Universidad
Rey Juan Carlos

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA DE SOFTWARE

Curso académico 2023/2024
Gráficos por Computador
Práctica 2: Sistema solar

Grupo 30:
Stefano Tomasini Hoefner
Rémi Cazoulat

Índice

Índice de figuras

1. Objetivos de la práctica

El objetivo general de esta práctica es implementar el sistema solar. Para conseguir esto, hace falta volverse proficiente en la manipulación de matrices modelo en el espacio 3-D para ser capaz de implementar órbitas, y rotaciones sobre el eje del objeto, este es el principal propósito didáctico de la práctica, junto a la configuración y alternado de distintos *vertex arrays* para dibujar distintas formas geométricas en una misma escena (esferas y anillos) y el manejo de texturas de diferentes tipos para conseguir ciertos efectos visuales. También se repasan conceptos de la práctica anterior: el modelo de iluminación Phong, y la implementación de una luz puntual. Como partes opcionales, se pide implementar la atmósfera de una tierra como una esfera semitransparente, un campo de asteroides mediante el uso de instanciación, la capacidad de montar la cámara sobre los planetas, y un *skybox* (2-D o 3-D).

2. Cámara

Para definir la cámara, se hace uso de la matriz `lookat` de la biblioteca `gl-matrix`. Para crear esta matriz 4x4, hacen falta tres vectores tridimensionales: el vector `pos` que comprende la posición de la cámara, el vector `front`, que es a donde la cámara mira, y el ultima, el vector `up`, que es a donde apunta la parte superior de la cámara.

2.1. Traslación

Para trasladar la cámara, se debe alterar el vector `cameraPos`. Al vector `cameraPos` se le suma el vector `cameraUp`, que determina el movimiento vertical, y/o el vector `cameraFront` si se adelanta o retrocede, y/o el vector resultante del producto vectorial entre el vector `cameraFront` y el vector `cameraUp` si se produce un movimiento lateral.

2.2. Rotación

Para cambiar a donde apunta la cámara, hay que actualizar el vector `front` según dos ángulos: `yaw` y `pitch`. `yaw` controla la rotación lateral, mientras que `pitch` determina la rotación vertical dentro de lo que sería un plano vertical paralelo a la dirección de la cámara.

2.3. Controles

- W / Z / Flecha arriba : Adelanta la cámara
- S / Flecha abajo : Retrocede la cámara
- A / Q / Flecha izquierda : Desplaza la cámara a la izquierda
- D / Flecha derecha : Desplaza la cámara a la derecha
- Espacio: ascender
- Shift + espacio: descender
- C : Reinicia la posición y la rotación de la cámara
- Cursor : Cambia la dirección de la cámara

3. El Sol

Es el cuerpo celestial más fácil de implementar de esta práctica, ya que, únicamente hay que escalar su matriz modelo. Al situarse permanentemente en el origen sin realizar rotaciones, no hace falta hacer ni una sola operación de traslación ni rotación.

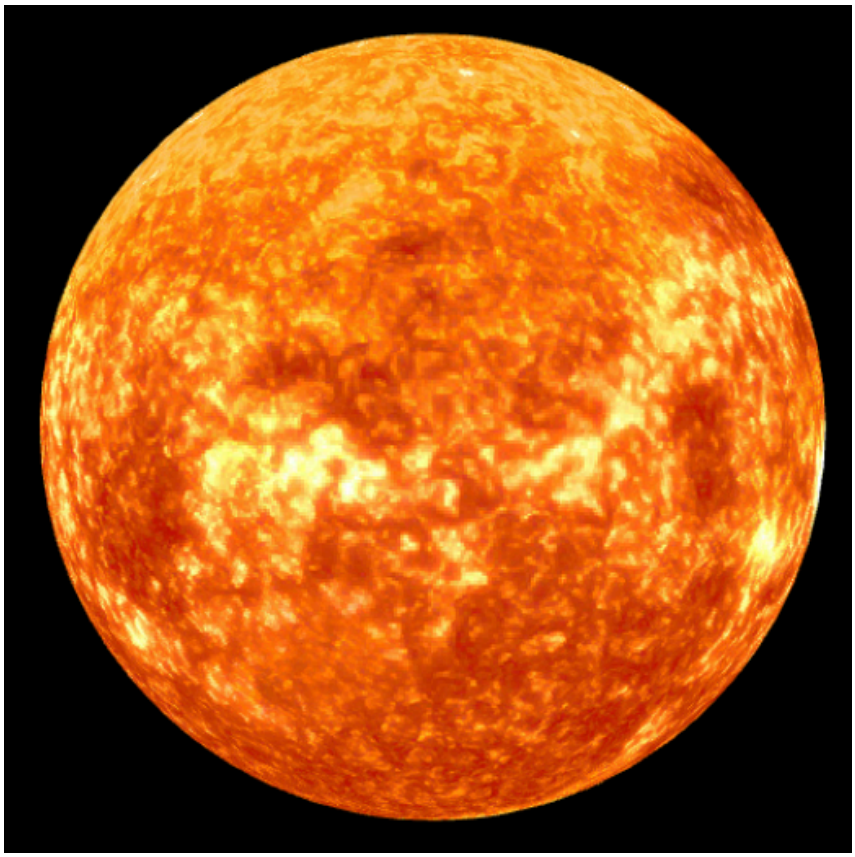

Figura 1: El Sol

Sin embargo, la estrella necesita un shader de fragmentos distinto y reducido. Esto es porque hay una luz puntual presente en la escena que simula la luz solar, y hay que dar la apariencia de que esta origina de la superficie del Sol (aunque en realidad tiene su origen en el centro), por lo que esta misma debe estar máximamente iluminada, así que directamente se omiten las computaciones de una iluminación Phong. Entonces, en vez de tener en cuenta la luz puntual, o cualquier otro tipo de luz al computar el color de sus fragmentos, simplemente se extraen los correspondientes colores de la textura sin tener nada más en cuenta:

```
in vec3 fragPos;
in vec2 vTexCoord;
out vec4 fragColor;
uniform sampler2D diffuseTex;
void main(){
    vec2 vertInvTexCoord = vec2(vTexCoord.x, -vTexCoord.y);

    fragColor = texture(diffuseTex, vertInvTexCoord).rgba;
}
```

Se usa la componente y negativizada de vTexCoord al extraer el color correspondiente de la textura del fragmento con el fin de que la textura no se vea invertida verticalmente. Esto se realiza en todos los shaders de fragmentos.

4. Planetas y otros cuerpos celestiales

Cada planeta tiene su correspondiente radio de órbita, sol, periodos de órbita, periodos de rotación sobre su eje, y texturas. Aunque los planetas son un poco más grandes de lo que deberían ser relativo al espacio que ocupan dentro del sistema solar, sí mantienen entre ellos sus ratios diametrales reales.

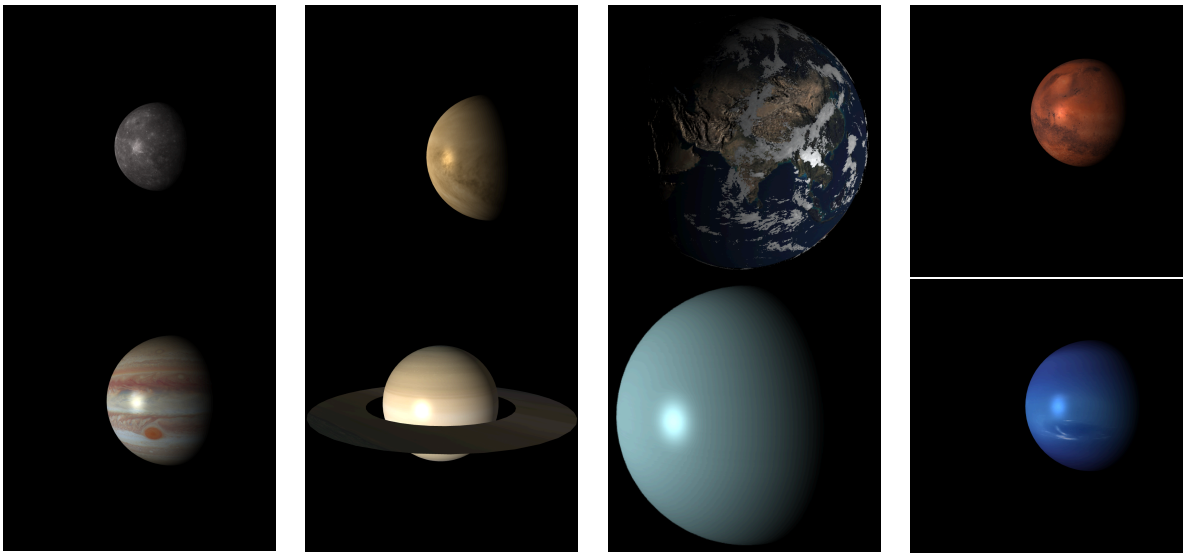

Figura 2: Los planetas implementados

Todos los planetas y algunos cuerpos celestiales (la Luna, la atmósfera de la Tierra, y el anillo de Saturno), a excepción de la Tierra, usan el mismo shader de fragmentos con iluminación Phong, que es casi idéntico al de la práctica anterior, con unas pequeñas alteraciones:

```
[...]
void main(){

    vec2 vertInvTexCoord = vec2(vTexCoord.x, -vTexCoord.y);

    vec3 norm = normalize(normal);

    vec3 lightDir = normalize(sunLightPos - fragPos);
    vec3 diffuse = max(dot(norm, lightDir), 0.0) * diffuseStrength;

    vec3 viewDir = normalize(cameraPos - fragPos);
    vec3 reflectDir = reflect(-lightDir, norm);
    vec3 specular = specularStrength * pow(max(dot(viewDir, reflectDir), 0.0),
specularRadius);

    vec4 texColor = texture(diffuseTex, vertInvTexCoord);

    vec3 lighting =  texColor.a * ambient + diffuse + specular;
    fragColor = vec4(lighting * texColor.rgb, 1.0);

    if (texColor.a < 0.4) {discard;}
}
```

En esta versión, se eliminó la luz direccional que había (solo se tiene en cuenta el origen de la luz puntual para computar la dirección de esta única fuente de luz), y, se descartan aquellos fragmentos que no superen un cierto umbral de transparencia. Esto es para que la atmósfera de la Tierra (una esfera semitransparente levemente más grande) no tape la superficie terrestre.

La posición y la rotación de los cuerpos orbitantes están calculados con una matriz 4x4 modelo. Para calcular la posición de un cuerpo, se crea una primera matriz que se llama `modelPos`. Se debe utilizar la función `rotate` de la biblioteca gl-matrix. Se hace una rotación sobre la matriz modelo del cuerpo, y después una traslación con la función `translate`, usando el radio de órbita (como módulo). Y después se multiplican las matrices modelo de los dos, para pasar las coordenadas del cuerpo a coordenadas globales. Para computar la rotación de un cuerpo sobre su eje (y que se ajuste a la duración de un día), se crea una segunda matriz 4x4 que se llama `modelRot`. Se hace una rotación con la función `rotate` sobre el cuerpo, usando el angulo que se calculó haciendo uso del tiempo global de la simulación y el periodo de rotación sobre el eje. Al final, se multiplican las dos matrices modelo para obtener la matriz modelo final.

4.1. La Tierra

Al usar tres texturas distintas para el cómputo del color de cada fragmento, la Tierra requiere un shader de fragmentos distinto. Tiene una textura difusa, una de altitud, y una especular. La de altitud altera las normales de la geometría de la esfera que se usan para calcular la iluminación sobre el fragmento. Para combinar las dos últimas texturas, se debe primeramente calcular la tangente y bitangente de cada normal.

```
tangente = vec3(normal.y, - normal.x, normal.z)
bitangente = cross(tangente, normal)
```

Siguiente, se construye la matriz 3x3 TBN:

```
TBN = mat3(tangente, bitangente, normal)
```

Para calcular la nueva normal según la altura en un cierto punto, se debe multiplicar la matriz TBN por el rgb normalizado que se extraiga de la textura en ese punto, y normalizar:

```
worldNormal = normalize(TBN * normalMap);
```

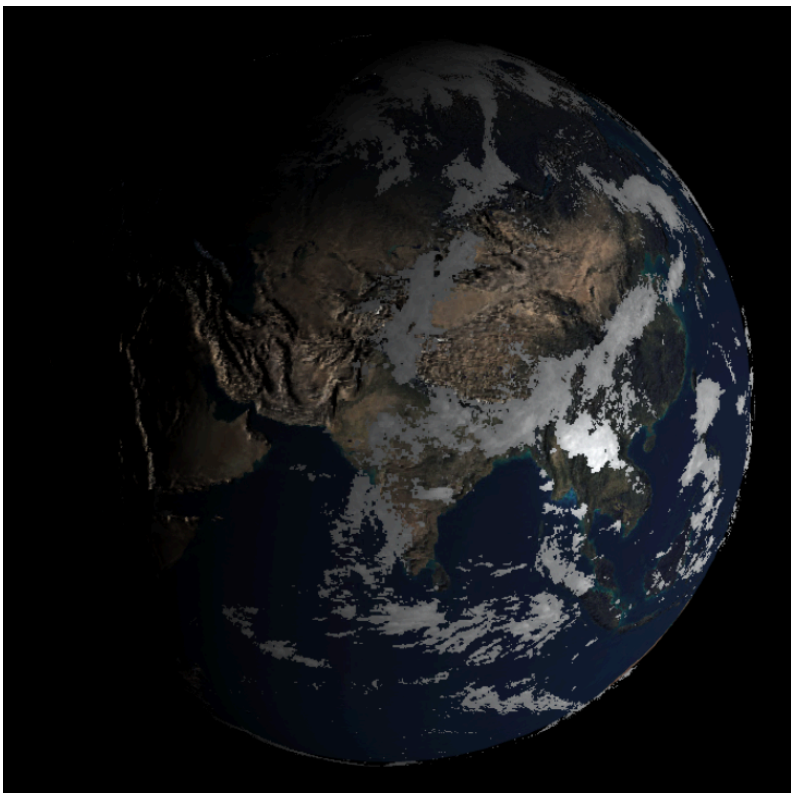

Figura 3: La Tierra con apariencia de relieves.

Como se puede observar, el uso de normales recalculadas según el mapeo de alturas de la textura normal da una apariencia más realista de relieves al afectar cómo se computa la iluminación. Sin embargo, si se mirase de costado se notaría que en realidad la superficie sigue siendo perfectamente plana.

4.2. El anillo de Saturno

El anillo esencialmente se programa como un planeta más, con la única diferencia de que se liga un VAO distinto antes de dibujarlo para que sus vértices, normales, y caras se configuren de tal forma que se genere con la geometría correcta un anillo que se ilumine normalmente, y se texturice bien.

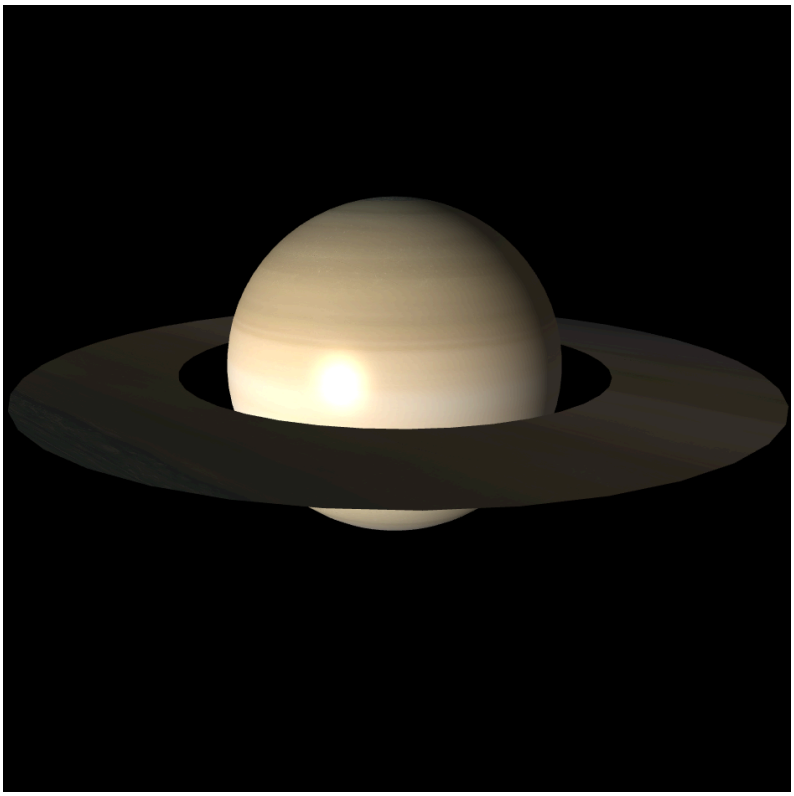

Figura 4: Saturno

4.3. La Luna

A diferencia del resto, la Luna orbita alrededor de la Tierra en vez del Sol. Así que, para no duplicar el código de orbitación, lo hicimos genérico, así que las operaciones para orbitar se realizan relativamente a la posición actual que tenga el cuerpo celestial orbitado.

```
if(dict.hasOwnProperty('orbitR')){
  const orbitAngle = 2 * Math.PI * timeDay / dict.orbitT;

  mat4.rotateY(dict.modelPos, dict.modelPos, orbitAngle)
  mat4.translate(dict.modelPos, dict.modelPos, new Float32Array([AU*dict.orbitR, 0,
0]));
  mat4.multiply(dict.modelPos, celBodies[dict.orbitedBody].modelPos, dict.modelPos)
[...]
```

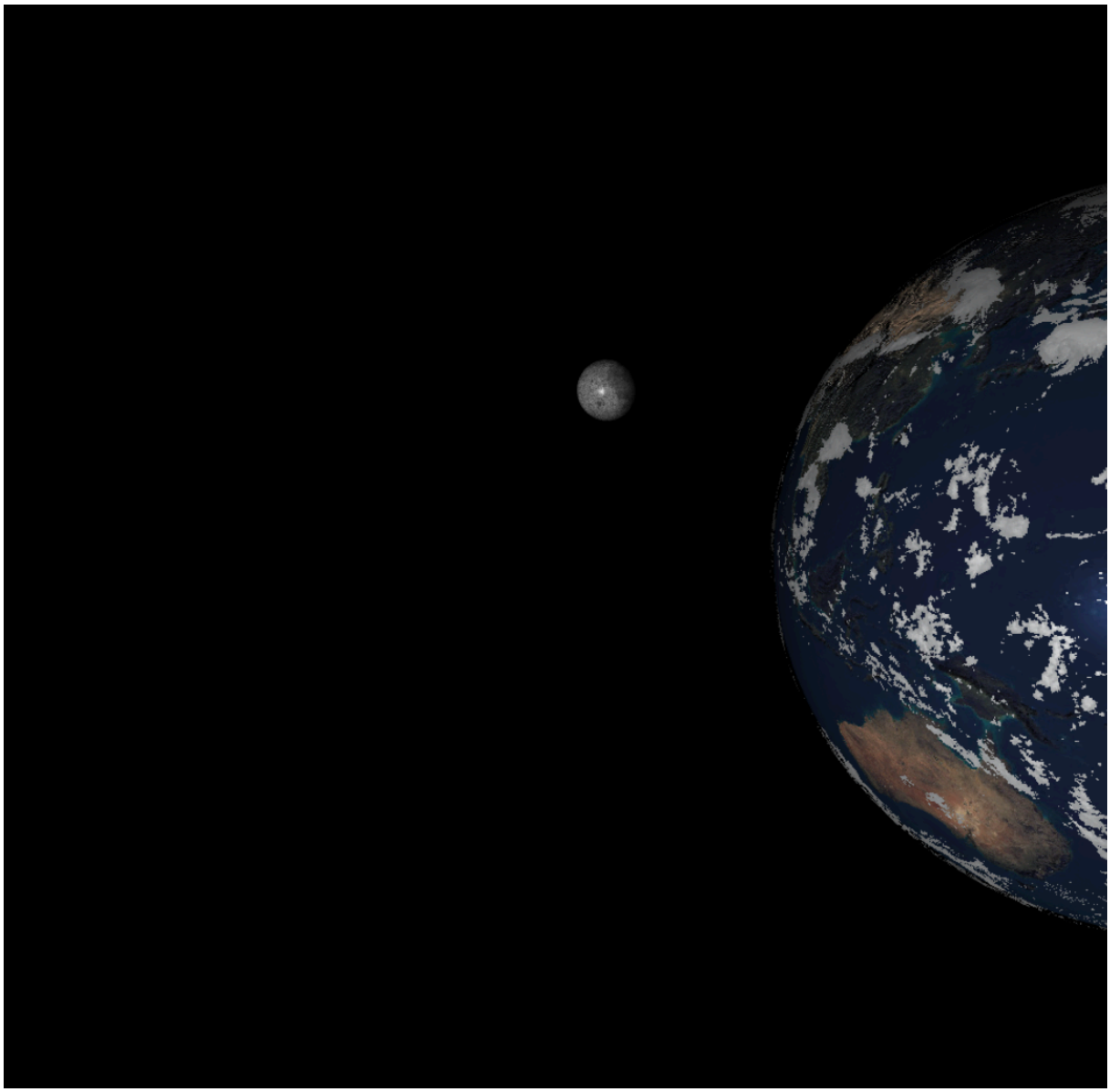

Figura 5: La Luna

Para programar que la misma cara de la Luna siempre apunte a la Tierra, primero, se obtiene la posición del cuerpo que la Luna orbita (la Tierra) extrayéndola de las posiciones 12, 13, 14 del `mat4 celBodies[dict.orbitedBody].modelPos`, y se guarda en `parentPos`

Se calcula un vector dirección desde la Luna hacia la Tierra usando `vec3.subtract(direction, parentPos, thisPos)`, y este vector se normaliza con `vec3.normalize`.

A continuación, se calcula el ángulo de rotación necesario para alinear la cara de la Luna hacia la Tierra usando `Math.atan2` sobre los componentes del vector dirección, y se rota el modelo de la Luna alrededor del eje Y mediante `mat4.rotateY` este mismo ángulo. Esto asegura que el mismo lado de la Luna apunte siempre hacia la Tierra.

4.4. El campo de asteroides

Para implementar un campo de asteroides con una gran cantidad de asteroides sin arruinar la performance con una cantidad excesiva llamadas a la GPU, hace falta usar la instanciación.

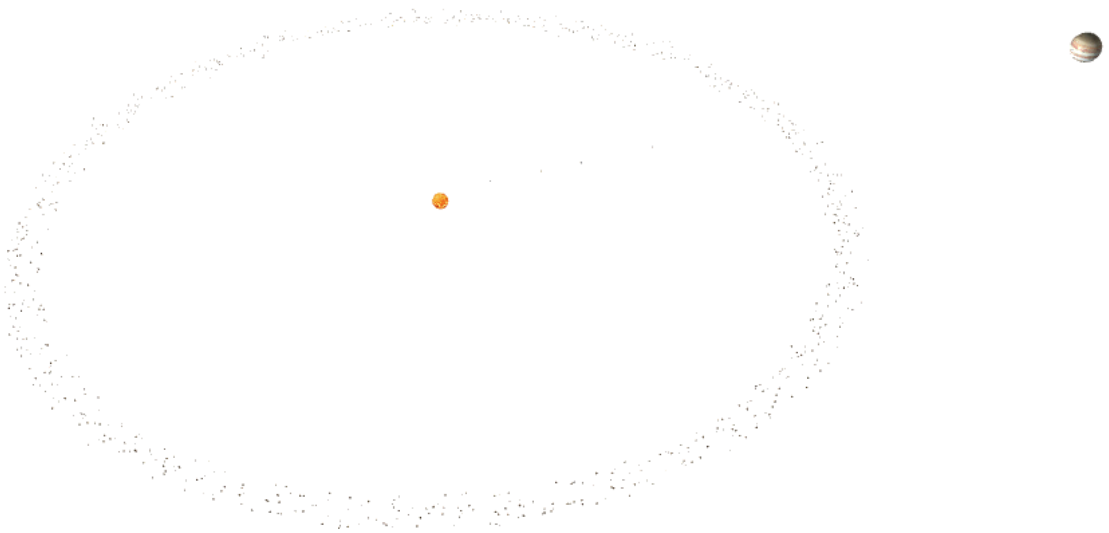

Figura 6: El cinturón de asteroides
(aumentados de tamaño y con un fondo blanco para que sean visibles)

Primero, se definen las N matrices modelo de los N asteroides a instanciar, con algo de aleatorización embebida para darle un aspecto más natural al cinturón de asteroides.

```
const ASTEROID_COUNT = 1000;
let asteroidModels = new Array(ASTEROID_COUNT);
const radius = 50; const offset = 2;

for (let i = 0; i < ASTEROID_COUNT; i++) {
    let model = mat4.create();

    [...] (operaciones)

    asteroidModels[i] = model;
}
```

Siguiente, se asignan todas las posiciones del array models (variable uniforme), con su correspondiente modelo definido en JavaScript. Tras esto, se liga el VertexArray VAOsphere para que tenga el EBO esférico acoplado (y los otros buffers para tener las normales y coordenadas de textura adecuadas), y se llama a especificando el número de instancias a dibujar con la única llamada que se hace (el argumento usado para este parámetro es `ASTEROID_COUNT`).

```
asteroidShaderProgram.bind()
asteroidShaderProgram.setUniform3f("cameraPos", ...cameraPos);
for (let i = 0; i < ASTEROID_COUNT; i++) {
    asteroidShaderProgram.setUniformMat4("models["+i+"]", asteroidModels[i]);
    gl.activeTexture(gl.TEXTURE0);
    asteroidShaderProgram.setUniform1i("diffuseTex", 0);
    gl.bindTexture(gl.TEXTURE_2D, celBodies.asteroids.textures.diffuseTex.id);
}

gl.bindVertexArray(VAOsphere)
gl.drawElementsInstanced(gl.TRIANGLES, sphereGeo.indices.length, gl.UNSIGNED_INT, 0,
ASTEROID_COUNT)
```

Hace falta un shader de vértices especial que aproveche gl_InstanceID para extraer del array-variable uniforme `models` el modelo que le corresponde a la instancia.

```
[...]
uniform mat4 models[1000];
void main(){
    mat4 model = models[gl_InstanceID];
    [...] //lo siguiente es igual al otro de shader de vértices.
}
```

5. Conclusión

Mediante la realización de esta práctica, se pudo conseguir una mayor destreza en el manejo de matrices modelo para describir movimientos circulares, rotaciones sobre el propio eje, y también se consiguió un conocimiento más profundo sobre cómo aprovechar el uso y alternancia entre distintos VAOs para dibujar diferentes formas geométricas. También se aprendió una técnica de optimización: la instanciación. Y, se aprendió a aplicar varias texturas a la vez a un mismo objeto.