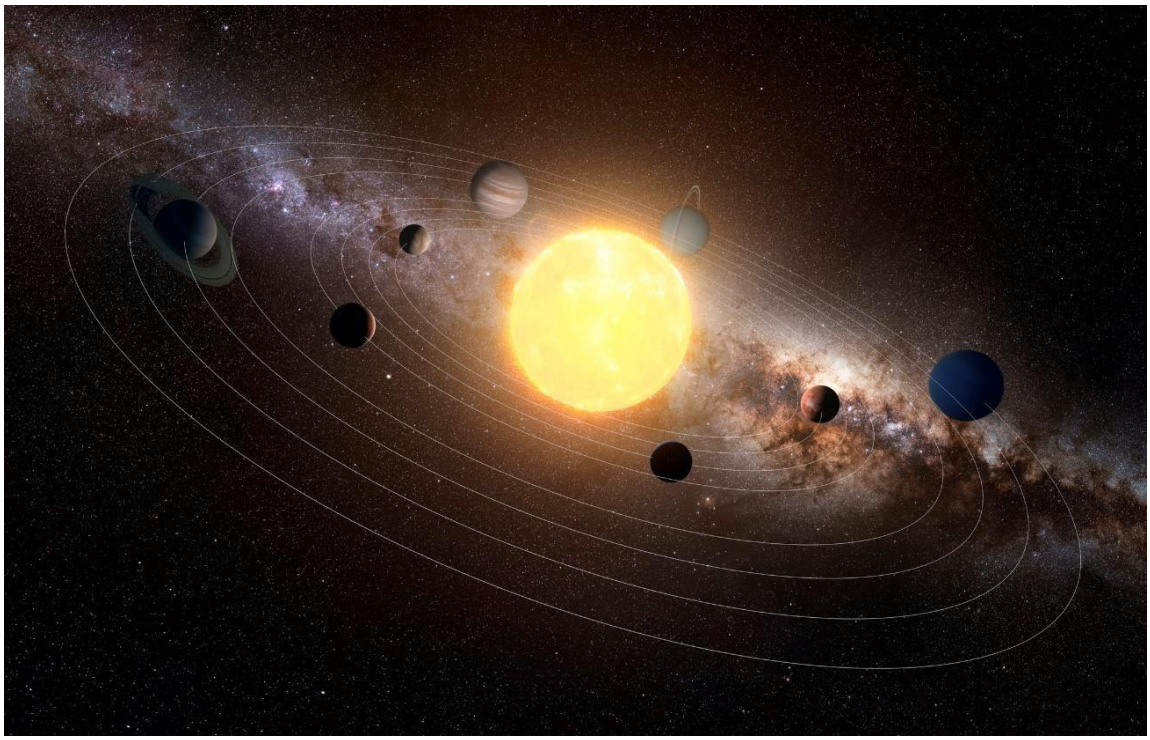
Relatório do 2º Trabalho – Realizado em WebGL

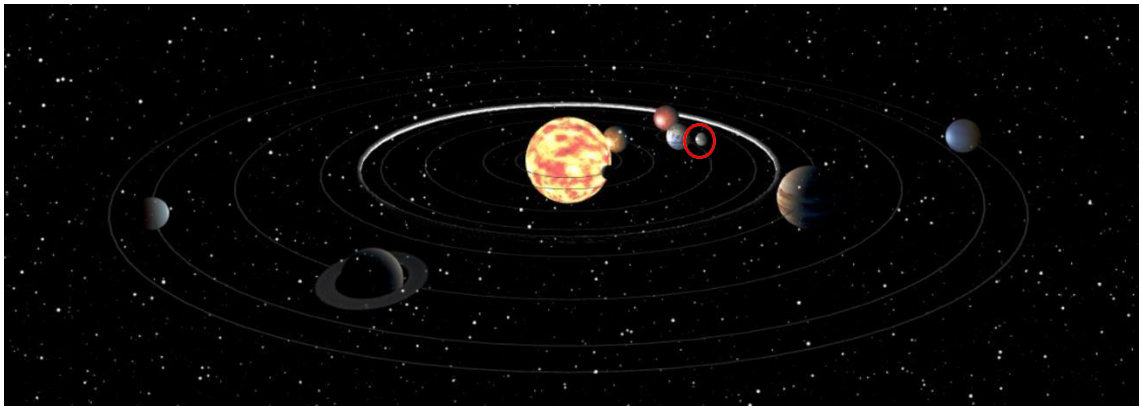
Computação Gráfica 2021/2022

Sistema Solar 3D



Alexandre Rodrigues nº 54472

Parte I – Implementação dos objetos 3D



A cena 3D acima representa o espaço com foco no sistema solar. São mostrados 1 estrela e 9 astros com as suas respetivas órbitas e cintura de asteroides:

- Os 8 planetas do sistema solar, ordenados de forma realista.

- A integração dos planetas no espaço mundo tem por base a posição do Sol.

- A lua, presente no círculo, junto à Terra.

- A presença de uma cintura de asteroides situada entre a órbita de Marte e Júpiter.

- A presença do disco de Saturno.

Parte II – Texturas

Como implementámos as texturas?

Para que as texturas sejam corretamente implementadas, alterámos o *Vertex Shader* (VS), o *Fragment Shader* (FS) e o Script WebGL.

Dentro do VS, adicionámos 2 atributos diferentes (**`attribute`** + **`varying`**). E, dentro do **`main()`** de VS, atribuímos o valor de **`attribute`** ao **`varying`**:

```
...
attribute vec2 texcoord;
varying vec2 v_texcoord;

void main() {
      ...
      v_texcoord = texcoord;
}
...
```

Dentro de FS, adicionámos 2 atributos diferentes (**`uniform`** + **`varying`**), sendo o primeiro aquele que representa o tipo de dados das texturas e o segundo aquele que foi adicionado no VS. Com isso, associámos ao output de FS (**`gl_FragColor`**) que vai mapear a textura interpolando o vetor das coordenadas da textura, através da função **`texture2D`**:

```
...
uniform sampler2D u_texture;
varying vec2 v_texcoord;

void main() {
      gl_FragColor = texture2D(u_texture, v_texcoord);
}
...
```

No Script WebGL, criámos a textura com a função **`createTexture`** (passando como argumento a imagem que servirá de textura) e atribuímos seu valor ao atributo **`uniform`** definido em FS:

```
...
sun.uniforms = {};
sun.uniforms.u_texture = twgl.createTexture(gl,
{src:'/text/SunTexture.png'});
...
```

Como cada astro tem uma textura diferente, repetimos o processo acima para cada um deles dentro do Script WebGL.

Por fim, para que as texturas possam aparecer no browser, lançámos o servidor (**`python3 -m http.server`**) dentro da pasta que contém o ficheiro HTML + as imagens que servirão de texturas e, no browser, procurámos **`localhost:8000/WebGLCG05.html`**.

Como as texturas são aplicadas aos objetos?

Cada textura é um vetor de cores, onde cada componente que lhe corresponde (denominada texel) contém um número correspondente ao de uma cor (de acordo com algum sistema, como o RGBA).

Depois, para cada pixel de um objeto, determina-se um conjunto de texels que serão úteis para modificar 1+ propriedades ópticas da superfície, de fora que o objeto inteiro seja “coberto” pela textura (normalmente guardadas no formato de imagens).

Parte III – Animação

Como implementámos as animações?

Para que os objetos possam girar em torno de si mesmos, criamos uma matriz que representa sua rotação em torno do eixo Y (na posição (0,0,0)) com a função **`rotationY`** (usando como argumento um valor numérico baseado no argumento **`time`** recebido por **`render`**) e atribuímos essa matriz ao atributo **`u_world`** de cada objeto.

Obs.: O atributo **`u_world`** será depois utilizado para calcular o atributo **`u_worldViewProjection`** (necessário para conhecer a posição em que o objeto estará na cena 3D) do objeto.

```
...
sun.uniforms.u_world = m4.rotationY(0.00036 * time);
...
```

Agora, para que os astros possam girar em torno da estrela, deixámos a estrela na posição (0,0,0) e, para cada astro:

- Criámos uma matriz que representa uma translação com a função **`translation`** (usando como argumento a distância do objeto em relação à estrela). Essa translação é feita de forma que o objeto fique ligeiramente afastado da origem (ou seja, da estrela), mas com atenção aos outros astros para que não haja colisão entre eles.

- Criámos uma matriz que representa uma rotação em torno do eixo Y com a função **`rotationY`** (usando como argumento um valor numérico baseado no argumento **`time`** recebido por **`render`**). Essa rotação, aplicada depois da translação, faz com que os astros se movimentem ao redor da estrela.

- Multiplicámos as duas matrizes criadas acima

- Multiplicámos a matriz resultante pela matriz definida anteriormente para a rotação dos objetos entre si

```
...
mercury.uniforms.u_world = m4.multiply(m4.rotationY(0.01 * time),
                           m4.multiply(m4.translation([0.75, 0, 0]),
                                       m4.rotationY(0.000169 * time)));
...
```

Em relação à lua, já que ela gira em torno da Terra, então, ao criar a matriz de translação, usámos como argumento de **`translation`** um valor que representa a distância dela em relação à Terra (não à estrela). Além disso, multiplicámos a sua matriz final pela matriz final da Terra para que a lua, apesar de não girar em volta da estrela, siga a Terra na sua órbita. A lua têm rotação em sentido retrógrado sobre si mesma e à volta da Terra.

```
...
moon.uniforms.u_world = m4.multiply(earth.uniforms.u_world,
                        m4.multiply(m4.rotationY(-0.001 * time),
                        m4.multiply(m4.translation([0.35, 0, 0]),
                                    m4.rotationY(-0.00036 * time))));
...
```

Obs.: Os valores usados ao definir a rotação das órbitas são proporcionalmente realistas.

Para definir o modo que vemos a animação, precisamos definir o tipo de projeção, assim como onde está a câmara e o seu foco. A projeção é definida com a ajuda da função **`perspective`**, que recebe como argumentos:

- O ângulo da câmara (de cima para baixo)

- A proporção da imagem (largura / altura)

- A profundidade do front clipping plane

- A profundidade do back clipping plane

```
...
const fov = 30 * Math.PI / 180;
const aspect = gl.canvas.clientWidth / gl.canvas.clientHeight;
const zNear = 0.5;
const zFar = 25;
const projection = m4.perspective(fov, aspect, zNear, zFar);
...
```

Já para definir a câmara e o seu foco, utilizámos a função **`lookAt`**, que recebe como argumentos:

- Onde a câmara se encontra (ou seja, sua posição)

- Para onde a câmara estará a olhar (no nosso caso, a posição da estrela)

- Um vetor apontado para cima

```
...
 const eye = [0, 5, -15];
 const target = [0, 0, 0];
 const up = [0, 1, 0];
 const camera = m4.lookAt(eye, target, up);
 const view = m4.inverse(camera);

 ...
```

Por fim, criámos a matriz **`viewProjection`**, que nos será útil para criar o atributo **`u_worldViewProjection`** posteriormente.

```
 ...
 const viewProjection = m4.multiply(projection, view);
 ...
```

Como funciona o processo das animações?

Para entender melhor o processo de animação, é útil conhecer o pipeline gráfico, que mostra a ordem em que cada matriz tem de ser criada e multiplicada com outra determinada matriz (de forma a ter uma animação).

O pipeline direciona para a GPU onde serão realizados cálculos complexos para não sobrecarregar o processador sendo que o pipeline é dividido no vertex shader, rasterizer e fragment shader que vai posteriormente ser renderizado para gerar a imagem. As várias mudanças dos valores das coordenadas ao longo do tempo vão permitir uma interpolação criando uma ilusão de animação do objeto.

Link para as animações

A animação do Sistema Solar encontra-se no link:

https://www.youtube.com/watch?v=UiygAP5Lvlk

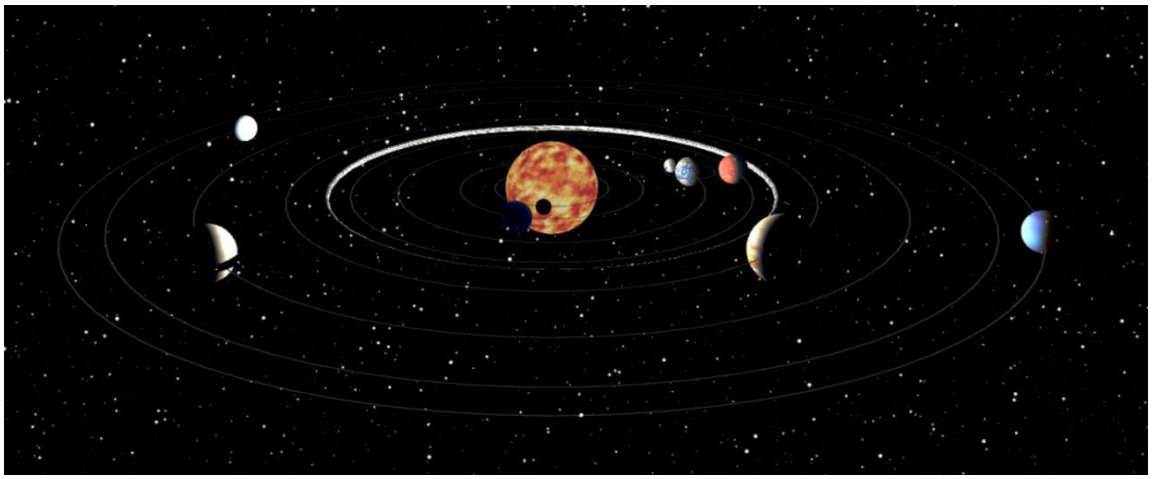
A animação lua/terra encontra-se no link:

https://www.youtube.com/watch?v=IPeyBJxDbPg

Parte IV – Iluminação

1.

Shading Phong com luz ambiente e difusa:



Implementação do vertex shader(VS):

```
<script id="vs" type="notjs">
    uniform mat4 u_worldViewProjection;

    attribute vec4 position;
    varying vec4 vcolor;
    attribute vec2 texcoord;
    varying vec2 v_texcoord;
    uniform vec3 lightPosition;
    uniform mat4 u_world;
    attribute vec3 normal;
    varying vec3 v_normal;
    uniform mat4 u_worldit;
    varying vec3 surfaceToLight;

    void main() {
      v_texcoord = texcoord;
      v_normal = mat3(u_worldit) * normal;
      vec4 vertexPos = u_world * position;
      surfaceToLight = lightPosition - vertexPos.xyz;
      gl_Position = u_worldViewProjection * position;
    }
```

Implementação do fragment shader(FS):

- Para o objeto sol o fragment shader é o mesmo sendo a única diferença a direção dos v_normal, tendo sido colocado negativo

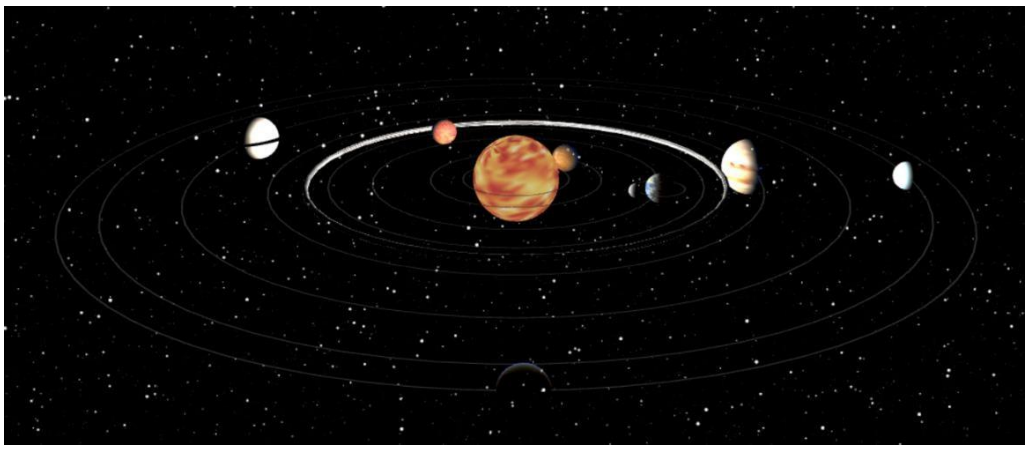
```
<script id="fs" type="notjs">
    precision highp float;
    varying vec2 v_texcoord;
    uniform sampler2D u_texture;
    varying vec3 surfaceToLight;
    varying vec3 v_normal;
    float ka;
    float kd;
    vec4 ambientColor;
    vec4 diffuseColor;
    float c1, c2, c3;

    void main() {
     vec3 normal = normalize(v_normal);
     vec3 surfaceToLightDirection = normalize(surfaceToLight);
     float intensidade = dot(v_normal, surfaceToLight);
     float distance = length(surfaceToLight);
     c1 = 0.05;
     c2 = 0.05;
     c3 = 0.05;
     float attenuation = min(1.0 / (c1 + c2 * distance + c3 * pow(distance, 2.0)),1.0);
     ambientColor = vec4(1.0, 1.0, 1.0, 1.0);
     diffuseColor = vec4(0.02, 0.02, 0.02, 1);
     ka = 0.2;
     kd = 0.8;
     gl_FragColor = ka * ambientColor + attenuation * intensidade * (kd * diffuseColor + texture2D(u_texture, v_texcoord));
    }
```

Em cada uniforms de um planeta dentro do JavaScript(JS):

```
surfaceToLightDirection = m4.inverse(mercurio.uniforms.u_world);
mercurio.uniforms.u_worldit = m4.transpose(surfaceToLightDirection);
```

Shading Gouraud com luz ambiente e difusa:



Implementação do vertex shader(VS):

- Para o objeto sol o vertex shader é o mesmo sendo a única diferença a direção dos v_normal, tendo sido colocado negativo

```
<script id="vs" type="notjs">
    uniform mat4 u_worldViewProjection;

    attribute vec4 position;
    varying vec4 vcolor;
    attribute vec2 texcoord;
    varying vec2 v_texcoord;
    uniform vec3 lightPosition;
    uniform mat4 u_world;
    attribute vec3 normal;
    varying vec3 v_normal;
    uniform mat4 u_worldit;
    varying vec3 surfaceToLight;
    float ka;
    float kd;
    vec4 ambientColor;
    vec4 diffuseColor;
    float c1, c2, c3;
    uniform sampler2D u_texture;

    void main() {
      v_texcoord = texcoord;
      v_normal = mat3(u_worldit) * normal;
      vec4 vertexPos = u_world * position;
      surfaceToLight = lightPosition - vertexPos.xyz;
      vec3 normal = normalize(v_normal);
      vec3 surfaceToLightDirection = normalize(surfaceToLight);
      float intensidade = dot(v_normal, surfaceToLight);
      float distance = length(surfaceToLight);
      c1 = 0.05;
      c2 = 0.05;
      c3 = 0.05;
      float attenuation = min(1.0 / (c1 + c2 * distance + c3 * pow(distance, 2.0)),1.0);
      ambientColor = vec4(1.0, 1.0, 1.0, 1.0);
      diffuseColor = vec4(0.02, 0.02, 0.02, 1);
      ka = 0.2;
      kd = 0.8;
      gl_Position = u_worldViewProjection * position;
      vcolor =  ka * ambientColor + attenuation * intensidade * (kd * diffuseColor + texture2D(u_texture, v_texcoord));
    }
```

Implementação do fragment shader(FS):

```
<script id="fs" type="notjs">
    precision highp float;
    varying vec2 v_texcoord;
    uniform sampler2D u_texture;
    varying vec4 vcolor;

    void main() {
     gl_FragColor = vcolor;
    }
```

Em cada uniforms de um planeta dentro do JavaScript(JS):

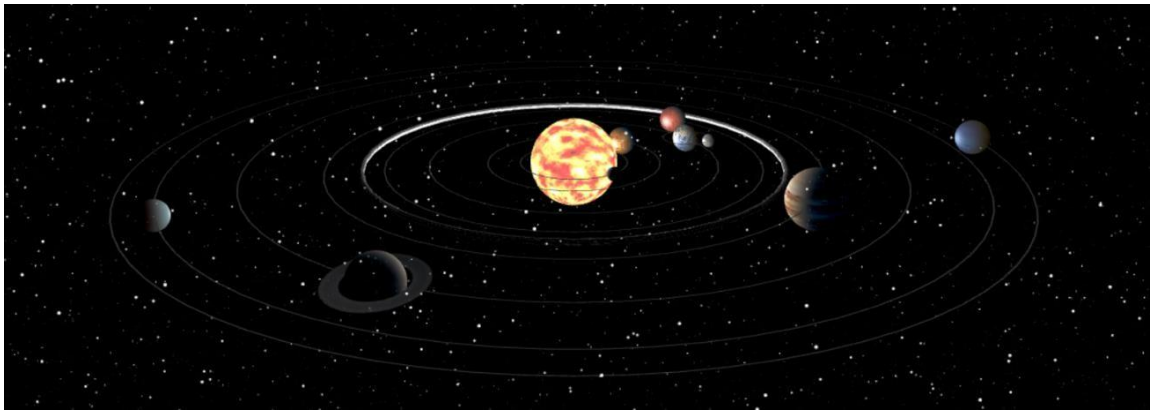
```
surfaceToLightDirection = m4.inverse(mercurio.uniforms.u_world);
mercurio.uniforms.u_worldit = m4.transpose(surfaceToLightDirection);
```

Comparação do resultado:

Como se pode observar pelas imagens, a aplicação dos métodos de Shading Phong e Shading Gouraud com incidência de apenas as componentes de luz ambiente e difusa apresentam um aspeto muito semelhante. Isto porque em ambos os casos a única variável é a posição da normal à superfície do objeto com o vetor da direção da luz que como é igual em ambas simulações não se notam mudanças significativas.

2.

Shading Phong com luz ambiente, difusa e especular:



Implementação do vertex shader(VS):

```
<script id="vs" type="notjs">
    uniform mat4 u_worldViewProjection;

    attribute vec4 position;
    varying vec4 vcolor;
    attribute vec2 texcoord;
    varying vec2 v_texcoord;
    uniform vec3 lightPosition;
    uniform vec3 u_viewWorldPosition;
    uniform mat4 u_world;
    attribute vec3 normal;
    varying vec3 v_normal;
    uniform mat4 u_worldit;
    varying vec3 surfaceToLight;
    varying vec3 surfaceToView;

    void main() {
      v_texcoord = texcoord;
      v_normal = mat3(u_worldit) * normal;
      vec3 surfaceWorldPosition = (u_world * position).xyz;
      surfaceToLight = lightPosition - surfaceWorldPosition;
      surfaceToView = u_viewWorldPosition - surfaceWorldPosition;
      gl_Position = u_worldViewProjection * position;
    }
```

Implementação do fragment shader(FS):

- Para o objeto sol o fragment shader é o mesmo sendo a única diferença a direção dos v_normal, tendo sido colocado negativo

```
<script id="fs" type="notjs">
   precision highp float;
   varying vec2 v_texcoord;
   uniform sampler2D u_texture;
   varying vec3 surfaceToLight;
   varying vec3 v_normal;
   varying vec3 surfaceToView;
   float u_shininess;
   float ka;
   float kd;
   float ks;
   vec4 ambientColor;
   vec4 diffuseColor;
   float c1, c2, c3;

   void main() {
    vec3 normal = normalize(v_normal);
    vec3 surfaceToLightDirection = normalize(surfaceToLight);
    vec3 surfaceToViewDirection = normalize(surfaceToView);
    vec3 halfVector = normalize(surfaceToLightDirection + surfaceToViewDirection);
    float intensidade = dot(normal, surfaceToLightDirection);
    float distance = length(surfaceToLight);
    c1 = 0.05;
    c2 = 0.05;
    c3 = 0.05;
    float attenuation = min(1.0 / (c1 + c2 * distance + c3 * pow(distance, 2.0)),1.0);
    ambientColor = vec4(1.0, 1.0, 1.0, 1.0);
    diffuseColor = vec4(0.02, 0.02, 0.02, 1);
    ka = 0.2;
    kd = 0.1;
    ks = 0.7;
    float specular = 0.0;
    float u_shininess = 10.0;
    if (intensidade > 0.0) {
        specular = pow(dot(normal, halfVector), u_shininess);
    }
    gl_FragColor = ka * ambientColor + attenuation * intensidade * (kd * diffuseColor + ks * specular + texture2D(u_texture, v_texcoord));
   }
```

Em cada uniforms de um planeta dentro do JavaScript(JS):

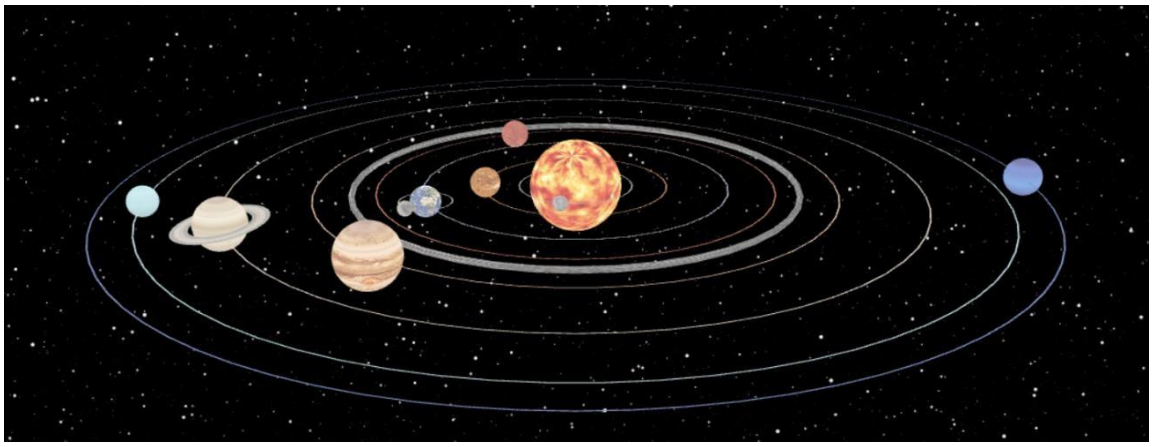
```
surfaceToLightDirection = m4.inverse(mercurio.uniforms.u_world);
mercurio.uniforms.u_worldit = m4.transpose(surfaceToLightDirection);
```

Comparação com o resultado obtido na alínea anterior:

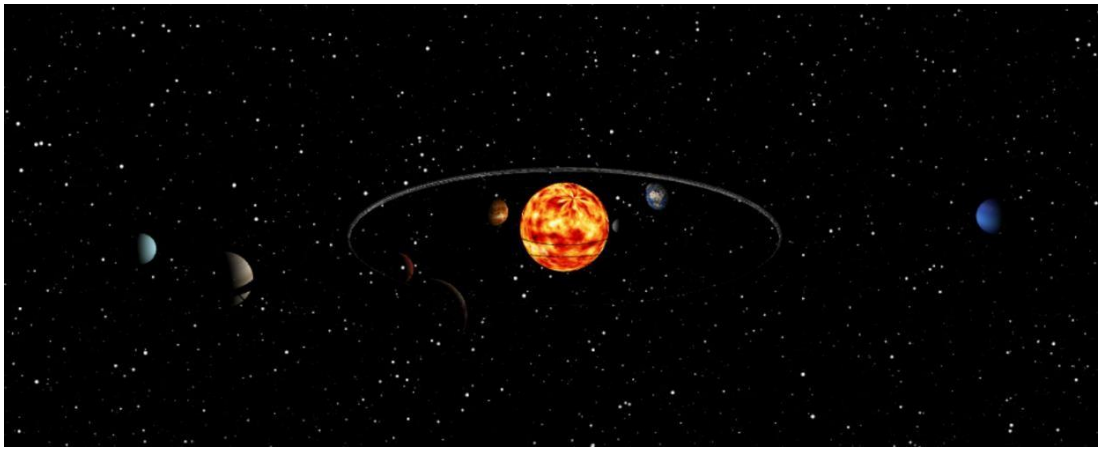
Como se pode observar pelas imagens, a aplicação do método de Shading Phong com as componentes de luz do item anterior junto com a incidência de uma componente de luz especular faz com que surjam highlights nos objetos que têm em consideração o eye (posição da câmara) no cálculo da sua intensidade.

3.

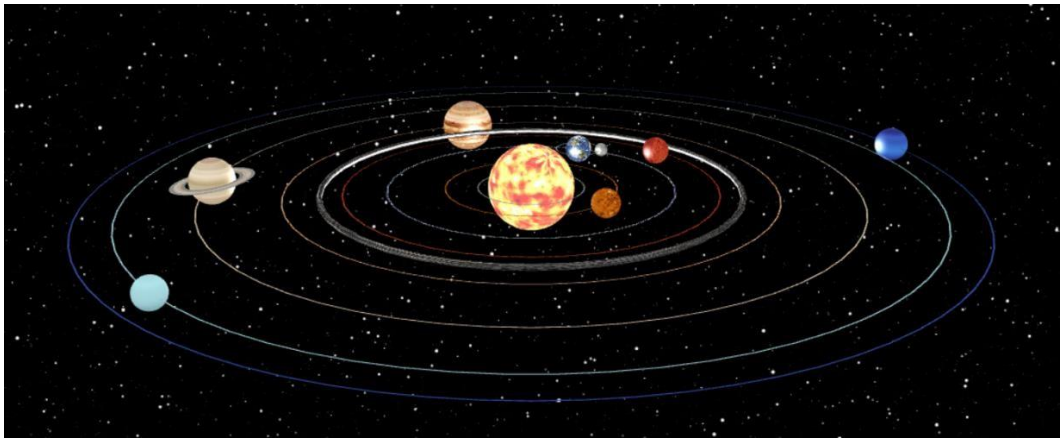
Luz ambiente:



Luz difusa:

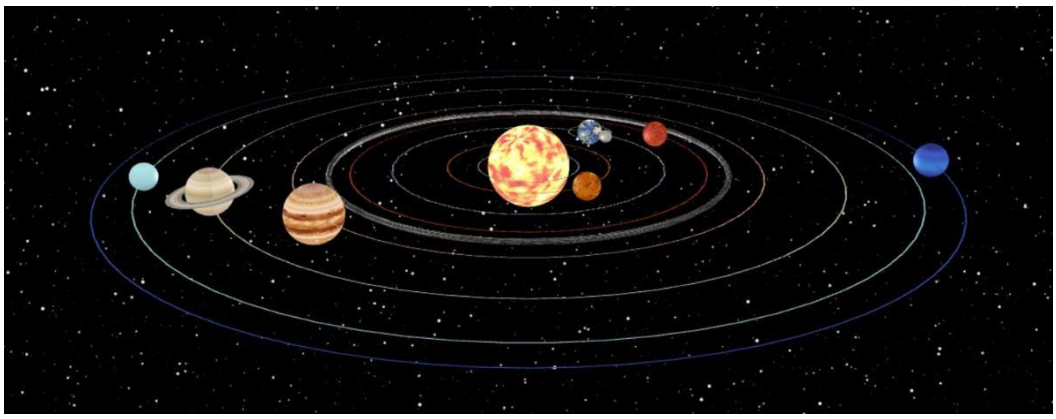


Luz especular:

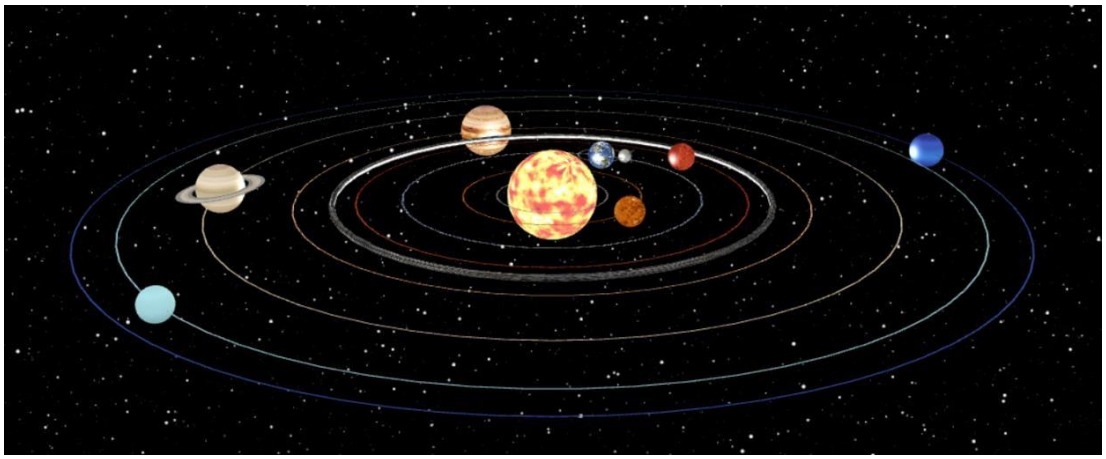


4.

Com ks = 0.3



Com ks = 1.0



Comparação entre as duas imagens, tendo em conta o coeficiente de especularidade:

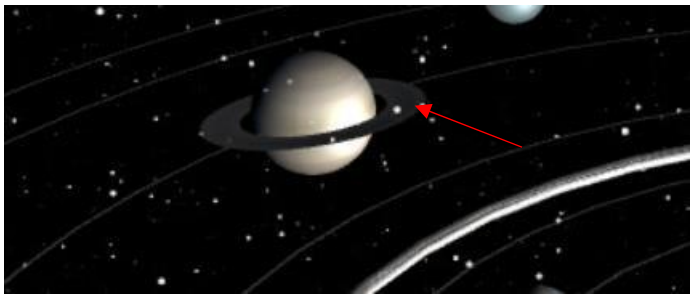
Como é possível observar pelas imagens, aplicando um coeficiente de especularidade menor (0.3) nos objetos, os highlights presentes nesses objetos aparecem pouco brilhantes com reflexão baixa, enquanto que, aplicando um coeficiente de especularidade maior (1.0), os highlights aparecem muito brilhantes com reflexão alta.

Parte V – Extras

1.

Anel de saturno:

Imagem:

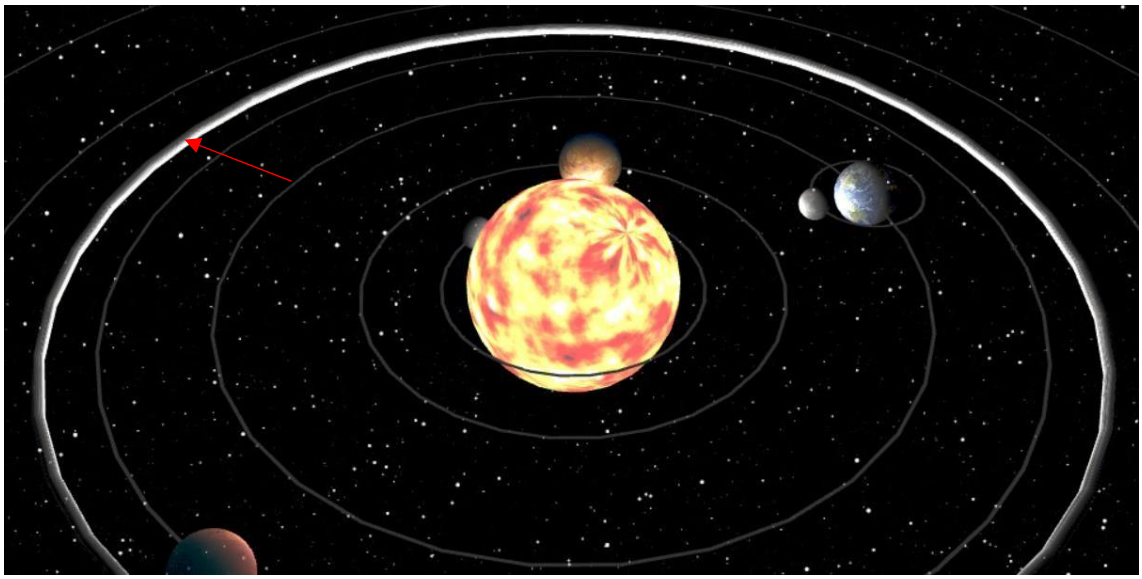


Implementação:

```
const satRings = {};
satRings.bufferInfo = twgl.primitives.createDiscBufferInfo(gl, 0.60, 30, 1, 0.39);
satRings.uniforms = {};
satRings.uniforms.u_texture = twgl.createTexture(gl, { src: '/text/disco.jpg' });
arrayObj.push(satRings);
```

Cintura de asteroides:

Imagem:



Implementação:

```
const asteroBelt = {};
asteroBelt.bufferInfo = twgl.primitives.createTorusBufferInfo(gl, 2.9, 0.04, 40, 40);
asteroBelt.uniforms = {};
asteroBelt.uniforms.u_texture = twgl.createTexture(gl, { src: '/text/astero.jpg' });
arrayObj.push(asteroBelt);
```

Realismo no tempo de órbita (à escala):

Os tempos de órbita foram determinados (à escala) usando como base a tabela em baixo com os períodos de rotação e de translação de cada planeta.

Planeta	Período de Rotação	Período de Translação
Mercúrio	58,6 dias	87,97 dias
Vênus	243 dias	224,7 dias
Terra	0,99 dia	365,26 dias
Marte	1,03 dias	1,88 anos
Júpiter	0,41 dia	11,86 anos
Saturno	0,45 dia	29,46 anos
Urano	0,72 dia	84,01 anos
Netuno	0,67 dia	164,79 anos
Plutão	6,39 dias	248,59 anos

Por exemplo, o período de translação de Marte é de 1,88 anos e o de Júpiter é de 11,86 anos. O tempo de translação no código definido para Marte é time*0.0013 e para Júpiter, time*0.00021

```
marte.uniforms.u_world = m4.multiply(m4.rotationY(0.0013*time), m4.multiply(m4.translation([2.6, 0, 0]), m4.rotationY(0.0096*time)));
surfaceToLightDirection = m4.inverse(marte.uniforms.u_world);
marte.uniforms.u_worldit = m4.transpose(surfaceToLightDirection);
```

```
jupiter.uniforms.u_world = m4.multiply(m4.rotationY(0.00021*time), m4.multiply(m4.translation([3.4, 0, 0]), m4.rotationY(0.024*time)));
surfaceToLightDirection = m4.inverse(jupiter.uniforms.u_world);
jupiter.uniforms.u_worldit = m4.transpose(surfaceToLightDirection);
```

A razão entre o período de translação real de Júpiter e o período de translação real de Marte é aproximadamente 6,3.

A razão entre o período de translação no código de Marte e o período de translação no código de Júpiter é aproximadamente 6,2.

Isto significa que Marte faz a translação 6 vezes mais rápido que Júpiter. Esta razão mantém-se no nosso projeto.