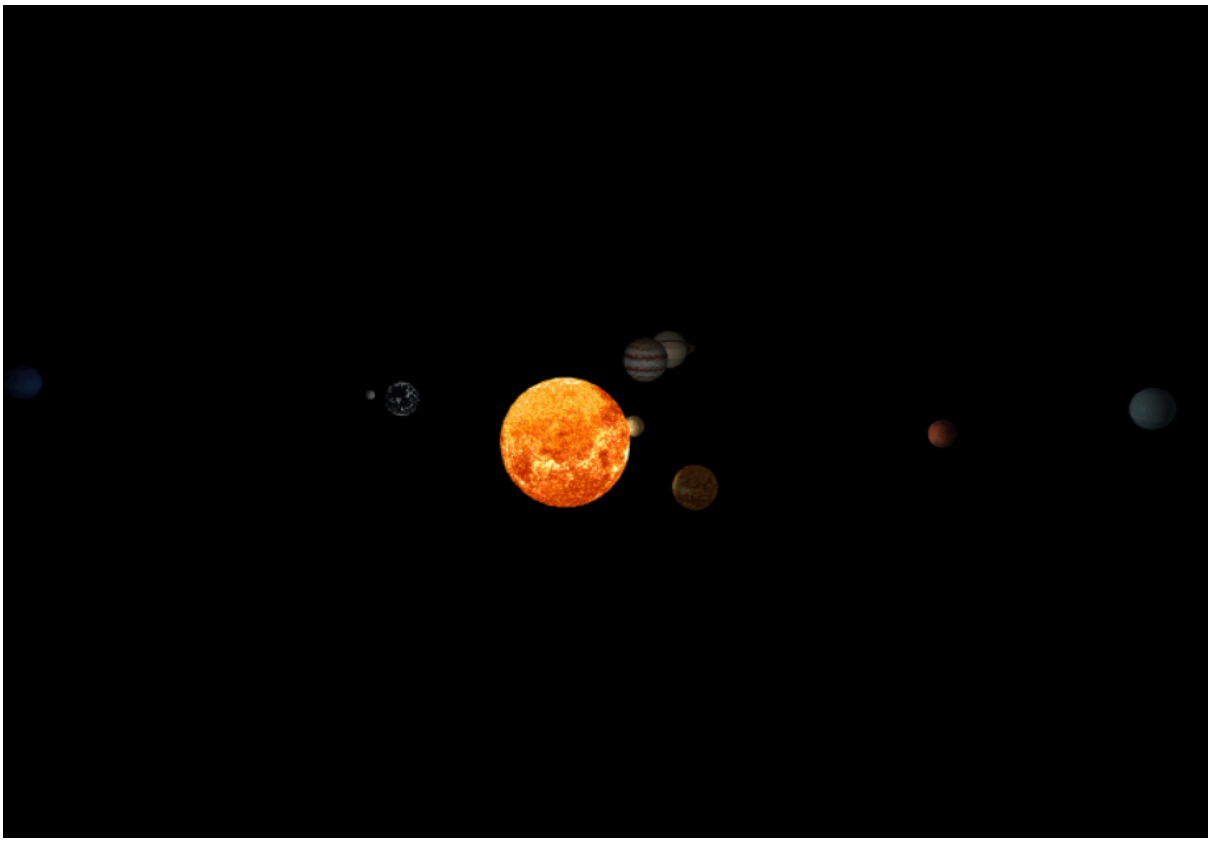
SISTEMA SOLAR

GRÁFICOS POR COMPUTADOR - PRÁCTICA 2



Ariel Carnés Blasco
Adrián Martín Martín

ÍNDICE

1. Implementación del Sistema Solar

Se han generado dos pares de shaders, uno para el Sol y otro para el resto de los cuerpos dibujados. Esto es debido a que el sol siempre ha de estar iluminado ya que será en él donde se sitúe el punto de luz, mientras que el resto de los cuerpos celestes sólo se ven iluminados en la dirección del sol.

```
<script id="shader-vs" type="x-shader/x-vertex">#version 300 es
    precision mediump float;

    layout (location = 0) in vec3 aPos;
    layout (location = 2) in vec2 aTexCoord;
    uniform UBO {
      mat4 projection;
      mat4 view;
      vec3 viewPos;
    };

    uniform mat4 model;
    out vec2 TexCoords;

    void main()
    {
        gl_Position = projection * view * model * vec4(aPos, 1.0);
        TexCoords = aTexCoord;
    }
```

```
<script id="shader-fs" type="x-shader/x-fragment">#version 300 es
  precision mediump float;

  uniform vec3 color;
  in vec2 TexCoords;
  uniform sampler2D diffuseTexture;

  out vec4 FragColor;

  void main()
  {
    vec3 textureColor = texture(diffuseTexture, TexCoords).rgb;
    FragColor = vec4(textureColor, 1.0);
  }
```

En estos shaders (los del sol) vemos como únicamente buscamos obtener su posición y asignarle su respectiva textura, pero no se establece ninguna relación con la luz.

El resto de los cuerpos dibujados tienen asignados los siguientes shaders:

```
<script id="shader-vs-texture" type="x-shader/x-vertex">#version 300 es
        precision mediump float;

        layout (location = 0) in vec3 aPos;
        layout (location = 1) in vec3 aNormal;
        layout (location = 2) in vec2 aTexCoord;
        uniform UBO {
          mat4 projection;
          mat4 view;
          vec3 viewPos;
        };
        uniform mat4 model;
        out vec3 Normal;
        out vec3 fragPos;
        out vec2 TexCoords;

        void main()
        {
         gl_Position = projection * view * model * vec4(aPos, 1.0);
         Normal = aNormal;
         mat3 normalM = mat3( transpose(inverse( model ) ) );
         Normal = normalM * aNormal;
         fragPos = vec3(model * vec4(aPos, 1.0));
         TexCoords = aTexCoord;
        }
</script>
```

```
  void main()
  {
    // Calculo de la atenuacion
    float distance = length(lightPos - fragPos);
    float attenuation = 1.0 / (constant + linear * distance + quadratic * (distance * distance));

    // Ambient
    float ambientStrength = 0.4;
    vec3 ambient = ambientStrength * lightColor;
    ambient *= attenuation;

    // Diffuse
    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(lightPos - fragPos);
    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = diff * lightColor;
    diffuse *= attenuation;

    // Specular
    float specularStrength = 0.2;
    vec3 viewDir = normalize(viewPos - fragPos);
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32.0);
    vec3 specular = specularStrength * spec * lightColor;
    specular *= attenuation;

    // Calculo de la implementacion de la textura y las luces
    vec3 result = (ambient + diffuse + specular);
    vec4 textureColor = texture(diffuseTexture, TexCoords);
    result *= textureColor.rgb;

    FragColor = vec4(result, 1);
  }
</script>
```

En estos shaders, además de cargar la textura correspondiente, se introduce el sombreado de Phong, definiendo los distintos factores normal, difuso y especular. También está implementada la atenuación, de esta manera la luz del sol va perdiendo intensidad.

El siguiente cambio que comentar, es la creación de distintos shaderPrograms, uno por cada cuerpo representado:

```
let shaderProgramSun = new ShaderProgram();
let shaderProgramMercury = new ShaderProgram();
let shaderProgramVenus = new ShaderProgram();
let shaderProgramEarth = new ShaderProgram();
let shaderProgramMoon = new ShaderProgram();
let shaderProgramMars = new ShaderProgram();
let shaderProgramJupiter = new ShaderProgram();
let shaderProgramSaturn = new ShaderProgram();
let shaderProgramUranus = new ShaderProgram();
let shaderProgramNeptune = new ShaderProgram();
let shaderProgramRing = new ShaderProgram();
```

Tras ello, se crean la esfera (una única esfera la cual será transformada para representar todos los planetas) y se definen sus distintos buffers:

```
const sphereGeo = createSphere();
const ring = createRing();

var VAO, VBO, VBO2, VBO3, EBO;
VAO = gl.createVertexArray();
VBO = gl.createBuffer();
VBO2 = gl.createBuffer();
VBO3 = gl.createBuffer();
EBO = gl.createBuffer();
```

A continuación, se le asignan los distintos shaders a los cuerpos. El sol tiene un tipo de shader mientras el resto de los planetas tienen otro debido a que el sol está siempre iluminado:

```
// SOL (TIENE SHADERS DISTINTOS YA QUE SIEMPRE HA DE ESTAR ILIMINADO)
shaderProgramSun.createVertexShader(
  document.getElementById("shader-vs").text
);
shaderProgramSun.createFragmentShader(
  document.getElementById("shader-fs").text
);
shaderProgramSun.compile();
shaderProgramSun.link();

shaderProgramSun.autocatching();

// MERCURIO
shaderProgramMercury.createVertexShader(
  document.getElementById("shader-vs-texture").text
);
shaderProgramMercury.createFragmentShader(
  document.getElementById("shader-fs-texture").text
);
shaderProgramMercury.compile();
shaderProgramMercury.link();

shaderProgramMercury.autocatching();
```

Posteriormente, se crea el UBO, en el cual se declaran las variables que se van a compartir y se asignan todos los shaderPrograms que van a hacer uso de él:

```
// ------------------------------------------------
// SE CREA EL UBO Y SE LE ASIGNAN LOS SHADERPROGRAMS
// ------------------------------------------------

let ubo = new UBO( "UBO", shaderProgramSun, [ "projection", "view", "viewPos"]);
ubo.attachProgram( shaderProgramSun );
ubo.attachProgram( shaderProgramMercury);
ubo.attachProgram( shaderProgramVenus);
ubo.attachProgram( shaderProgramEarth);
ubo.attachProgram( shaderProgramMoon);
ubo.attachProgram( shaderProgramMars);
ubo.attachProgram( shaderProgramJupiter);
ubo.attachProgram( shaderProgramSaturn);
ubo.attachProgram( shaderProgramUranus);
ubo.attachProgram( shaderProgramNeptune);
ubo.attachProgram( shaderProgramRing);
```

Se crean algunas variables necesarias para el código y se establece que el color de fondo sea negro.

```
gl.clearColor(0.0, 0.0, 0.0, 1.0);
gl.viewport(0, 0, gl.canvas.width, gl.canvas.height);

// Variables de la luz
var lightPos = [0.0, 0.0, 0.0];    // Posicion del foco de luz
var lightColor = [1.0, 1.0, 1.0];   // Color de la luz (blanco)

// Variables para la atenuacion de la luz
var constant = 1.0;
var linear = 0.045;
var quadratic = 0.0075;
```

Después se crean y cargan las texturas de cada uno de los cuerpos que se van a representar:

```
// TEXTURA TIERRA
var earthTexture = new Image();
var earthTextureID = gl.createTexture();
earthTexture.onload = function () {
  gl.bindTexture(gl.TEXTURE_2D, earthTextureID);
  gl.texImage2D(
    gl.TEXTURE_2D,
    0,
    gl.RGBA,
    gl.RGBA,
    gl.UNSIGNED_BYTE,
    earthTexture
  );
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
  gl.generateMipmap(gl.TEXTURE_2D);
  gl.bindTexture(gl.TEXTURE_2D, null);
  renderFunc(0.0);
};
earthTexture.src = "resources/earth_diffuse.jpg"; // Ruta relativa de la textura
```

El procedimiento es el mismo para todas las texturas.

Ahora pasamos a crear la función renderFunc. El primer cambio realizado es la creación de los modelos de todos los cuerpos representados.

```
var view = mat4.create();
var modelSun = mat4.create();
var modelMercury = mat4.create();
var modelVenus = mat4.create();
var modelEarth = mat4.create();
var modelMoon = mat4.create();
var modelMars = mat4.create();
var modelJupiter = mat4.create();
var modelSaturn = mat4.create();
var modelUranus = mat4.create();
var modelNeptune = mat4.create();
var modelRing = mat4.create();
```

Tras ello, se comienzan a dibujar los planetas siguiendo un mismo esquema.

```
// VENUS
shaderProgramVenus.bind();

shaderProgramVenus.setUniform3f(
  "lightPos",
  lightPos[0],
  lightPos[1],
  lightPos[2]
);

shaderProgramVenus.setUniform1f("constant", constant);
shaderProgramVenus.setUniform1f("linear", linear)
shaderProgramVenus.setUniform1f("quadratic", quadratic)

shaderProgramVenus.createUniform("diffuseTexture");

// Renderizado de la textura
gl.activeTexture(gl.TEXTURE1);
shaderProgramVenus.setUniform1i("diffuseTexture", 1);
gl.bindTexture(gl.TEXTURE_2D, venustextureID);
```

```
// ------------------------------------------------
// ROTACION Y TRANSLACION
// ------------------------------------------------
var venusOrbitRadius = 6.0; // Radio de la órbita de la Venus
var venusOrbitSpeed = -1.176; // Velocidad de rotación de la Venus alrededor del sol
var venusOrbitAngle = venusOrbitSpeed * time;
var venusPosition = [
  6.0,
  0.0,
  0.0
];
// Calcular el ángulo de rotación de la Venus sobre su propio eje
var venusRotationSpeed =  0.0078;
var venusRotationAngle = venusRotationSpeed * time;

mat4.rotate(modelVenus, modelVenus, venusOrbitAngle, [0, 1, 0]);
mat4.translate(modelVenus, modelVenus, venusPosition);
mat4.rotateY(modelVenus, modelVenus, venusRotationAngle);
mat4.scale(modelVenus, modelVenus, [0.6, 0.6, 0.6]);

shaderProgramVenus.setUniformMat4("model", modelVenus);

gl.bindVertexArray(VAO);
gl.drawElements(
  gl.TRIANGLES,
  sphereGeo.indices.length,
  gl.UNSIGNED_INT,
  0
);
```

En primer lugar, se asigna el shaderProgram respectivo del planeta que se vaya a crear en ese momento. Luego, se envían las variables uniformes necesarias. A continuación, se carga la textura correspondiente. Tras ello, se establece la rotación y la traslación realizada por cada planeta. Y por último se asigna el array de vértices correspondiente y se pinta el planeta.

Las velocidades de los planetas se han intentado mantener proporcionales de la siguiente manera, se han establecido las velocidades de la tierra como 1 y 1.5 para la velocidad de translación y de rotación respectivamente. Y en consecuencia con esta conversión de los 107.244 km/h y 1674 km/h de sus velocidades se han obtenido el resto de las velocidades de los planetas para mantener una proporción.

En el caso de los tamaños se ha ignorado esta proporción, ya que para la práctica si se mantenía el tamaño de algunos planetas eran tan pequeños que apenas se podían ver.

En el caso de la luna, ya que esta no rota sobre sí misma y no rota sobre el sol, el procedimiento es el mismo, pero sin añadir esta rotación, y a la hora de calcular su posición se utiliza la posición de la Tierra para que rote alrededor de ella.

```
// ------------------------------------------------
// TRANSLACION
// ------------------------------------------------
var moonOrbitRadius = 2.5; // Radio de la órbita de la Luna
var moonOrbitSpeed = -3; // Velocidad de rotación de la Luna alrededor de la Tierra
var moonOrbitAngle = moonOrbitSpeed * time;
var moonPosition = [
  2.5,
  0.0,
  0.0,
];

mat4.multiply(modelMoon, modelMoon, modelEarth);
mat4.rotate(modelMoon, modelMoon, moonOrbitAngle, [0, 1, 0]);
mat4.translate(modelMoon, modelMoon, moonPosition);
mat4.scale(modelMoon, modelMoon, [0.3, 0.3, 0.3]);
```

2. Texturas especular y de normales en la Tierra

Para implementar la textura especular y de normales de la tierra, hemos tenido que crear nuevos shaders para la tierra, que reciben parámetros para calcular las normales globales a partir de las del mapa de normales, ya que si no utiliza las normales del plano tangencial que no son las que tiene el objeto realmente. En el shader-fs también cambiamos el cálculo de la fuerza especular por lo que podemos obtener de la textura especular.

```
<script id="shader-vs-tierra" type="x-shader/x-vertex">#version 300 es
  precision mediump float;

  layout (location = 0) in vec3 aPos;
  layout (location = 1) in vec3 aNormal;
  layout (location = 2) in vec2 aTexCoord;
  layout (location = 3) in vec3 aTangent;
  layout (location = 4) in vec3 aBitangent;

  uniform UBO {
    mat4 projection;
    mat4 view;
    vec3 viewPos;
    vec3 lightColor;
  };

  uniform mat4 model;
  out vec3 fragPos;
  out vec2 TexCoords;
  out mat3 TBN;

  void main()
  {
    gl_Position = projection * view * model * vec4(aPos, 1.0);

    fragPos = vec3(model * vec4(aPos, 1.0));
    TexCoords = aTexCoord;

    vec3 T = normalize(vec3(model * vec4(aTangent,   0.0)));
    vec3 B = normalize(vec3(model * vec4(aBitangent, 0.0)));
    vec3 N = normalize(vec3(model * vec4(aNormal,    0.0)));
    mat3 TBN = mat3(T, B, N);
  }
</script>
```

```
out vec4 FragColor;
in mat3 TBN;
in vec3 fragPos;
in vec2 TexCoords;
```

Cambiamos vec3 Normal por TBN en el fragment shader

```
uniform sampler2D diffuseTexture;
uniform sampler2D normalTexture;
uniform sampler2D specularTexture;

void main()
{
  // Obtener la normal desde la textura de normales
  vec3 normal = texture(normalTexture, TexCoords).rgb;
  normal = normal * 2.0 - 1.0;
  normal = normalize(TBN * normal);

  // Calculo de la atenuacion
  float distance = length(lightPos - fragPos);
  float attenuation = 1.0 / (constant + linear * distance + quadratic * (distance * distance));

  // Ambient
  float ambientStrength = 0.4;
  vec3 ambient = ambientStrength * lightColor;
  ambient *= attenuation;

  // Diffuse
  vec3 lightDir = normalize(lightPos - fragPos);
  float diff = max(dot(normal, lightDir), 0.0);
  vec3 diffuse = diff * lightColor;
  diffuse *= attenuation;

  // Calcular la luz especular
  vec3 viewDir = normalize(viewPos - fragPos);
  vec3 reflectDir = reflect(-lightDir, normal); // Dirección del reflejo
  float specularStrength = texture(specularTexture, TexCoords).r; // Intensidad especular
  float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32.0);
  vec3 specular = specularStrength * spec * lightColor;
  specular *= attenuation;

  // Calculo de la implementacion de la textura y las luces
  vec3 result = (ambient + diffuse + specular);
  vec4 textureColor = texture(diffuseTexture, TexCoords);
  result *= textureColor.rgb;

  FragColor = vec4(result, 1);
}
```

```
//Calculo de las tangentes y bitangentes para usar para el mapa normal
let tangents = [];
let bitangents = [];

for (let i = 0; i < sphereGeo.indices.length; i += 3) {

  let vertices = new Float32Array (sphereGeo.vertices);
  let normals = new Float32Array (sphereGeo.normals);
  let texCoords = new Float32Array (sphereGeo.textCoords);
  let indices = new Float32Array (sphereGeo.indices);

  let i0 = indices[i];
  let i1 = indices[i + 1];
  let i2 = indices[i + 2];

  let pos0 = [vertices[i0 * 3], vertices[i0 * 3 + 1], vertices[i0 * 3 + 2]];
  let pos1 = [vertices[i1 * 3], vertices[i1 * 3 + 1], vertices[i1 * 3 + 2]];
  let pos2 = [vertices[i2 * 3], vertices[i2 * 3 + 1], vertices[i2 * 3 + 2]];

  let uv0 = [texCoords[i0 * 2], texCoords[i0 * 2 + 1]];
  let uv1 = [texCoords[i1 * 2], texCoords[i1 * 2 + 1]];
  let uv2 = [texCoords[i2 * 2], texCoords[i2 * 2 + 1]];

  let edge1 = [pos1[0] - pos0[0], pos1[1] - pos0[1], pos1[2] - pos0[2]];
  let edge2 = [pos2[0] - pos0[0], pos2[1] - pos0[1], pos2[2] - pos0[2]];

  let deltaUV1 = [uv1[0] - uv0[0], uv1[1] - uv0[1]];
  let deltaUV2 = [uv1[0] - uv0[0], uv1[1] - uv0[1]];

  let f = 1.0 / (deltaUV1[0] * deltaUV2[1] - deltaUV1[1] * deltaUV2[0]);
  let tangent = [
      f * (deltaUV2[1] * edge1[0] - deltaUV1[1] * edge2[0]),
      f * (deltaUV2[1] * edge1[1] - deltaUV1[1] * edge2[1]),
      f * (deltaUV2[1] * edge1[2] - deltaUV1[1] * edge2[2])
  ];

  let bitangent = [
      f * (deltaUV1[0] * edge2[0] - deltaUV2[0] * edge1[0]),
      f * (deltaUV1[0] * edge2[1] - deltaUV2[0] * edge1[1]),
      f * (deltaUV1[0] * edge2[2] - deltaUV2[0] * edge1[2])
  ];
```

Función que calcula las tangentes y bitangentes de los distintos vértices de la esfera.

Para que el shader de vértices reciba las tangentes y bitangentes tendremos que crear nuevos VBO para el VAO, en las posiciones 3 y 4 y asignarles estos valores.

```
gl.vertexAttribPointer(2, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(2);
// tangent attribute
gl.bindBuffer(gl.ARRAY_BUFFER, VBO4);
gl.bufferData(
  gl.ARRAY_BUFFER,
  new Float32Array(tangents),
  gl.STATIC_DRAW
);
gl.vertexAttribPointer(3, 3, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(3);
// bitangent attribute
gl.bindBuffer(gl.ARRAY_BUFFER, VBO5);
gl.bufferData(
  gl.ARRAY_BUFFER,
  new Float32Array(sphereGeo.bitangents),
  gl.STATIC_DRAW
);
```

Por último tenemos que cargar las texturas y enviarlas como uniform al fragment shader.

```
//textura de normales
var earthNormalTexture = new Image();
var earthNormalTextureID = gl.createTexture();

earthNormalTexture.onload = function () {
  gl.bindTexture(gl.TEXTURE_2D, earthNormalTextureID);
  gl.texImage2D(
    gl.TEXTURE_2D,
    0,
    gl.RGBA,
    gl.RGBA,
    gl.UNSIGNED_BYTE,
    earthNormalTexture
  );
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
  gl.generateMipmap(gl.TEXTURE_2D);
  gl.bindTexture(gl.TEXTURE_2D, null);
  renderFunc(0.0);
};

earthNormalTexture.src = "resources/earth_normal.jpg"; // Ruta a tu mapa de normales
```

Con la especular se hace igual.

```
// Declarar uniformes
shaderProgramEarth.createUniform("diffuseTexture");
shaderProgramEarth.createUniform("normalTexture");
shaderProgramEarth.createUniform("specularTexture");

gl.activeTexture(gl.TEXTURE1);
gl.bindTexture(gl.TEXTURE_2D, earthTextureID);
shaderProgramEarth.setUniform1i("diffuseTexture", 1);// Vincular la textura difusa

gl.activeTexture(gl.TEXTURE2);
gl.bindTexture(gl.TEXTURE_2D, earthNormalTextureID);
shaderProgramEarth.setUniform1i("normalTexture", 2);  // Vincular la textura de normales

gl.activeTexture(gl.TEXTURE3);
gl.bindTexture(gl.TEXTURE_2D, earthSpecularTextureID);
shaderProgramEarth.setUniform1i("specularTexture", 3); // Vincular la textura especular
```

Dentro de la funcion renderFunc, en el momento de dibujar la Tierra.

3. Anillos de Saturno

Tenemos que crear el anillo y sus buffers.

```
const sphereGeo = createSphere();
const ring = createRing();
```

```
//Buffers para el anillos
var VAO_RING, VBO_RING, VBO2_RING, VBO3_RING, EBO_RING;
VAO_RING = gl.createVertexArray();
VBO_RING = gl.createBuffer();
VBO2_RING = gl.createBuffer();
VBO3_RING = gl.createBuffer();
EBO_RING = gl.createBuffer();
```

De la misma forma que con la esfera.

Tras esto lo trataremos como al resto de objetos, cargaremos su textura, enviaremos las variables que el shader necesita y lo renderizamos. Para los shaders usa los shaders de texturas que usan los planetas.

```
// TEXTURA ANILLO
var ringTexture = new Image();
var ringTextureID = gl.createTexture();
ringTexture.onload = function () {
  gl.bindTexture(gl.TEXTURE_2D, ringTextureID);
  gl.texImage2D(
    gl.TEXTURE_2D,
    0,
    gl.RGBA,
    gl.RGBA,
    gl.UNSIGNED_BYTE,
    ringTexture
  );
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
  gl.generateMipmap(gl.TEXTURE_2D);
  gl.bindTexture(gl.TEXTURE_2D, null);
  renderFunc(0.0);
};
ringTexture.src = "resources/anillo.jpg"; // Ruta relativa de la textura
```

```
// ANILLO
shaderProgramRing.bind();

shaderProgramRing.setUniform3f(
  "lightPos",
  lightPos[0],
  lightPos[1],
  lightPos[2]
);

shaderProgramRing.setUniform1f("constant", constant);
shaderProgramRing.setUniform1f("linear", linear)
shaderProgramRing.setUniform1f("quadratic", quadratic)

shaderProgramRing.createUniform("diffuseTexture");

// Renderizado de la textura
gl.activeTexture(gl.TEXTURE1);
shaderProgramRing.setUniform1i("diffuseTexture", 1);
gl.bindTexture(gl.TEXTURE_2D, ringTextureID);
var ringPosition = [modelSaturn[12],modelSaturn[13], modelSaturn[14]];


mat4.translate(modelRing, modelRing, ringPosition);
mat4.rotateX(modelRing, modelRing, Math.PI/3);

mat4.scale(modelRing, modelRing, [1.25, 1.25, 1.25]);

shaderProgramRing.setUniformMat4("model", modelRing);

gl.bindVertexArray(VAO_RING);
gl.drawElements(
  gl.TRIANGLES,
  ring.indices.length,
  gl.UNSIGNED_INT,
  0
);
```

La posición del anillo es la de Saturno.

4. Añadir funciones de cámara

Para esta parte hemos reutilizado el código de la práctica anterior, en la que ya teníamos implementado tanto el movimiento de la cámara con teclas como el cambiar el ángulo de la cámara con ratón. A continuación las funciones que hemos usado.

```
// Variables para implementar el movimiento del raton
var lastX = 450.0;
var lastY = 450.0;
var yaw = -90.0;
var pitch = 0.0;

// Funcion que mueve la camara en funcion del movimiento del raton.
function mouse_callback(event) {
  var xoffset = event.clientX - lastX;
  var yoffset = lastY - event.clientY;
  lastX = event.clientX;
  lastY = event.clientY;

  var sensitivity = 0.15;
  xoffset *= sensitivity;
  yoffset *= sensitivity;

  yaw += xoffset;
  pitch += yoffset;

  if (pitch > 89.0) {
    pitch = 89.0;
  }
  if (pitch < -89.0) {
    pitch = -89.0;
  }

  var direction = vec3.create();
  direction[0] =
    Math.cos(Math.radians(yaw)) * Math.cos(Math.radians(pitch));
  direction[1] = Math.sin(Math.radians(pitch));
  direction[2] =
    Math.sin(Math.radians(yaw)) * Math.cos(Math.radians(pitch));
  cameraFront = vec3.normalize(vec3.create(), direction);
}

document.addEventListener("mousemove", mouse_callback);
```

```
document.addEventListener("keydown", function (event) {
  var step = 0.5;

  // Posicion de la camara
  if (event.keyCode == 37) {
    // Flecha izquierda
    cameraPos[0] -= step; // Disminuye la posicion X de la camara
  } else if (event.keyCode == 39) {
    // Flecha derecha
    cameraPos[0] += step; // Aumenta la poscion X de la camara
  } else if (event.keyCode == 38) {
    // Flecha arriba
    cameraPos[1] += step; // Aumenta la poscion Y de la camara
  } else if (event.keyCode == 40) {
    // Flecha abajo
    cameraPos[1] -= step; // Disminuye la posicion X de la camara
  } else if (event.keyCode == 188) {
    // ,
    cameraPos[2] += step; // Aumenta la poscion Z de la camara
  } else if (event.keyCode == 190) {
    // .
    cameraPos[2] -= step; // Disminuye la posicion Z de la camara
  }
});
```

5. Atmósfera en la Tierra (Uso de Blending)

La creación de una atmósfera alrededor de la Tierra conlleva crear otro tipo de shader para una textura que va a ser translúcida. El mayor cambio es que el valor alfa de la textura que se usa representa la opacidad, por lo que tendremos que añadir lo siguiente al final del main del shader de texturas.

```
// Calculo de la implementacion de la textura y las luces
vec3 result = (ambient + diffuse + specular);
vec4 textureColor = texture(cloudTexture, TexCoords);
result *= textureColor.rgb;

if (textureColor.a < 0.1)
discard;

FragColor = vec4(result, textureColor.a);
}
```

Los siguientes pasos son seguir las instrucciones que marcaba el enunciado de la práctica. Creamos una nueva esfera, la configuramos para que sea algo más grande que la tierra y que rote sobre sí misma a una velocidad distinta y por último la renderizamos con blend.

```
// NUBES
shaderProgramClouds.createVertexShader(
  document.getElementById("shader-vs-texture").text
);
shaderProgramClouds.createFragmentShader(
  document.getElementById("shader-fs-nubes").text
);
shaderProgramClouds.compile();
shaderProgramClouds.link();

shaderProgramClouds.autocatching();
```

```
// Calcular el ángulo de rotación de la atmósfera sobre su propio eje
var cloudRotationSpeed = 2.0;
var cloudRotationAngle = cloudRotationSpeed * time;

mat4.rotate(modelClouds, modelClouds, earthOrbitAngle, [0, 1, 0]);
mat4.translate(modelClouds, modelClouds, earthPosition);
mat4.rotateY(modelClouds, modelClouds, cloudRotationAngle);
mat4.scale(modelClouds, modelClouds, [0.62, 0.62, 0.62]);

shaderProgramClouds.setUniformMat4("model", modelClouds);

// Habilitar blending
gl.enable(gl.BLEND);
gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);

gl.bindVertexArray(VAO);
gl.drawElements(
  gl.TRIANGLES,
  sphereGeo.indices.length,
  gl.UNSIGNED_INT,
  0
);

// Deshabilitar blending
gl.disable(gl.BLEND);
```

Las texturas translúcidas se renderizan lo último.

6. Cámara sobre la Tierra

Hemos elegido añadir la opción de que la cámara orbite junto con la tierra en una posición más alta. Para ello hemos creado una variable cam que indica el modo de cámara que se usa en el momento. Habrá una comprobación en renderFunc para fijar la cámara sobre la tierra si está en ese modo y una forma de cambiar el modo (pulsando la 'c').

```
// Variables de la camara
var cameraPos = [0.0, 0.0, 20.0];
var cameraFront = [0.0, 0.0, -1.0];
var cameraUp = [0.0, 1.0, 0.0];
var cam = 0;
```

```
if (cam!=0){
  cameraPos = [
  9.0 * Math.cos(time),
  2.0,
  9.0 * Math.sin(time),
  ];

}
```

Son los valores de posición de la Tierra, menos en el eje Y que es mayor en este caso.

```
  if (event.keyCode == 67){
    // C
    cam = (cam + 1)%2;    // Cambia la camara de modo
  }
});
```

7. Fotos del proyecto

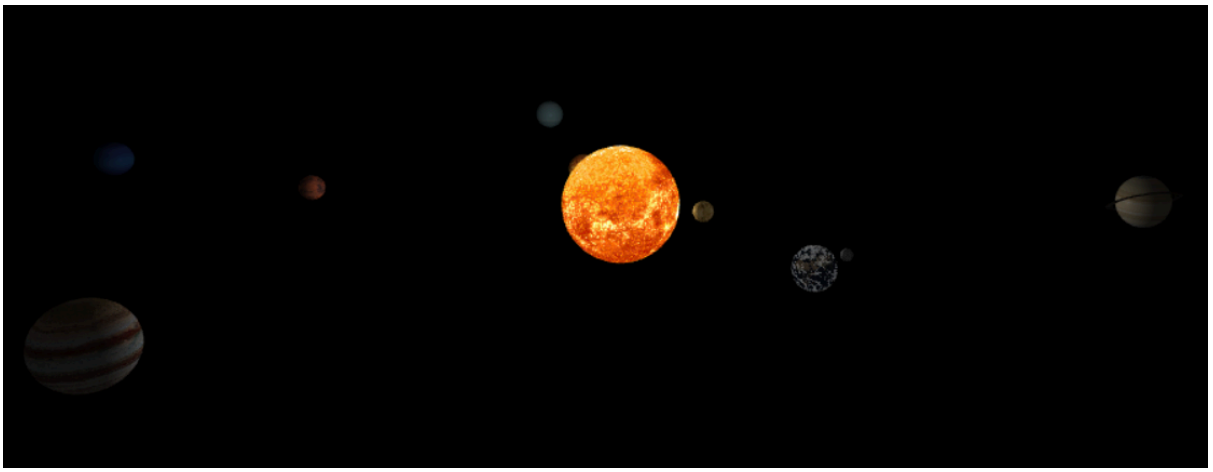


Foto del Sistema Solar con los planetas orbitando.

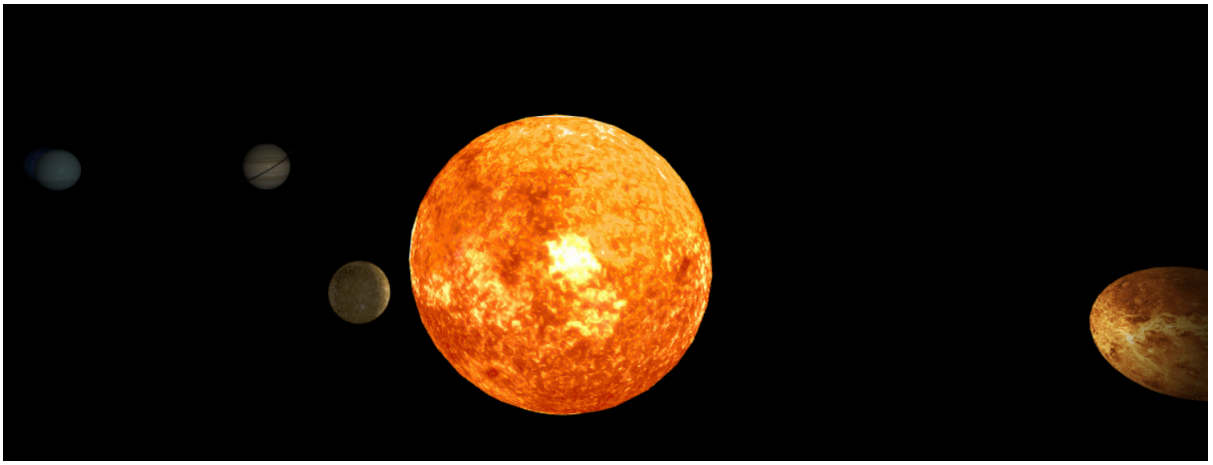


Foto del Sol desde la cámara sobre la Tierra.

Foto de la Tierra y la Luna desde la cámara sobre la Tierra.

8. Bibliografía

Del aula virtual hemos aplicado partes de las prácticas anteriores y contenido de las clases. Lo que hemos buscado en internet ha sido en general de esta página:

- https://learnopengl.com

Para las partes de texturas de normales, especulares y blending, hemos usado respectivamente:

- https://learnopengl.com/Advanced-Lighting/Normal-Mapping
- https://learnopengl.com/Lighting/Lighting-maps
- https://learnopengl.com/Advanced-OpenGL/Blending