Gonçalves, a78793

Francisco José Moreira Oliveira, a78416

Raul Vilas Boas, a79617

Vitor Emanuel Carvalho Peixoto, a79175

Ano letivo 2017/2018

Braga, Maio de 2018

ÍNDICE

1.	Introdução	1
2.	Generator.....	3
2.1	Gerar normais.....	3
2.2	Gerar pontos de textura.....	4
3.	Engine	6
3.1	Ler ficheiros “.3d”.....	6
3.2	Ler XML.....	6
3.2.1	Lighting	6
3.2.2	Texturas	7
3.3	Desenhar grupo	7
3.4	Função <i>lights</i>	8
3.5	Desenhar recorrendo a VBOs	8
3.6	Estruturas de dados.....	9
4.	Ficheiro XML.....	11
5.	Câmara de visualização.....	12
6.	Extras	13
7.	Resultado final	14
8.	Conclusões finais.....	16

1. INTRODUÇÃO

Nas últimas três etapas foi desenvolvido um sistema solar dinâmico, estando capaz de apresentar as seguintes funcionalidades:

- Generator:
 1. Pode gerar pontos para variadas figuras. Esses pontos são escritos em ficheiros “.3d” e podem ser ordenados quer para a sua leitura usando VBOs quer para o modo de geração de imagens imediato (usando triângulos).
 2. É capaz de ler um ficheiro “.patch” e gerar os pontos da figura no ficheiro “.3d” usando para isso o método matemático de superfícies de Bezier.
- Engine:
 1. É capaz de ler os pontos nos ficheiros “.3d” e gerar as figuras, tendo a habilidade de poder detetar se esse ficheiro “.3d” está preparado para ser desenhado recorrendo a VBOs ou através do método imediato.
 2. É capaz de ler um conjunto de transformações a cada objeto definidas no ficheiro “scene.xml” e aplicar à respetiva figura e a todas aquelas que pertencem ao seu subgrupo.
 3. Lê do ficheiro “scene.xml” um conjunto de pontos definidos com uma tag *Translate* e transforma esses pontos numa curva de *Catmull-Rom*, sendo que, para além disso, efetua um movimento de translação de cada planeta sob essa curva, de acordo com um tempo definido nessa mesma tag também.
 4. É capaz de efetuar dois tipos de rotação: uma rotação capaz de alterar o grau inclinação do astro; e uma rotação, com um tempo definido, capaz de efetuar um movimento de rotação desse mesmo astro.

Após serem desenvolvidas todas estas funcionalidades, é-nos agora pedido que finalizemos o projeto.

Esta fase apresenta-nos então, como requisitos os seguintes pontos:

- O *Engine* deverá estar equipado com *lighting*, tendo um ponto de luminosidade definido no centro do sistema solar e a capacidade de definir o *GL_MATERIAL* de cada objeto.
- Deverá também estar preparado para ler texturas para cada objeto.

- Isto tudo implica que a *Engine* deverá ser capaz de ler normais e pontos de textura através dos ficheiros “.3d” e gerar essas normais e pontos de textura usando VBOs ou de modo imediato, dependendo de como o ficheiro “.3d” foi organizado.
- Para os ficheiros “.3d” terem as normais e os pontos de textura, devemos também alterar o *Generator* para permitir a criação desses pontos, relativamente a cada uma das figuras presentes no nosso sistema solar.

O resultado final deverá ser um sistema solar dinâmico, com luminosidade e com as respetivas texturas aplicadas a cada astro.

2. GENERATOR

2.1 Gerar normais

De modo a podermos introduzir a iluminação no *Engine* precisamos de introduzir as normais de cada objeto no seu ficheiro “.3d”.

No caso da esfera, as normais são fáceis de implementar visto que o próprio ponto é o vetor, caso não multipliquemos pelo raio. Esse vetor inicia-se no centro da esfera até ao ponto.

Diferença entre um ponto e a normal desse ponto numa esfera.

```
1  pontos.push_back(Ponto(raio * cosf(stack*beta) * sinf(slice*alpha), raio * sinf(stack*beta), raio  
* cosf(stack*beta) * cosf(slice*alpha)));  
2  
3  normais.push_back(Ponto(cosf(stack*beta) * sinf(slice*alpha), sinf(stack*beta),  
cosf(stack*beta) * cosf(slice*alpha)));  
4
```

De facto, a única diferença entre o ponto e a normal desse ponto é mesmo que temos de multiplicar o x, y e z de cada coordenada pelo raio, no caso dos pontos.

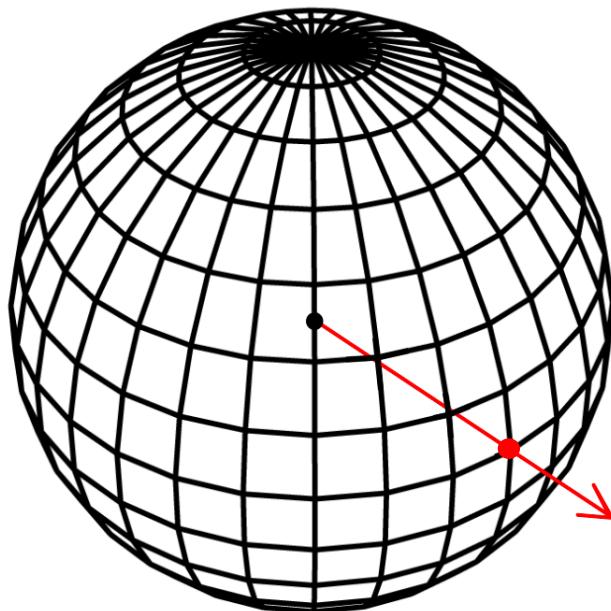


Figura 1 A normal de um ponto da esfera é o próprio ponto, tendo em vista um vetor com origem no centro da esfera e com coordenadas (0,0,0).

Outro dos objetos usados no nosso sistema solar é o anel. Neste caso, sendo o anel um objeto plano, não é necessário calcular qualquer normal, mas sim, para cada ponto, definir um vetor no sentido positivo ($y=1$) e negativo ($y=-1$) visto que o anel é visível quer do seu lado superior quer inferior.

Diferença entre um ponto e a normal desse ponto num anel.

```
1   fprintf(f, "%f %f %f\n", pontos1[i].getX(), pontos1[i].getY(), pontos1[i].getZ()); //ponto  
2   fprintf(f, "%f %f %f\n", pontos1[i].getX(), 1.0f, pontos1[i].getZ()); //normal
```

2.2 Gerar pontos de textura

Para conseguirmos inserir texturas nos objetos existentes no nosso sistema solar é necessário definir os pontos de textura para cada uma dessas figuras e inserir no ficheiro “.3d”.

No caso da esfera, esses pontos de textura são definidos dividindo 1 pelo número de slices ou stacks, visto que o comprimento e largura de cada textura é sujeito a uma escala de 0 a 1.

Cálculo do intervalo de iteração dos pontos de textura.

```
1   float itStacks = 1.0f/((float) stacks);  
2   float itSlices = 1.0f/((float) slices);
```

Assim, se uma esfera tiver 4 slices, os pontos de textura começarão em 0, depois 0.25 e por aí adiante. Foram definidas duas variáveis: cSlices e cStacks que representam a atual slice ou stack (current Slice/Stack) e inicializadas a 0.

Depois é apenas necessário calcular os pontos de textura para cada metade da esfera. Neste caso demonstrado em baixo, para a metade superior da esfera, a ordem de inserção dos pontos no ficheiro “.3d” é a mesma pela qual se inserem os pontos de coordenadas e as normais.

Cálculo dos pontos de textura para a metade superior da esfera.

```
1   for (int stack = stacks/2; stack >= 0; stack--){  
2       for (int slice=0; slice <= slices; slice++) {  
3           ...  
4           texturas.push_back(PontoText(cSlices, cStacks));  
5           texturas.push_back(PontoText(cSlices, cStacks - itStacks));  
6  
7           cSlices = cSlices + itSlices;  
8       }  
9       cStacks = cStacks - itStacks;  
10      cSlices = 0.0;  
11  }
```

No caso do anel, é relativamente fácil, uma vez que o anel tem pontos apenas nos seus limites interior e exterior. Assim, torna-se desnecessário calcular pontos de textura, visto que basta apenas, para cada coordenada do limite inferior definir como ponto de textura um ponto na parte mais à esquerda da figura e vice-versa.



Figura 2 Diferença na ordem de leitura das coordenadas, relativamente ao modo imediato (esquerda) e com VBOs (direita).

Pontos de textura dos anéis.

```
1  for (unsigned int i=0; i<pontos.size(); i++) {  
2      fprintf(f, "%f %f\n", 0.1f, 0.5f);  
3      fprintf(f, "%f %f\n", 0.9f, 0.5f);  
4  }  
5  for (unsigned int i=0; i<pontos.size(); i++) {  
6      fprintf(f, "%f %f\n", 0.1f, 0.5f);  
7      fprintf(f, "%f %f\n", 0.9f, 0.5f);  
8  }
```

3. ENGINE

3.1 Ler ficheiros “.3d”

Após preparamos o *Generator* para as normais e os pontos de textura, temos primeiramente de preparar a *Engine* para ser capaz de ler os ficheiros “.3d” e guardar a informação. Cada ficheiro “.3d” é estruturado da seguinte forma:

- N° de pontos (caso seja para desenhar com VBOs)
- Coordenadas de cada ponto (x, y e z separados por um espaço e uma linha por cada coordenada)
- Normais de cada ponto
- Pontos de textura de cada ponto

Assim, somos capazes de detetar se uma figura é para desenhar através de VBOs ou através do método imediato e também conseguimos armazenar as coordenadas, normais e pontos de textura, visto que o número de cada um destes pontos é o mesmo, então basta contar o número de linhas, dividir por 3 e iterar no primeiro terço e armazenar as coordenadas; no segundo terço e armazenar as normais; e no último terço e armazenar os pontos de textura.

Todos estes pontos são armazenados em vetores passados como argumentos da função.

3.2 Ler XML

Após registarmos nos vetores os pontos recolhidos do ficheiro “.3d” é necessário passar esses vetores às estruturas criadas, bem como ler a restante informação declarada no XML.

3.2.1 Lighting

Relativamente às novas funcionalidades desta etapa, tivemos de adicionar a capacidade de registar a informação contida na tag “lights”.

Sempre que encontramos essa tag começamos por dar *enable* ao *Lighting* e posteriormente iteramos por cada elemento dentro dessa tag, limitado sempre a 8, uma vez que o OpenGL apenas suporta 8 fontes de luz simultaneamente. Dentro dessa tag registamos o valor do *type* e as coordenadas posicionais da fonte de luz. Procuramos pelo valor de *red* de cada uma das variantes de luz (uma vez que se houver *red*, necessariamente, tem de haver *green* e *blue* também) e registamos cada uma das variantes. As

variantes possíveis são *diffuse*, *specular*, *emission* e *ambient*. Para além disso, podemos definir a atenuação da luminosidade (*attenuation*) e o ângulo de foco da luz (*spot*) dependendo do que foi definido no *type*.

Todos os dados capturados dentro da tag são armazenados numa nova estrutura, que, para além de armazenar o que referimos acima, armazena também um ID relativo a cada fonte de luz.

3.2.2 Texturas

Relativamente às texturas, estas serão “capturadas” na tag “models”, juntamente com o ficheiro “.3d” respetivo. O ficheiro da textura é referido no atributo *texture*, sendo que esse ficheiro é imediatamente carregado, recorrendo à função *loadTexture*, e armazenado na estrutura no seu formato *GLuint*.

Para além disso, cada figura pode ter associada a si uma variante na sua textura, relativamente ao seu “material”. As variantes existentes são *ambient*, *specular*, *diffuse* e *emission* e o processo de leitura destes atributos é exatamente igual ao efetuado nas variantes das fontes de luz.

3.3 Desenhar grupo

Para além do que foi desenvolvido nas fases anteriores, nesta última fase, adicionamos então a capacidade de definir um material à textura de cada objeto, usando para isso a função *glMaterial*.

Exemplo para a variante *emission*.

```
1 if (t.figure.material.emissive)
2     glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, t.figure.material.emissive);
```

No fim da função que desenha o grupo, voltamos a aplicar a função *glMaterial* para restaurar os valores padrão e não aplicar a objetos seguintes que não tenham definido nenhum tipo de textura de material.

Relativamente à textura do objeto, esta é inicializada antes ainda da função que gera a figura recorrendo a VBOs.

Inicialização da textura do objeto.

```
1 if (t.figure.buffer > 1){ //se for maior que 1 é para desenhar usando VBOs
2     glEnable(GL_TEXTURE_2D);
3     glBindTexture(GL_TEXTURE_2D, t.figure.texture);
4     drawFiguresVBOs(t.figure.figuras, t.figure.normais, t.figure.texturas, t.figure.buffer);
5     glBindTexture(GL_TEXTURE_2D, 0);
6     glDisable(GL_TEXTURE_2D);
7 }
```

Caso seja para desenhar imediatamente, recorrendo a triângulos, devemos apenas passar o vetor das normais, para além do vetor das coordenadas dos pontos.

No fim de desenhar o grupo, é executada a função que *renderiza* todas as fontes de luz.

3.4 Função *lights*

A função *lights* é a responsável por “ligar” todas as fontes de luz existentes. Para tal ela itera sobre o vetor das fontes de luz existentes e para cada uma define a posição dessa fonte e as “texturas” da fonte de luz, dependendo se estas se encontram registadas ou não (ou seja, se foram pedidas no XML).

Exemplo para a fonte de luz difusa.

```
1 if(it->color.diffuse)
2     glLightfv(GL_LIGHT0+it->id, GL_DIFFUSE, it->color.diffuse);
```

De referir ainda que a atenuação da fonte de luz só é executada se o atributo *type* estiver definido como POINT no XML e a direção e ângulo da fonte de luz só é executada se esse atributo estiver definido como SPOTLIGHT.

3.5 Desenhar recorrendo a VBOs

Para além do *buffer* criado na fase anterior, nesta temos de criar mais dois: um para as normais (relativas ao *lighting*); e outro para os pontos de textura (relativos às texturas). Para tal, criamos *buffers (arrays)* para cada um dos 2 novos vetores e “despejamos” todo o conteúdo dos vetores nesses *arrays*.

Depois foi necessário executar esses *buffers* do mesmo modo que fizemos na última fase para as coordenadas dos pontos:

Buffers.

```
1 //Coordenadas dos vértices
2 glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);
3 glBufferData(GL_ARRAY_BUFFER, i*sizeof(float), bufVertex, GL_STATIC_DRAW);
4 glVertexPointer(3, GL_FLOAT, 0, 0);
5
6 //Normais
7 glBindBuffer(GL_ARRAY_BUFFER, buffers[1]);
8 glBufferData(GL_ARRAY_BUFFER, i*sizeof(float), bufNormal, GL_STATIC_DRAW);
9 glNormalPointer(GL_FLOAT, 0, 0);
10
```

```

11 //Pontos de textura
12 glBindBuffer(GL_ARRAY_BUFFER, buffers[2]);
13 glBufferData(GL_ARRAY_BUFFER, n * sizeof(float), bufTextures, GL_STATIC_DRAW);
14 glVertexAttribPointer(2, GL_FLOAT, 0, 0);
15
16 glDrawArrays(GL_TRIANGLE_STRIP, 0, nPoints / 3);
17
18 delete[] bufVertex;
19 delete[] bufNormal;
20 delete[] bufTextures;
21 glDeleteBuffers(3, buffers);

```

E apagamos o conteúdo dos *arrays* e os *buffers* criados para libertar toda a memória alocada.

3.6 Estruturas de dados

Tivemos de criar algumas novas estruturas de dados:

- A classe **TexCoordinate** é uma classe nova que foi criada para aloca os pontos das texturas, visto que estes são bidimensionais, ao contrário dos pontos tridimensionais que usamos até esta fase.

Classe TexCoordinate.

```

1 class TexCoordinate {
2 public:
3     bool empty = true;
4     float x, y;
5 };

```

- A classe **Material** aloca os *arrays* das 4 variantes de texturas do material possíveis para um objeto. Esta classe é também usada dentro da classe para as *lights*, que irá ser falada a seguir.

Classe Material.

```

1 class Material {
2 public:
3     float *diffuse=NULL, *specular=NULL, *emissive=NULL, *ambient=NULL;
4 };

```

- A classe **Light** regista toda a informação relativa às fontes de luz já abordadas nos capítulos anteriores.

Classe Light.

```

1 class Light {
2 public:
3     string type;
4     float pos[4];
5     Material color;
6     float cons=NULL, quad=NULL, linear=NULL, spotCutOff=NULL, *spotDirection=NULL,
7         spotExponent=NULL;
8     unsigned int id;
9 }
```

- A classe **Figure** já existia, porém efetuamos algumas adições dignas de registo, tais como: os vetores das coordenadas das normais e dos pontos de textura, a textura já lida pela *loadTexture* e armazenada no seu tipo próprio e a textura de material desse objeto.

Classe Figure.

```

1 class Figure {
2 public:
3     ...
4     vector<Coordinate> normais;
5     vector<TexCoord> texturas;
6     GLuint texture;
7     Material material;
8 }
```

4. FICHEIRO XML

Nesta fase, o ficheiro XML sofreu poucas mudanças na sua estrutura, tendo sido apenas adicionadas algumas novas tags e atributos.

Para começar, adicionamos logo no início, após a `<scene>` a tag das *lights*. Definimos apenas um ponto de luz, posicionado no centro do sistema solar (coordenadas (0,0,0)) e com componente difusa e ambiente.

Tag *lights*.

```
1  <lights>
2    <light type="POINT" X="0" Y="0" Z="0" diffR="0.8" diffG="0.8" diffB="0.8" ambR="0.4"
  ambG="0.4" ambB="0.4"/>
3  </lights>
```

Para além disto, dentro de cada *model*, adicionamos o atributo da sua textura e o seu material (facultativo). O atributo *file* faz referência ao ficheiro “.3d” e o atributo *texture* à imagem, enquanto que, neste caso abaixo, adicionamos componente emissiva, visto tratar-se do Sol.

Tag *models* para o Sol.

```
1  <models>
2    <model file="sol.3d" texture="sol.jpg" emiR="1" emiG="1" emiB="1"/>
3  </models>
```

No caso da Terra, visto não ser fonte de qualquer luz, não adicionamos nenhuma componente de textura de material.

Tag *models* para a Terra.

```
1  <models>
2    <model file="planetamed.3d" texture="terra.jpg"/>
3  </models>
```

Estas foram, de facto, as únicas alterações que efetuamos no ficheiro “scene.xml”.

5. CÂMARA DE VISUALIZAÇÃO

A câmara não sofreu alterações relativamente à última etapa, no entanto deixamos aqui os comandos existentes como auxiliar de uso à aplicação desenvolvida.

As teclas registadas para os movimentos e alterações foram as seguintes:

'w' e 'W' para movimentar a câmara para a frente.
's' e 'S' para movimentar a câmara para trás.
'a' e 'A' para movimentar a câmara para a esquerda.
'd' e 'D' para movimentar a câmara para a direita.
'q' e 'Q' para movimentar a câmara para baixo.
'e' e 'E' para movimentar a câmara para cima.

'8' para, mantendo o mesmo centro de visualização, mover a câmara para cima.
'2' para, mantendo o mesmo centro de visualização, mover a câmara para baixo.
'4' para, mantendo o mesmo centro de visualização, mover a câmara para a esquerda.
'6' para, mantendo o mesmo centro de visualização, mover a câmara para a direita.

'LEFT_BUTTON' e 'KEY_UP' para aproximar a câmara do centro.
'RIGHT_BUTTON' e 'KEY_DOWN' para afastar a câmara do centro.

'i' e 'I' para mudar o modo de visualização do modelo para *GL_FILL*.
'o' e 'O' para mudar o modo de visualização do modelo para *GL_LINE*.
'p' e 'P' para mudar o modo de visualização do modelo para *GL_POINT*.

Foram atribuídas pares de teclas de movimentação da câmara para acomodar a rapidez com que queremos percorrer o modelo, sendo que a segunda tecla atribuída nestes pares resulta numa maior alteração nas variáveis correspondentes.

6. EXTRAS

Adicionamos um pequeno extra ainda. Como “background” do sistema solar, criamos o cosmos. Para tal, criamos uma esfera virada ao “avesso” no *Generator* e definimo-la no XML com um tamanho muito grande e com uma textura, tal como definimos para qualquer outra esfera, como o Sol, por exemplo.

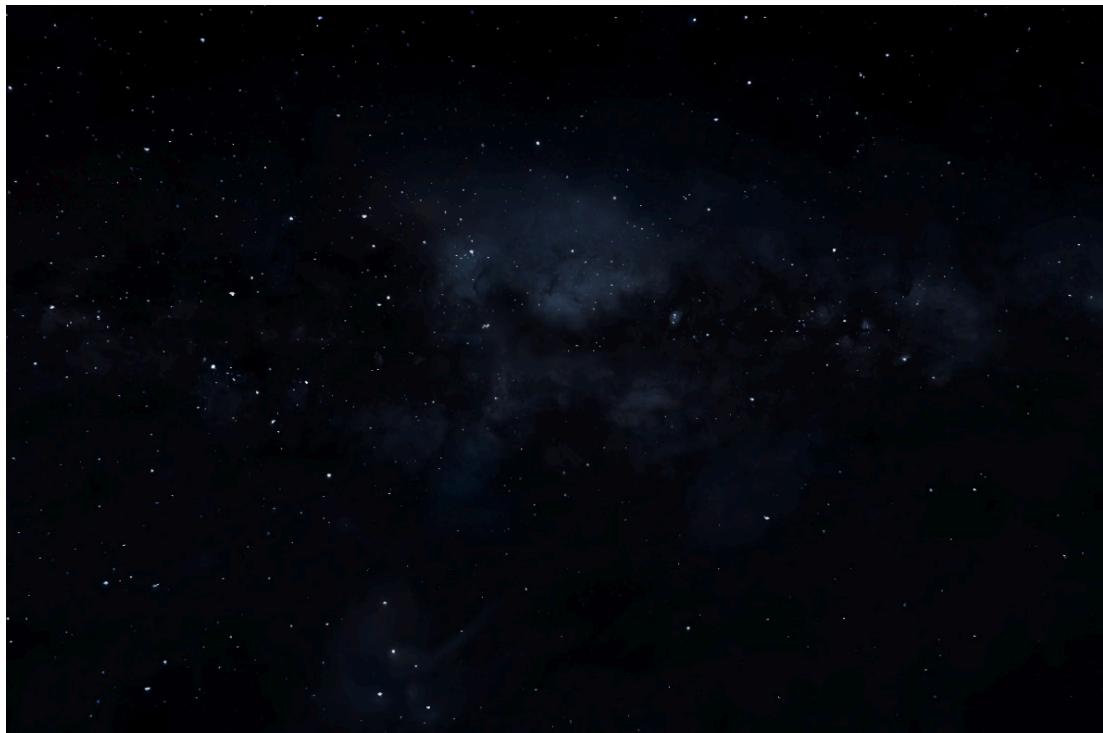


Figura 3 O cosmos.

7. RESULTADO FINAL

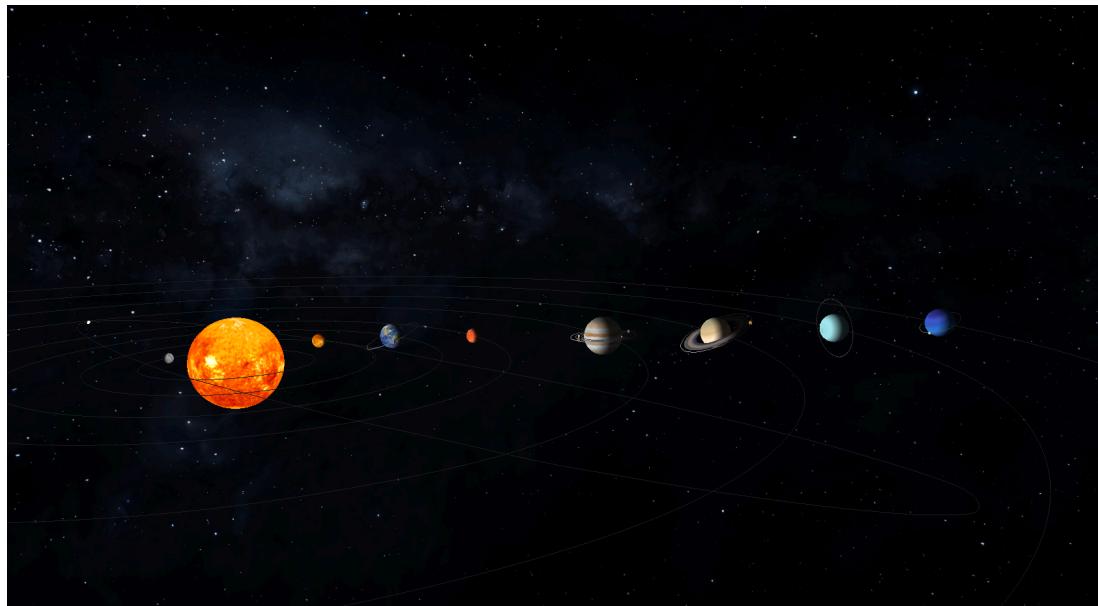


Figura 4 Resultado final do Sistema Solar.

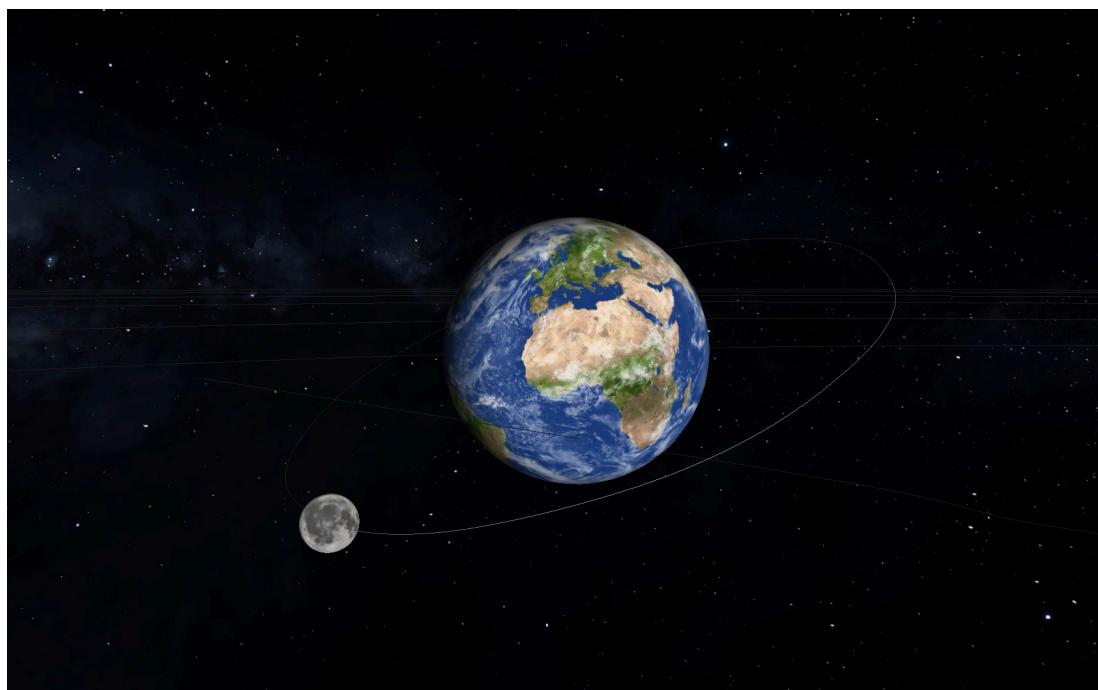


Figura 5 Terra e a Lua.

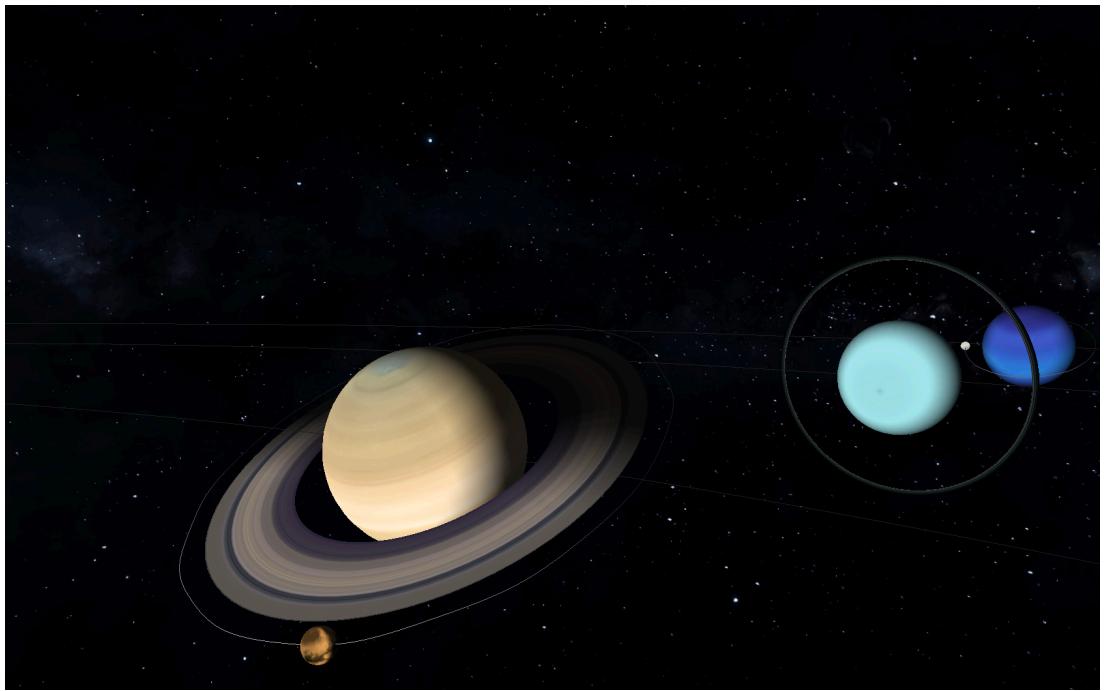


Figura 6 Anéis de Saturno e Úrano e Titã, lua de Saturno.

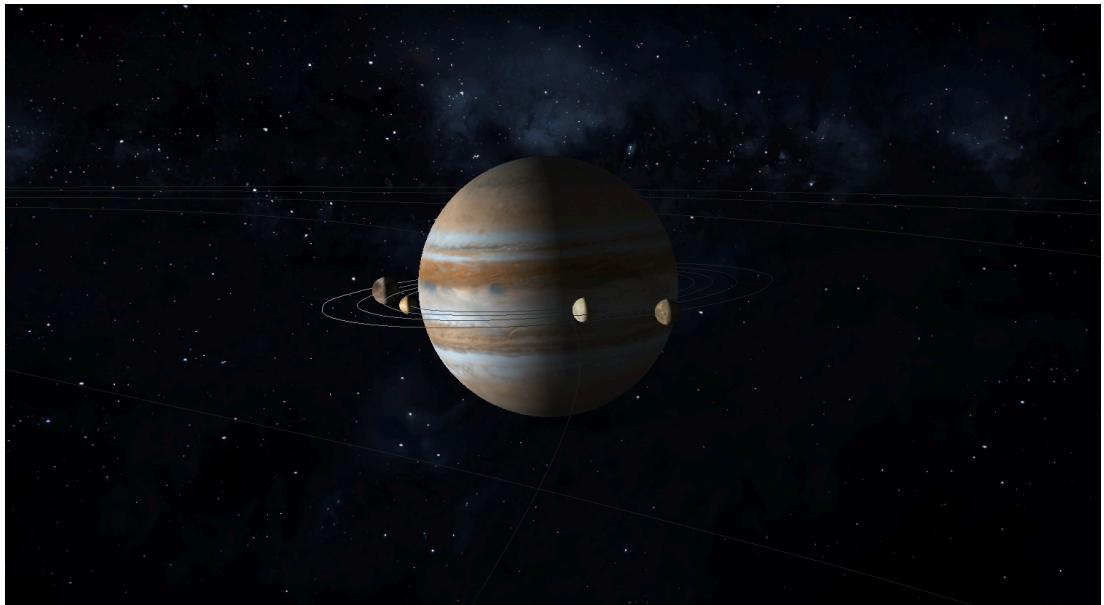


Figura 7 Io, Europa, Ganimedes e Calisto, luas de Júpiter.

8. CONCLUSÕES FINAIS

Esta última fase permitiu concluir por fim este trabalho prático.

Falando especificamente desta última etapa, foi bastante enriquecedora, pois permitiu explorar novos conhecimentos relativos ao *Lighting* e à aplicação de texturas a objetos. Estas duas funcionalidades são bastante importantes dentro do mundo da computação gráfica, por isso concluímos que foi bastante importante absorver conhecimento relativamente a estes tópicos.

Relativamente à aprendizagem obtida no trabalho prático como um todo, concluímos que desempenhou um papel bastante importante na aprendizagem e contacto direto com a programação direcionada à computação gráfica.

Falando mais especificamente do trabalho prático, concluímos que está num estado final bastante satisfatório, porém podíamos ter implementado alguns extras que certamente saberíamos como implementar e que gostaríamos como o fazer. Extras como o frustum culling que melhorariam o desempenho do programa seriam bastantes úteis, porém não os implementamos dada a falta de tempo. Apesar deste trabalho prático estar concluído, tal não significa que não possamos futuramente implementar novas funcionalidades por diversão e certamente aprender novas coisas relacionadas com a computação gráfica.