Ejercicio 6.3. Iluminación 'pervertex' con shaders.

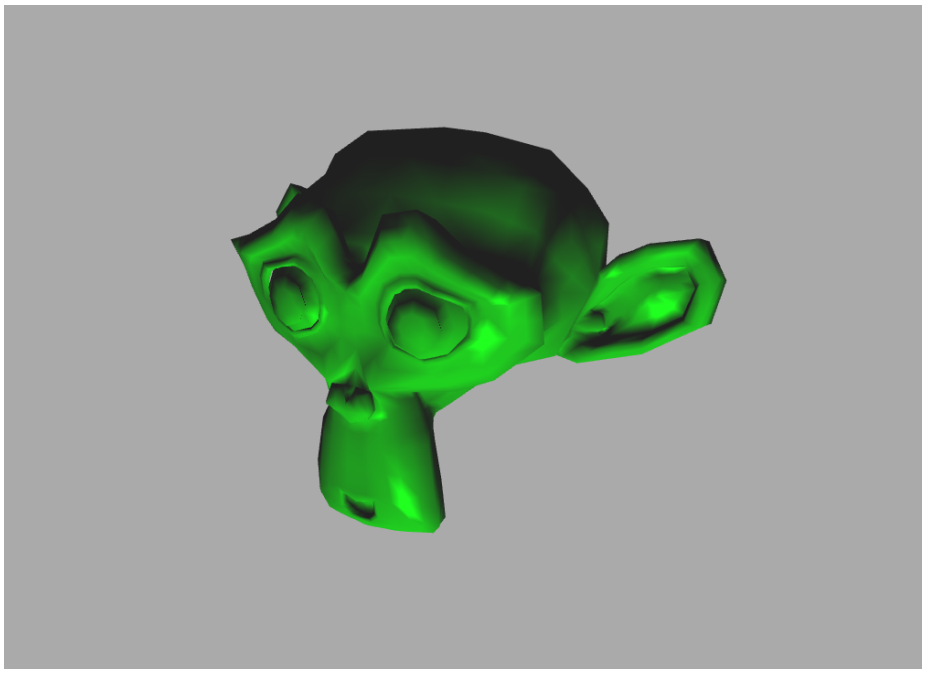

Figura 1: Ejercicio 6.3

El objetivo de este ejercicio es programar una iluminación de tipo *pervertex* por medio de shaders. Específicamente, dada una escena que contiene lo siguiente:

- una luz ambiental
- una luz direccional
- un objeto con un material asignado

debemos calcular los colores en los vértices del objeto, teniendo en cuenta las componentes **ambiental**, **difusa** y **especular**. Como su nombre indica, en el ejercicio realizaremos el llamado *per vertex shading*, es decir, realizaremos

los cálculos de iluminación en cada vértice, y los fragmentos de los triángulos interpolarán los colores de los tres vértices.

Para realizar el ejercicio hay que editar el fichero `ej63_pervertex.js` y el fichero `ej63_pervertex.html`. Veamos cada uno de estos ficheros por separado.

ej63_pervertex.js

Este es el fichero principal con la aplicación. En él se crea una escena y se añade un objeto (*suzanne*), al que se le asocia un material con nuestros shaders.

En el fichero debeis modificar la función `createShaderMaterial` para que cree un material de tipo `THREE.ShaderMaterial` con nuestros *shaders*, y además incluya las siguientes variables de tipo *uniform*:

- `uMaterialColor` (tipo color 'c'). Esta variable contiene el color del material. Inicializadlo con un `THREE.Color` con color verde (`0x00ff00`).
- `uKd` (tipo float 'f'): la componente difusa del material. Inicializadlo con un valor de '0.7'.
- `uKs` (tipo float 'f'): la componente especular del material. Inicializadlo con un valor de '0.7'.
- `uShininess` (tipo float 'f'): el *shininess* del material. Inicializadlo con un valor de '60'.
- `uLightAmbientColor` (tipo color 'c'): el color de la luz ambiente. Inicializadlo con un `THREE.Color` con color gris (`0x202020`).
- `uLightDirection` (vector de tres componentes 'v3'): La dirección de la luz en el espacio de la cámara. Inicializadlo con un `THREE.Vector3` con valores $(0{,}0, 0{,}5, -0{,}3)$
- `uLightColor` (tipo color 'c'): el color de la luz direccional. Inicializadlo con un `THREE.Color` con color blanco (`0xFFFFFF`).

Las variables uniform no las actualizaremos una vez inicializadas, pero se podría hacer fácilmente en la función `render`. Por ejemplo, podemos asignar un color rojo a la luz si escribimos dentro de la función:

```
var redColor = new THREE.Color(0xFF0000);
shaderMaterial.uniforms.uLightColor.value.copy(redColor);
```

ej63_pervertex.html

En el fichero HTML hay que implementar los *shader*s propiamente dichos. No modificaremos el *fragment* shader y centraremos todos nuestros esfuerzos

en programar el *vertex* shader. El *vertex* shader recibe atributos de los vértices y las variables *uniform*. Con ello, debe escribir en la variable *varying* llamada `vColor` el color del vértice según el cálculo de la iluminación.

El shader debe calcular las componentes ambiental, difusa y especular para iluminar el vértice, es decir, la intensidad (color) que recibirá el vértice se calcula según la ecuación:

$$\mathbf{i} = \mathbf{i}_{\text{amb}} + \mathbf{i}_{\text{dif}} + \mathbf{i}_{\text{spec}} \tag{1}$$

Analizemos cada componente por separado:

Componente ambiental: $\mathbf{i}_{\text{amb}}$

Simplemente debemos añadir el color de la luz ambiental, suministrada por medio de la variable *uniform* `uLightAmbientColor`

Componente difusa: $\mathbf{i}_{\text{dif}}$

Utilizaremos la siguiente ecuación difusa:

$$\mathbf{i}_{\text{dif}} = \max(0, \mathbf{n} \cdot \mathbf{l}) \cdot m_{\text{dif}} \cdot \mathbf{s}_{\text{color}} \tag{2}$$

donde:

n: la normal normalizada en el vértice en el sistema de referencia de la cámara.

l: el vector, normalizado y expresado en el sistema de referencia de la cámara, que va desde el vértice hacia la luz.

m_{dif}: la componente difusa del material (escalar).

$\mathbf{s}_{\text{color}}$: el color de la luz.

$\mathbf{n} \cdot \mathbf{l}$ es la multiplicación escalar entre **n** y **l**, y nos da el coseno del ángulo formado por los dos vectores. Pero para ello, hay que cumplir dos condiciones importantes:

1. **n** y **l** deben estar normalizados (función GLSL `normalize`)
2. **n** y **l** deben estar **representados en el mismo sistema de referencia**, en nuestro caso, el sistema de referencia de la cámara.

Componente especular: $\mathbf{i}_{\text{spec}}$

Utilizaremos la siguiente ecuación especular:

$$\mathbf{r} = \frac{2(\mathbf{n}\cdot\mathbf{l})\mathbf{n}-\mathbf{l}}{\|2(\mathbf{n}\cdot\mathbf{l})\mathbf{n}-\mathbf{l}\|} \quad (3)$$

$$\mathbf{i}_{\text{spec}} = \max(0,\mathbf{r}\cdot\mathbf{v})^{m}\cdot m_{\text{spec}}\cdot\mathbf{s}_{\text{color}} \quad (4)$$

donde:

$\mathbf{n}$: la normal normalizada en el vértice en el sistema de referencia de la cámara.

$\mathbf{l}$: el vector, normalizado y expresado en el sistema de referencia de la cámara, que va desde el vértice hacia la luz.

$\mathbf{v}$: el vector, normalizado y expresado en el sistema de referencia de la cámara, que va desde el vértice hacia la cámara.

m_{spec}: la componente especular del material (escalar).

m: *shininess* del material (escalar).

$\mathbf{s}_{\text{color}}$: el color de la luz.

De nuevo, todos los componentes en las ecuaciones deben estar representados en el mismo sistema de referencia (sistema de referencia de la cámara).

Notas generales

Algunas notas generales que os interesa conocer:

- la función GLSL `normalize(v)` normaliza el vector `v`.
- la función GLSL `dot(v1,v2)` realizar la multiplicación escalar entre los vectores `v1` y `v2`.
- la función GLSL `max(a,b)` obtiene el máximo entre `a` y `b`.
- la función GLSL `pow(a, b)` eleva el número `a` al exponente `b` (para aplicar el *shininess* (m) en la ecuación (4).

Cambio de sistema de referencia

En general, vais a necesitar convertir vectores y puntos del sistema de referencia del modelo (local) al sistema de referencia de la cámara. Esto es así porque tanto la posición del vértice como la normal del vértice son atributos que están representados en el sistema de coordenadas del modelo, pero para

realizar los cálculos de iluminación vamos a utilizar el sistema de coordenadas de la cámara (que es donde estan representados tanto la posición de la cámara, así como la dirección de la luz).

En general, recordad que para transformar un vector o punto locales al espacio de la cámara, hay que multiplicarlos por la matrix de *modelview*. Además, la operación es diferente si se trata de un punto o una matriz:

Cambiar el sistema de referencia de un punto Sean:

- `M` una matrix (4×4) de cambio de referencia (de tipo `mat4`).
- `p` un punto de tres coordenadas (de tipo `vec3`)

Para transformar el punto, hay que aplicar:

```
vec3 p_trans = vec3(M * vec4(p, 1.0));
```

fijaros que primero representamos `p` en coordenadas homogeneas añadiendo un 1 en su cuarta componente, ya que se trata de un punto.

Pasar un vector del sistema de coordenadas locales al sistema de coordenadas de la cámara Sean:

- `M` una matrix (4×4) de cambio de referencia (de tipo `mat4`).
- `v` un vector de tres coordenadas (de tipo `vec3`)

Para transformar el vector, hay que aplicar:

```
vec3 v_trans = normalize(vec3(M * vec4(v, 0.0)));
```

fijaros que primero representamos `n` en coordenadas homogeneas añadiendo un 0 en su cuarta componente, ya que se trata de un vector. Después, normalizamos el resultado.