

UNIVERSIDADE DO MINHO

ESCOLA DE ENGENHARIA



Computação Gráfica

Licenciatura em Engenharia Informática

Fase 4 - Normals and Texture Coordinates

Grupo 12

Ana Pires - [A96060]
Mariana Marques - [A93198]

Junho, 2023

Conteúdo

1	Introdução	3
1.1	Contextualização	3
1.2	Objetivos 4. ^a Fase	3
2	Generator	4
2.1	Normais	4
2.1.1	<i>Plano</i>	4
2.1.2	<i>Caixa</i>	4
2.1.3	<i>Esfera</i>	5
2.1.4	<i>Cone</i>	5
2.1.5	<i>Bezier</i>	5
2.1.6	<i>Torus</i>	7
2.2	Texturas	7
2.2.1	<i>Plano</i>	7
2.2.2	<i>Caixa</i>	7
2.2.3	<i>Esfera</i>	7
2.2.4	<i>Cone</i>	8
2.2.5	<i>Bezier</i>	8
2.2.6	<i>Torus</i>	9
3	Engine	10
3.1	Leitura ficheiros <i>.3d</i>	10
3.2	Luzes e Cor	10
3.3	VBOs	11
4	Extraordinário	13

4.1	Torus	13
4.2	SkyBox	13
5	Testes	14
6	Sistema Solar	16
7	Conclusão	18

1. Introdução

1.1 Contextualização

O presente relatório tem como objetivo documentar e detalhar o desenvolvimento da quarta fase do projeto da unidade curricular de Computação Gráfica.

De um modo geral, esta fase propõe a implementação de texturas e iluminação, permitindo uma representação mais realista da cena. Para tal, recorreu-se a cálculos de normais e coordenadas de textura.

1.2 Objetivos 4.^a Fase

No final desta fase do trabalho, é esperado obter-se a representação do sistema solar animado com texturas e iluminação. Como apresentado no enunciado do projeto, para tal ser possível é necessário conquistar alguns objetivos, tais como:

1. Ativar, no *engine*, as funcionalidades de iluminação e textura;
2. Atualizar o *generator*, de modo a que gere coordenadas de textura e normais para cada vértice;

2. Generator

De modo a cumprir os objetivos propostos, foi necessário realizar alterações no *Generator* relativamente às fases precedentes, nomeadamente:

- **Adicionar Normais** - Cada primitiva exige coordenadas que representam as normais, de forma a obter uma correta representação da iluminação.
- **Adicionar Texturas** - Cada primitiva exige coordenadas que representam as texturas, de forma a permitir uma representação mais realista de cada uma.
- **Escrita no ficheiro** - Com as coordenadas das normais e texturas geradas, foi necessário alterar a função responsável pela escrita no ficheiro *.3d*.

2.1 Normais

2.1.1 *Plano*

Em relação a esta primitiva, apenas foi necessário gerar as normais para a face superior e inferior.

- **Face Superior:** $(0, 1, 0)$
- **Face Inferior:** $(0, -1, 0)$

2.1.2 *Caixa*

A *Caixa* é uma figura composta por seis planos, sendo que cada um deles representa uma face. Dessa forma, aplica-se as normais do plano a cada face da primitiva, ou seja:

- **Face Topo:** $(0, 1, 0)$
- **Face Base:** $(0, -1, 0)$
- **Face Frontal:** $(0, 0, 1)$
- **Face Traseira:** $(0, 0, -1)$
- **Face Direita:** $(1, 0, 0)$
- **Face Esquerda:** $(-1, 0, 0)$

2.1.3 Esfera

Em relação à *Esfera*, considera-se o cálculos dos pontos realizados anteriormente e ajusta-se o tamanho do raio da esfera para valor 1, o que permite o cálculo correto das normais em cada ponto.

2.1.4 Cone

Em relação ao *Cone*, considera-se duas partes distintas da primitiva - Base e Lados, utilizando-se métodos de cálculos diferentes em ambas.

- **Base** - Em relação à base, considera-se que representa a normal $(0, -1, 0)$, uma vez que diz respeito a um plano com a sua direção para baixo.
- **Lados** - Em relação aos lados, considera-se o cálculo das normais através do produto vetorial e, posterior, normalização para cada vértice.

2.1.5 Bezier

Em relação à *superfície de Bezier*, o cálculo das normais é semelhante ao cálculo realizado na primitiva *Cone*, porém, neste caso, em vez de se calcular os vetores entre os pontos adjacentes, é calculada a derivada em relação aos parâmetros u e v .

Para isto ser possível, a função *getBezierPoint* é alterada para também suportar o cálculo de derivadas, onde o parâmetro o u representa o caso 1 e o parâmetro v representa o caso 2.

```
Point Bezier::getBezierPoint(int p, float u, float v, int flag) {
    (...)

    float bu[4], bv[4];
    switch (flag)
    {
        case 0:
            bu[0] = powf(u,3);
            bu[1] = powf(u,2);
            bu[2] = u;
            bu[3] = 1;

            bv[0] = powf(v,3);
            bv[1] = powf(v,2);
            bv[2] = v;
            bv[3] = 1;
            break;

        case 1:
            bu[0] = 3*powf(u,2);
            bu[1] = 2*u;
```

```
bu[2] = 1;  
bu[3] = 0;  
  
bv[0] = powf(v,3);  
bv[1] = powf(v,2);  
bv[2] = v;  
bv[3] = 1;  
break;  
  
case 2:  
  
bu[0] = powf(u,3);  
bu[1] = powf(u,2);  
bu[2] = u;  
bu[3] = 1;  
  
bv[0] = 3*powf(v,2);  
bv[1] = 2*v;  
bv[2] = 1;  
bv[3] = 0;  
break;  
  
default:  
    break;  
}  
  
(...)
```

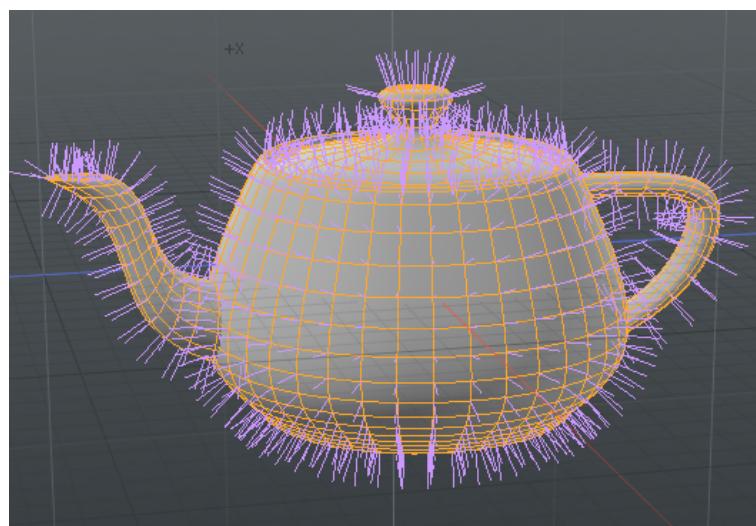


Figura 2.1: Normais à Superfície de Bezier

2.1.6 Torus

Em relação ao *Torus*, opta-se por simplificar o cálculo das normais.

Esta primitiva é extraordinária, na medida em que, apenas foi implementada para representar o Anel de Saturno. Assim, com o número de *slices* igual a 2, é idêntico à primitiva *Plane*.

Por fim, é atribuída a normal $(0, 1, 0)$ às faces em questão.

2.2 Texturas

Para ser possível o cálculo das coordenadas de textura, considera-se para qualquer imagem que a altura e o comprimento da mesma está entre 0 e 1. Desta forma, é apenas necessário saber quantas divisões é necessário realizar, tanto na horizontal como na vertical.

2.2.1 Plano

A estratégia para o cálculo das coordenadas de textura do *Plano* centra-se no cálculo do número de divisões que é necessário realizar, que é dado por: $\text{div} = \frac{1.0f}{grid}$.

Desta forma, para cada ponto calcula-se a coordenada de textura correspondente, ou seja:

```
Point2D t1 = Point2D(i * div, (j+1) * div), // (Ponto p1)
t2 = Point2D(i * div, j * div), // (Ponto p2)
t3 = Point2D((i+1) * div, j * div), // (Ponto p3)
t4 = Point2D((i+1) * div, (j+1) * div); // (Ponto p4)
```

2.2.2 Caixa

A estratégia para realizar o cálculo das coordenadas de textura da *Caixa* é semelhante ao da primitiva anterior, visto que, é composta por 6 planos.

É de salientar que as faces da Esquerda e Direita sofreram uma rotação de coordenadas, para o resultado do *test_4-6.xml* unificar com o resultado esperado.

2.2.3 Esfera

A estratégia para realizar o cálculo das coordenadas de textura da *Esfera* centra-se no cálculo de divisões a ser feitas na vertical e horizontal, dado por: $\text{div_slices} = \frac{1.0f}{slices}$ e $\text{div_stacks} = \frac{1.0f}{stacks}$.

Desta forma, para cada ponto calcula-se a coordenada de textura correspondente, ou seja:

```
Point2D t1 = Point2D(1 - i * div_slices, 1 - (j+1) * div_stacks), // (Ponto p1)
t2 = Point2D(1 - (i+1) * div_slices, 1 - (j+1) * div_stacks), // (Ponto p2)
t3 = Point2D(1 - (i+1) * div_slices, 1 - j * div_stacks), // (Ponto p3)
t4 = Point2D(1 - i * div_slices, 1 - j * div_stacks); // (Ponto p4)
```

2.2.4 Cone

A estratégia para realizar o cálculo das coordenadas de textura do *Cone* divide-se em duas vertentes - Base e Lados.

Em relação à base, é definido um ponto central da imagem, dado por $i + 0.5$, visto que, os lados se encontram no mesmo ponto.

```
addTexture(Point2D((i+1) * div_slices, 0));
addTexture(Point2D(i * div_slices, 0));
addTexture(Point2D((i + 0.5) * div_slices, 1));
```

Em relação aos lados, a estratégia centra-se no cálculo de divisões a ser feitas na vertical e horizontal, dado por: $\text{div_slices} = \frac{1.0f}{slices}$ e $\text{div_stacks} = \frac{1.0f}{stacks}$.

Desta forma, para cada ponto calcula-se a coordenada de textura correspondente, ou seja:

```
Point2D t1 = Point2D(i * div_slices, (j + 1) * div_stacks), // (Ponto p1)
t2 = Point2D((i + 1) * div_slices, (j + 1) * div_stacks), // (Ponto p2)
t3 = Point2D((i + 1) * div_slices, j * div_stacks), // (Ponto p3)
t4 = Point2D(i * div_slices, j * div_stacks); // (Ponto p4)
```

2.2.5 Bezier

A estratégia para realizar o cálculo das coordenadas de textura da *superfície de Bezier* centra-se no cálculo de divisões a ser feitas na vertical e horizontal, dado por: $\frac{1.0f}{tesellation}$.

Desta forma, para cada ponto calcula-se a coordenada de textura correspondente, ou seja:

```
Point2D t1 = Point2D(u + (1.0f/tessellation), v + (1.0f/tessellation)), // (Ponto
p1)
t2 = Point2D(u, v + (1.0f/tessellation)), // (Ponto p2)
t3 = Point2D(u, v), // (Ponto p3)
t4 = Point2D(u + (1.0f/tessellation), v); // (Ponto p4)
```

2.2.6 *Torus*

A estratégia para realizar o cálculo das coordenadas de textura do *Torus* centra-se no cálculo de divisões a ser feitas na vertical e horizontal, dado por: $\text{div_slices} = \frac{1.0f}{\text{slices}}$ e $\text{div_stacks} = \frac{1.0f}{\text{stacks}}$.

Desta forma, para cada ponto calcula-se a coordenada de textura correspondente, ou seja:

```
Point2D t1 = Point2D(slice * div_slices, stack * div_stacks), // (Ponto p1)
t2 = Point2D((slice + 1) * div_slices, stack * div_stacks), // (Ponto p2)
t3 = Point2D((slice + 1) * div_slices, (stack + 1) * div_stacks), // (Ponto
    p3)
t4 = Point2D(slice * div_slices, (stack+1) * div_stacks); // (Ponto p4)
```

3. Engine

De modo a cumprir os objetivos propostos, foi necessário realizar alterações no *Generator* relativamente ás fases precedentes, nomeadamente:

- **Leitura dos ficheiros .3d** - Com a implementação das normais e texturas, foi necessário redefinir as funções correspondentes à leitura dos ficheiros .3d.
- **Estrutura de Dados Light e Color** - Nesta fase, foi necessário adicionar a estrutura de dados *Light* e *Color* para a correta implementação da luz.
- **VBOs** - Com a implementação das coordenadas das normais e textura no *Generator*, é necessário que o *Engine* mapeie corretamente cada uma delas para o respetivo modelo.

3.1 Leitura ficheiros .3d

Para a correta leitura das coordenadas de textura, foi criada uma nova estrutura de dados, *Point2D*. Desta forma, por adição à função *read_3DPoint*, é implementada a função *read_2DPoint*.

3.2 Luzes e Cor

Para as luzes, foi criada uma nova classe *Light*, que guarda os atributos para as luzes.

```
class Light {
    private:
        int id;
};

class Lights {
    private:
        vector<Light*> lights;
};

```

Estas poderão ser de três tipos: *Point*, *Directional* ou *Spotlight*.

No caso de uma luz posicional, é necessário atribuir a posição de onde partirá a mesma.

```
class PointLight : public Light {
    private:
        Point position;
```

No caso de uma luz direcional, é necessário atribuir apenas a direção que a mesma guiará.

```
class DirectionalLight : public Light {
    private:
        Point direction;
}
```

No caso das *Spotlight*, para além da posição, já é necessário atribuir uma direção de onde partirá a mesma.

```
class SpotLight : public Light {
    private:
        Point position, direction;
        float cutoff;
}
```

Para as cores, criou-se a classe *Color*, admitindo o tipo difusa, ambiente, especular e emissiva.

```
class Color {
    private:
        Point diffuse, ambient, specular, emissive;
        float shininess;
}
```

O que difere nestas duas estruturas de dados, é as chamadas das funções no OpenGL, visto que, a estrutura *Light* ativa as luzes e a estrutura *Color* ativa os materiais, através das funções *glLightfv* e *glMaterialfv*, respetivamente.

3.3 VBOs

Para automatizar a renderização, foi modificada a estrutura de dados responsável pelas VBOs, dada por *modelsVBOs*, de forma a incluir não só os vértices, mas também as normais e texturas.

map<string, unsigned int*>modelsVBOs = map<string, unsigned int*>();

Esta nova estrutura de dados respeita as regras da estrutura anterior, ou seja, só é criada uma *VBO* para um modelo que ainda não tenha sido lido, mas agora, guarda para cada um deles, os identificadores responsáveis pelos vértices, normais e texturas.

Por adição, foi criada uma estrutura de dados para as texturas, de forma a impedir que a mesma textura seja carregada mais que uma vez, dado por *mapTextures*.

```
map<string, unsigned int> mapTextures = map<string, unsigned int>();
```

4. Extraordinário

4.1 Torus

Ao longo do desenvolvimento deste trabalho, foi desenvolvida uma nova primitiva com o intuito de representar o Anel de Saturno, de forma a garantir uma aparência mais realista.

4.2 SkyBox

De modo a contribuir para uma melhor aparência do sistema solar desenvolvido, assim como tornar mais realista, decidiu-se implementar uma *SkyBox*, que desenha uma caixa com os triângulos invertidos (relativamente à regra da mão direita), oferecendo uma perspetiva interior.

5. Testes

Nesta secção, apresenta-se o resultado obtido nos testes relativos a esta etapa do projeto.

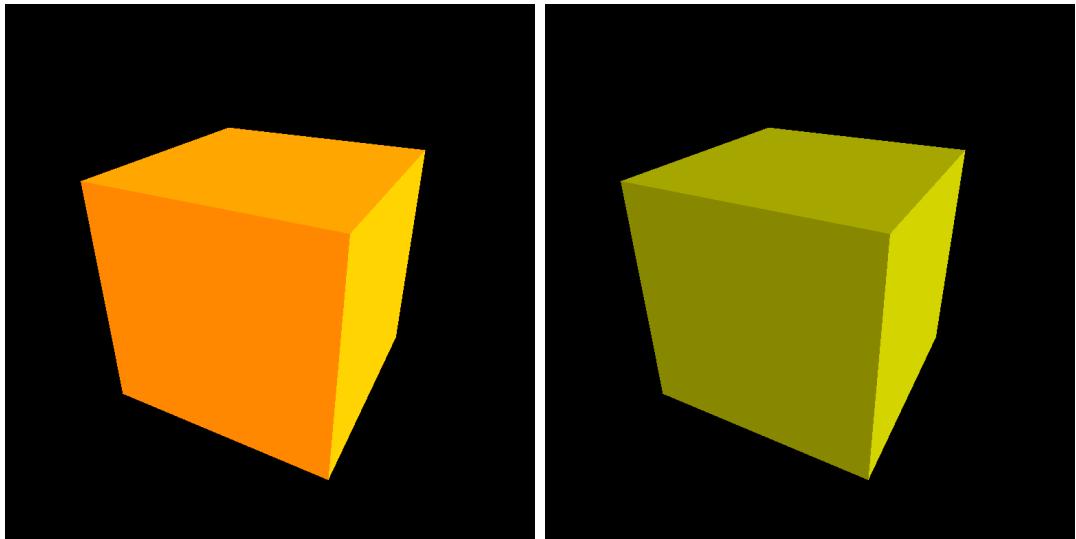


Figura 5.1: Resultados dos ficheiros de teste 1 e 2, respetivamente

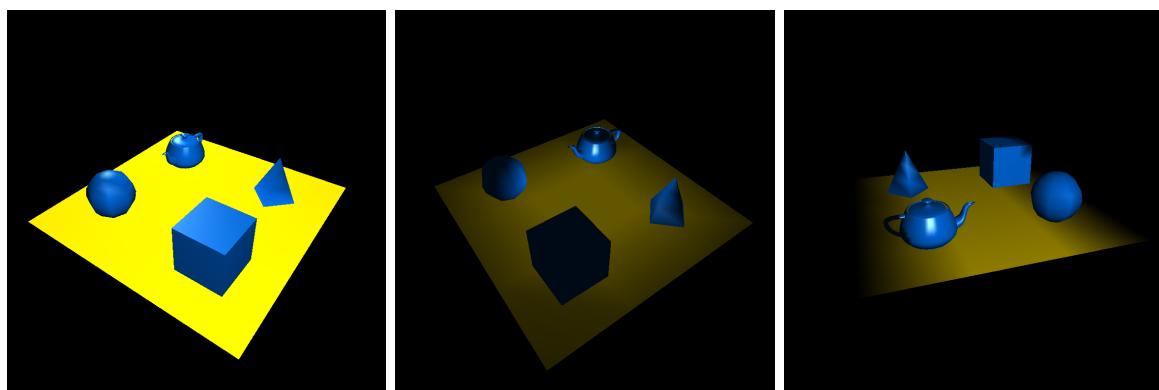


Figura 5.2: Resultados dos ficheiros de teste 3, 4 e 5, respetivamente

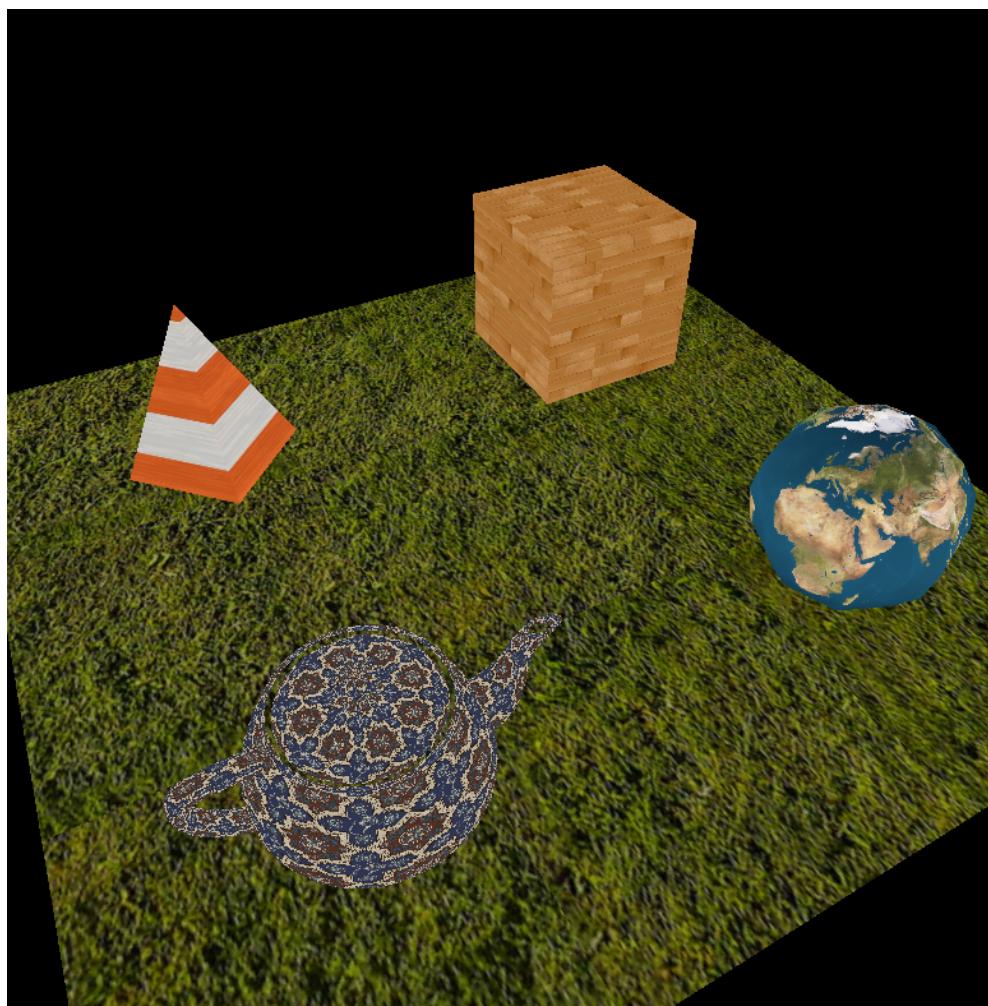
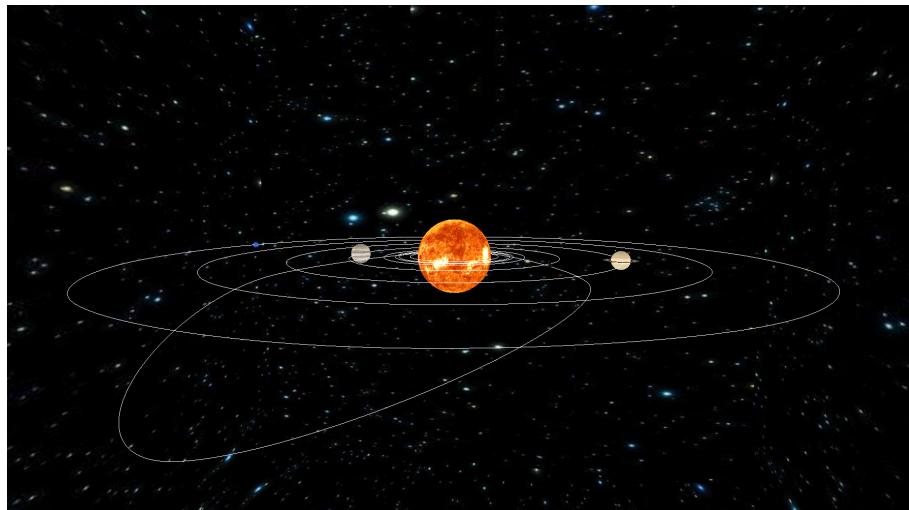
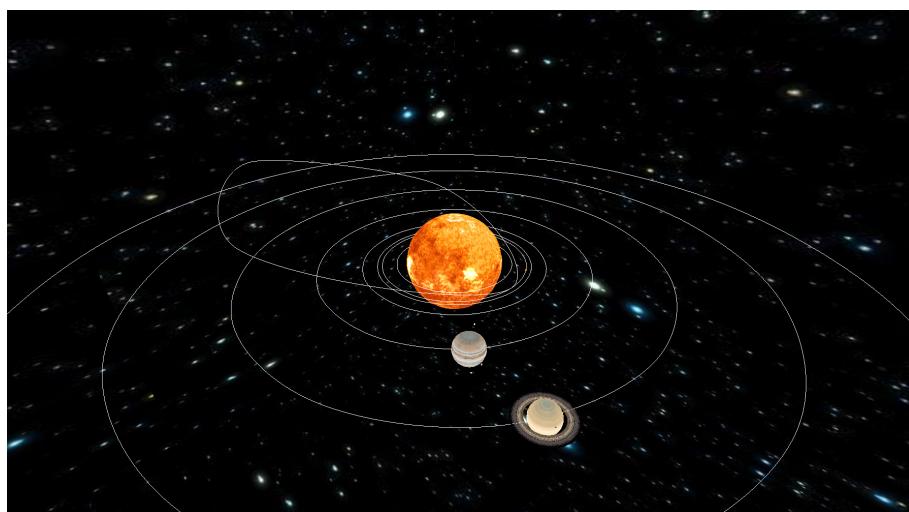
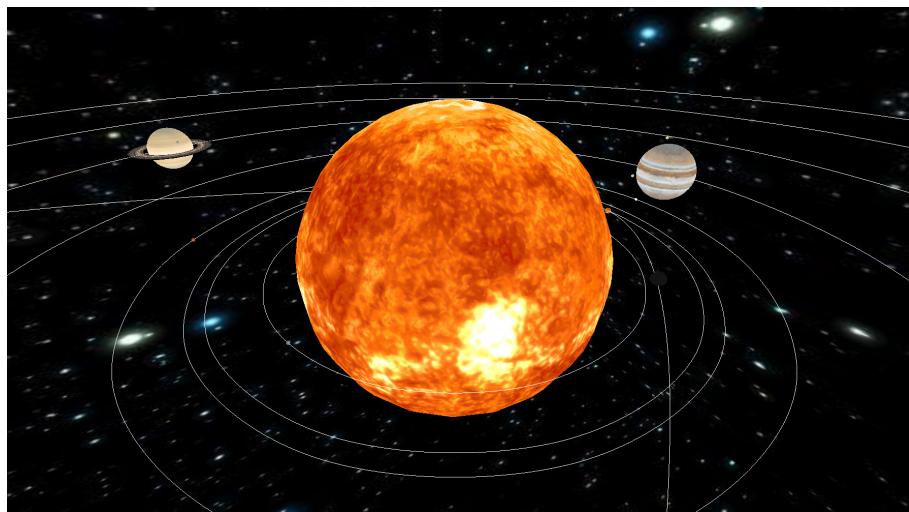


Figura 5.3: Resultado do ficheiro de teste 6

6. Sistema Solar

Após a implementação da iluminação e texturas, adicionou-se essas novas funcionalidades nas primitivas do Sistema Solar. Desta forma, obteve-se um cenário mais realista e apelativo, assemelhando-se mais ao real.





7. Conclusão

Esta foi, então, a fase final do trabalho prático. Em suma, conseguiu-se a implementação de texturas e iluminação com sucesso, tendo em conta os objetivos iniciais.

No geral, este projeto foi sem dúvida bastante enriquecedor, pois conseguiu-se colocar em prática todas as matérias lecionadas em aula. Apesar do aumento de complexidade e detalhe, este revelou-se bastante satisfatório e apelativo, uma vez que se conseguiu ver, no decorrer das etapas, uma evolução nos cenários, nomeadamente no Sistema Solar. A sua representação mostrou-se cada vez mais semelhante a uma representação real.

Como análise global, considera-se que ao longo do projeto, todos os objetivos foram bem conseguidos, tentando aplicar tudo o que foi lecionado. Para além disto, procurou-se sempre o melhor, superando todos os obstáculos e dificuldades.