

# Visualización

Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Sistemas Gráficos

Grado en Ingeniería Informática  
Curso 2017-2018

# Contenidos

- 
- 1 Introducción
  - 2 Vistas
  - 3 Luces
  - 4 Materiales
  - 5 Programación de shaders en GPU

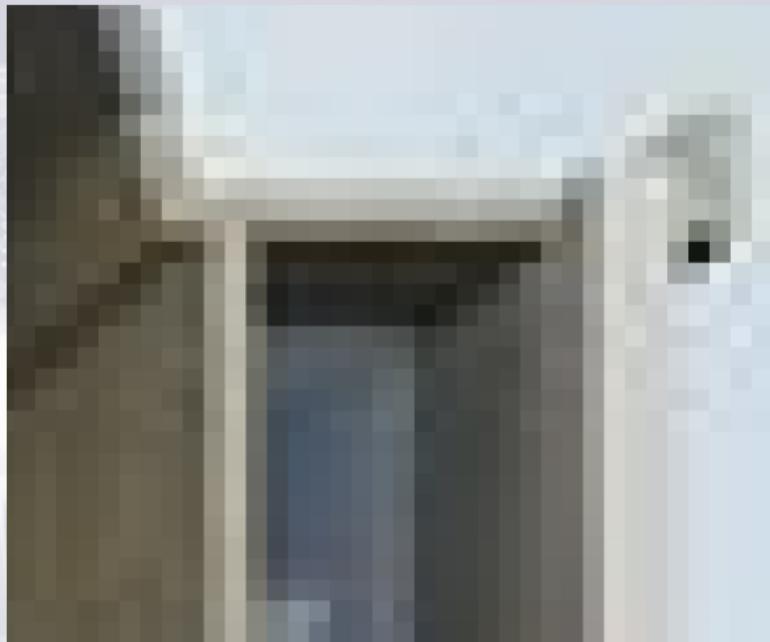
# Objetivos

- Saber crear y modificar vistas de la escena
- Saber crear fuentes de luz
- Saber crear materiales, basados en color y en texturas
- Tener nociones de la programación de shaders en GPU

# Introducción

Captando la realidad

(Recordatorio de Modelo de iluminación)



# Modelo de iluminación

Recordatorio

## Rendering

- Proceso de cálculo desarrollado por un ordenador destinado a generar una imagen o secuencia de imágenes.
- Influyen los elementos siguientes:

**Geometría:** (*Paso de 3-D a 2-D*)

Proyección, ocultación, distorsiones de la perspectiva, etc.

**Fotometría:** (*Luz reflejada por los objetos de la escena*)

Tipo, intensidad y dirección de la iluminación, reflectancia de las superficies, etc.

## Modelo de Iluminación

Conjunto de propiedades físicas de los objetos y de la luz que intervienen en el proceso de rendering.

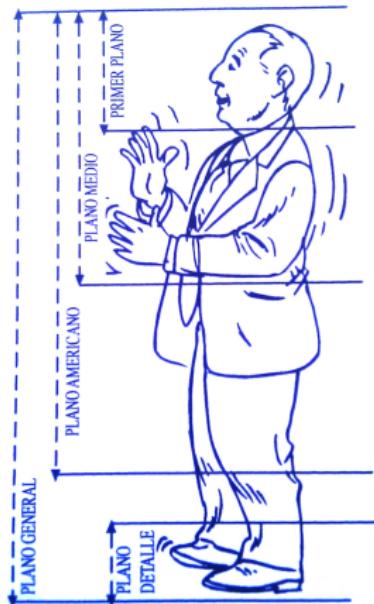
# Vistas

## La cámara

- La cámara ayuda a atraer la atención del espectador
- Puede ser un personaje más
- Se puede animar, cambiar su posición, orientación, etc.
- La animación de la cámara debe realizarse con precaución

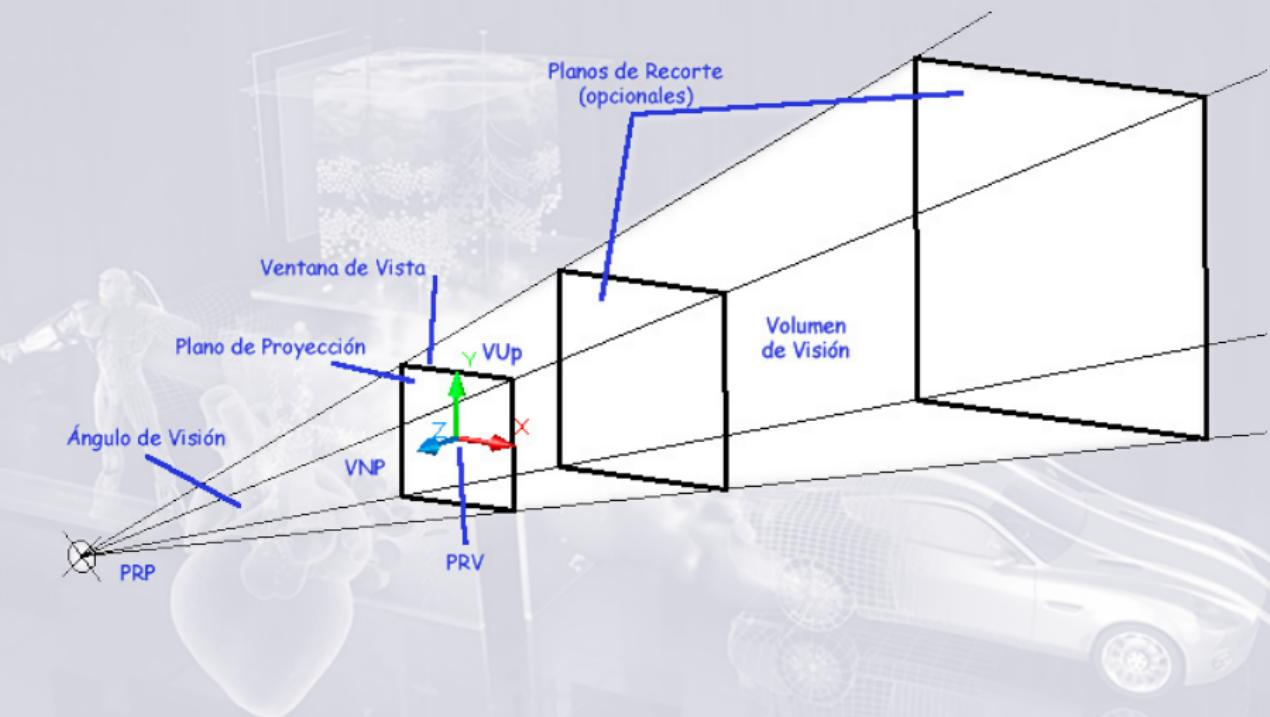


# Planos de encuadre

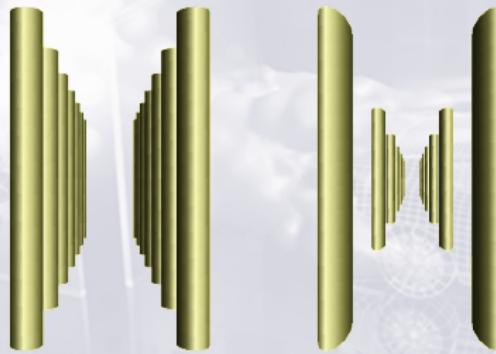
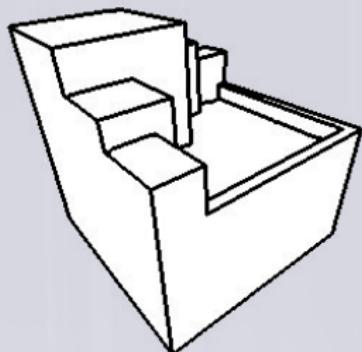
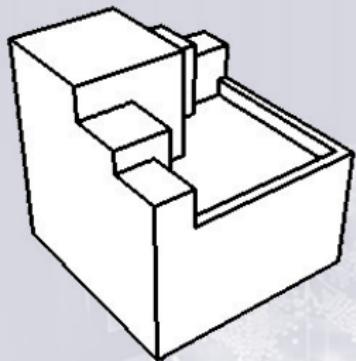


# Parámetros de una cámara

## Vista cónica



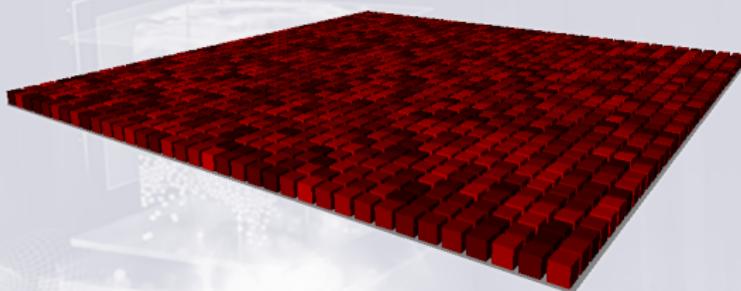
# Parámetros de una cámara



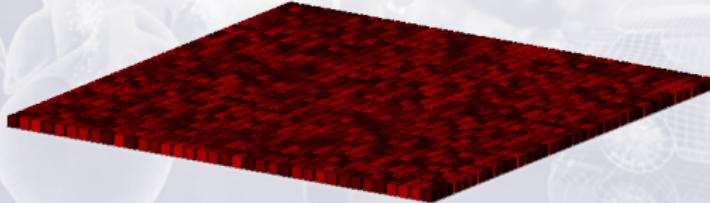
# Vistas

## Three.js

- Se pueden definir vistas
  - ▶ En perspectiva, con `THREE.PerspectiveCamera`



- ▶ Ortográficas, con `THREE.OrthographicCamera`



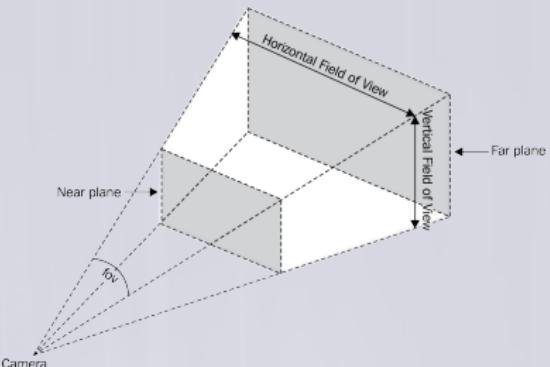
# Vistas

## Creación

- Se crean con:

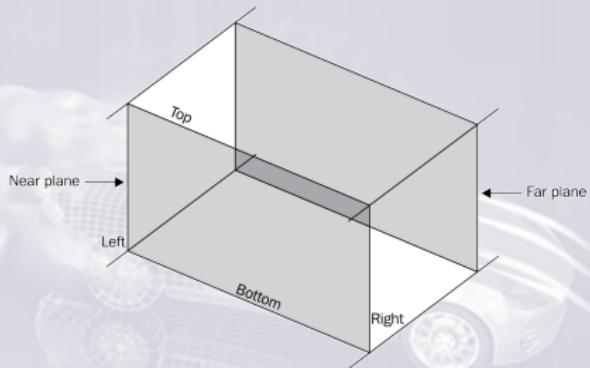
- ▶ THREE.PerspectiveCamera  
(fov, aspect, near, far)
- ▶ THREE.OrthographicCamera  
(left, right, top, bottom, near, far)

# Three.js



- Atributos

- ▶ position
- ▶ lookAt
- ▶ up

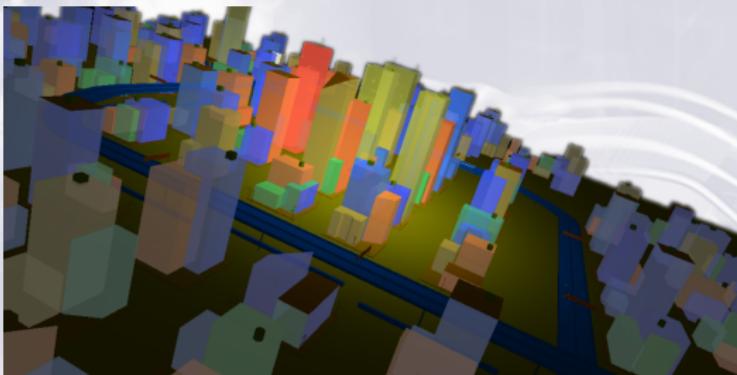


# Movimientos de cámara

## Simulando controles de vuelo

- Requiere la biblioteca [FlyControls.js](#)

Control	Movimiento
Botón izquierdo y central	Avanzar
Botón derecho del ratón	Retroceder
Movimiento del ratón	Giros hacia el puntero
Tecla Q	Inclinarse a la izquierda
Tecla E	Inclinarse a la derecha

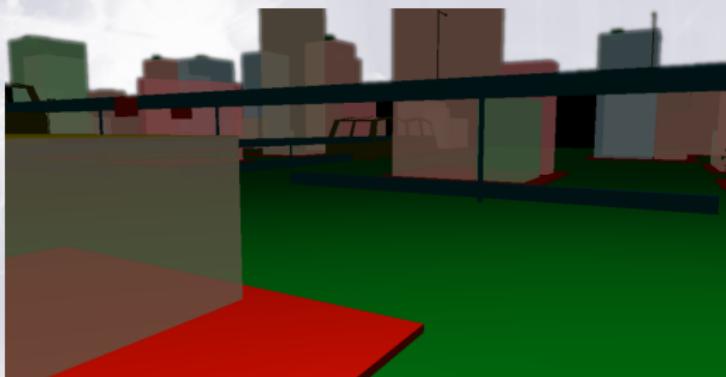


# Movimientos de cámara

## Primera persona

- Requiere la biblioteca `FirstPersonControls.js`

Control	Movimiento
Movimiento del ratón	Mirar alrededor
Cursores	Moverse en esa dirección
Tecla R	Subir
Tecla F	Bajar
Tecla Q	Detener el movimiento

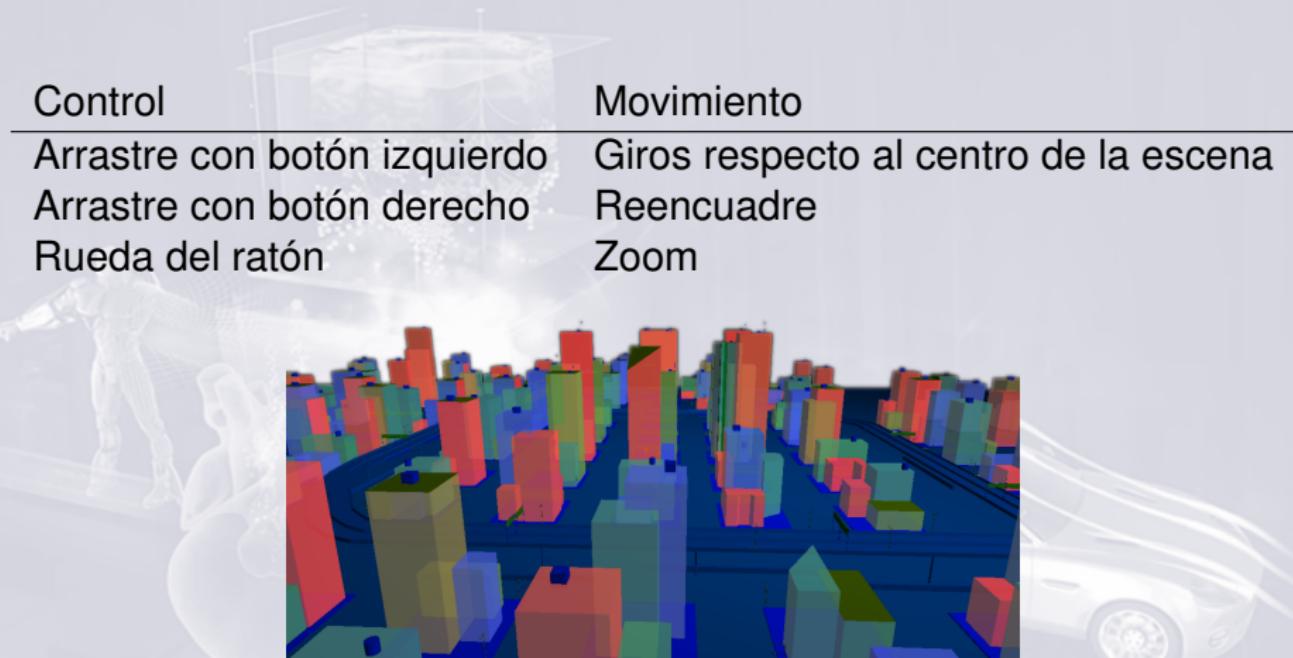


# Movimientos de cámara

## Órbita

- Requiere la biblioteca `TrackballControls.js`

Control	Movimiento
Arrastre con botón izquierdo	Giros respecto al centro de la escena
Arrastre con botón derecho	Reencuadre
Rueda del ratón	Zoom



# Ejemplo

## Ejemplo: Creación y control de una cámara (vuelo)

```
// Se crea la cámara
var camera = new THREE.PerspectiveCamera(45,
    window.innerWidth / window.innerHeight, 0.1, 1000);
camera.position.copy(new THREE.Vector3(100,100,100));
camera.lookAt(new THREE.Vector3(0, 0, 0));

// Se crea el control de vuelo
var clock = new THREE.Clock();
var flyControls = new THREE.FlyControls(camera);
flyControls.domElement = document.querySelector("#WebGL-output");
flyControls.movementSpeed = 25;
flyControls.rollSpeed = Math.PI / 24;
flyControls.autoForward = false;

// En la función de render
var delta = clock.getDelta();
flyControls.update(delta);
requestAnimationFrame(render);
webGLRenderer.render(scene, camera)
```

# Varias vistas en varios viewports

- Se pueden visualizar varios viewports mostrando una cámara distinta en cada viewport



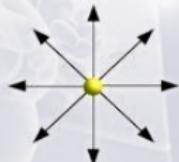
## Vistas: Varias cámaras en varios viewports

```
// Se define una función para agrupar las instrucciones necesarias
// Los parámetros left , top , width , height toman valores entre 0 y 1
funcion renderViewport (escena, camara, left , top , width , height ) {
    var l = left * canvas.width;           var t = top * canvas.height;
    var w = width * canvas.width;          var h = height * canvas.height;
    renderer.setViewport (l,t,w,h);        renderer.setScissor (l,t,w,h);
    renderer.setScissorTest (true );
    camara.aspect = w/h;
    renderer.render (escena , camara );
}

// En la función de visualización se llama a la función anterior
// para todas los viewports que se deseen actualizar
renderViewport (escena, camara1, 0, 0, 1, 1);
renderViewport (escena, camara2, 0.5, 0.5, 0.5, 0.5);
```

# Luces

- Imprescindibles para que se “vean las cosas”
- También usadas para crear “ambiente”
- Una fuente de luz básica se compone de:
  - ▶ Una posición
  - ▶ Un color
  - ▶ Una intensidad
- No obstante, añadiendo restricciones surgen otros tipos de fuentes de luz



Luz puntual



Luz focal



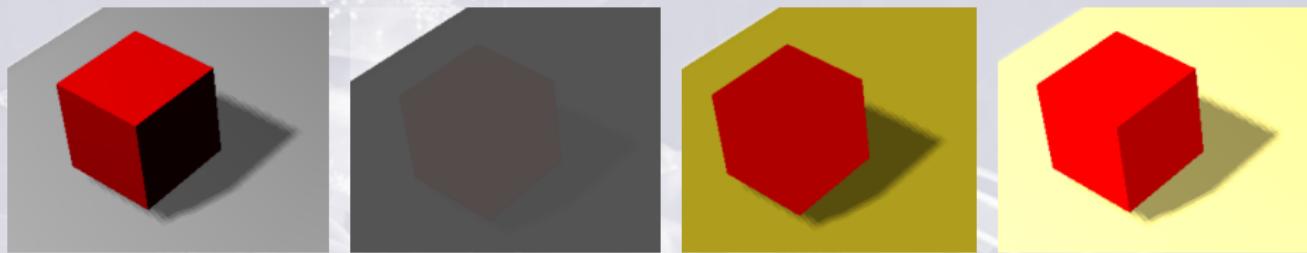
Luz direccional

# Fuentes de iluminación en

## Three.js

### Luz ambiental: AmbientLight

- La luz ambiental ilumina toda la escena por igual
- Se usa para simular la luz indirecta
- Permite añadir fácilmente una dominante de color
- Constructor: `AmbientLight (color, intensity)`



### Ejemplo: Definición de la luz ambiental

```
var ambientLight = new THREE.AmbientLight (0x0c0c0c);  
scene.add (ambientLight);
```

# Luz puntual

## PointLight

- Desde una determinada posición, ilumina en todas direcciones
- Constructor:  
`PointLight (color, intensity, distance, decay)`
  - ▶ `intensity`, por defecto es 1
  - ▶ `distance`, la intensidad es cero a partir de esa distancia  
si `distance == 0`, valor por defecto, no se toma en cuenta
  - ▶ `decay`, la caída: 1 (por defecto), lineal; 2, cuadrática
- Atributos:
  - ▶ `position`, sitúa la luz en dicha posición
  - ▶ `visible`, un booleano que enciende/apaga la luz

### Ejemplo: Definición de la luz puntual

```
var pointLight = new THREE.PointLight (0xfcfcfc, 1, 50);  
pointLight.position.set (2, 3, 5);
```



# Luz focal

## SpotLight

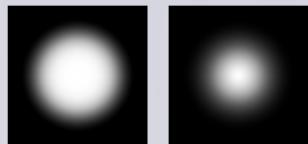
- Desde una posición, ilumina en una dirección y ángulo
- Constructor:

`SpotLight (color, intensity, distance,  
angle, penumbra, decay)`

- ▶ angle, desde su dirección, por defecto es  $\pi/3$  (máximo  $\pi/2$ )
- ▶ penumbra, cómo decrece la intensidad desde el centro.  
Toma valores entre 0 (por defecto) y 1.

- Atributos:

- ▶ target, el objeto a donde apunta la luz (debe estar en la escena)

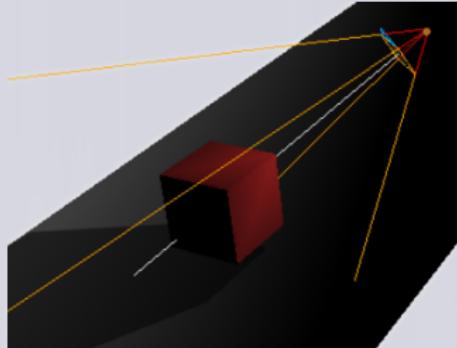


## Ejemplo: Definición de la luz puntual

```
var spotLight = new THREE.SpotLight (0xfcfcfc);
spotLight.position.set (2, 30, 5);
var target = new THREE.Object3D ();
target.position.set (2, 0, 5);    spotLight.target = target;
scene.add (spotLight);    scene.add (target);
```

# SpotLight

- Puede proyectar sombras
- Elementos a configurar:
  - ▶ El renderer para que las calcule
  - ▶ La luz para que las proyecte
  - ▶ Los objetos que las proyectan
  - ▶ Los objetos que las reciben



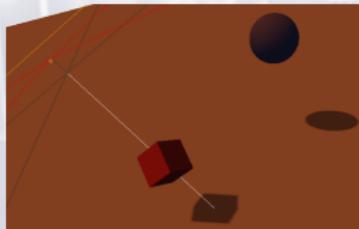
## Ejemplo: Configuración para proyectar sombras

```
renderer.shadowMapEnabled = true;  
  
spotLight.castShadow = true;  
spotLight.shadow.camera.near = 2;  
spotLight.shadow.camera.far = 200;  
spotLight.shadow.camera.fov = 30;  
spotLight.shadow.camera.mapSize.width = 1024;  
spotLight.shadow.camera.mapSize.height = 1024;  
  
sphere.castShadow = true;  
plane.receiveShadow = true;
```

# Luz direccional

## DirectionalLight

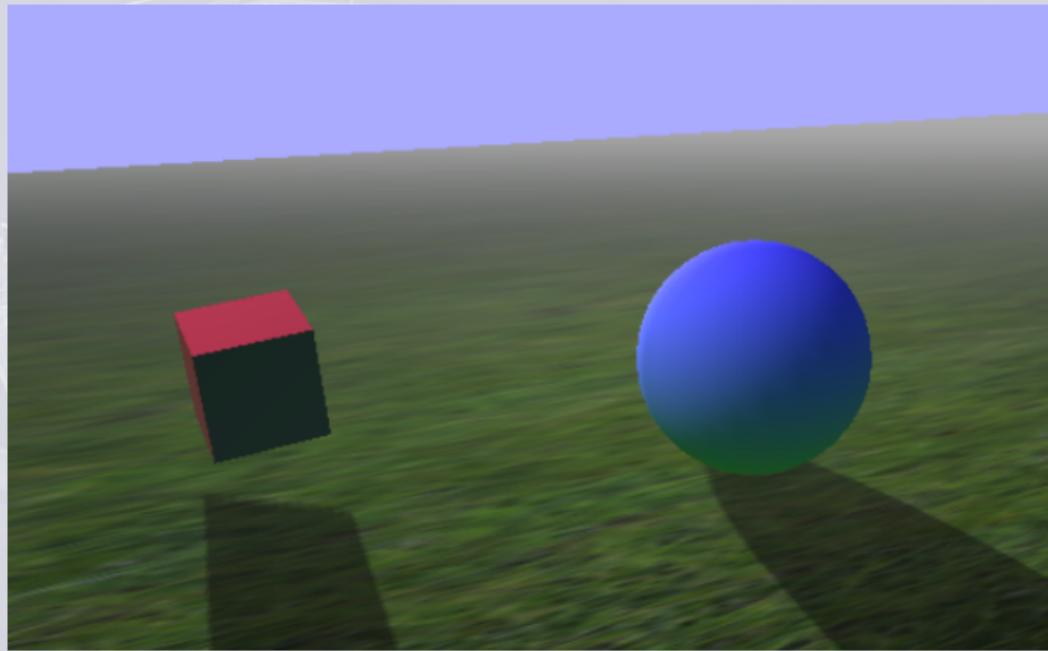
- Emite rayos paralelos entre sí según una dirección
- Constructor:  
`DirectionalLight (color, intensity)`
  - ▶ La dirección se establece con los atributos `position` y `target`
  - ▶ Si no hay `target` los rayos se dirigen al `(0, 0, 0)`
- También proyecta sombras, con los mismos atributos, salvo ...
  - ▶ `shadow.camera.fov`
- ... que se sustituye por:
  - ▶ `shadow.camera.left`, `shadow.camera.right`,  
`shadow.camera.top`, `shadow.camera.bottom`



# Luz natural en exteriores

## HemisphereLight

- Simula la luz difusa del cielo y la que pueda reflejar el suelo



# HemisphereLight

- Constructor:

```
HemisphereLight ( sKyColor,groundColor,intensity)
```

- Además hay que poner la luz direccional que represente al sol

## Ejemplo: Uso de luz del cielo y reflejo del suelo

```
// Suelo: Un plano con textura que recibe sombras
var loader = new THREE.TextureLoader();
var textureGrass = loader.load ("grass.jpg");
var planeMat = new THREE.MeshLambertMaterial ({map: textureGrass});

var plane = new THREE.Mesh (planeGeometry , planeMat);
plane.receiveShadow = true;
// Sol: Una luz direccional
var sun = new THREE.DirectionalLight (0xffffff);

// Luz del cielo y reflejos del suelo
var skyGroundLight = new THREE.HemisphereLight (0xa0a0ff , 0xa0a0ff);
```

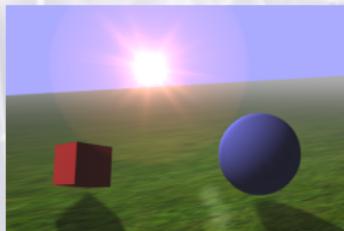
# Dibujado del disco del sol

## LensFlare

- Se puede simular el disco del sol mediante texturas
- O añadir el efecto de reflejo del sol en la lente
- Constructor:

```
LensFlare ( texture, size, distance, blending, color)
```

- ▶ texture, la textura a usar
- ▶ size, el tamaño en píxeles, -1 para tomar el tamaño de la imagen
- ▶ distance, la distancia entre la fuente de luz (0) y la cámara (1)
- ▶ blending, cómo se combina la textura con el resto
- ▶ color



# LensFlare

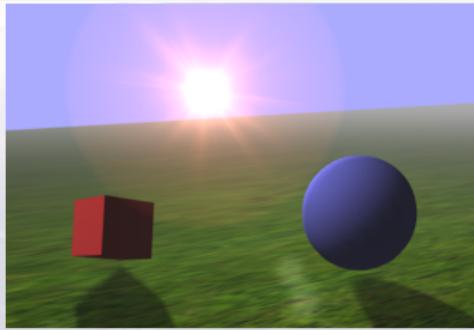
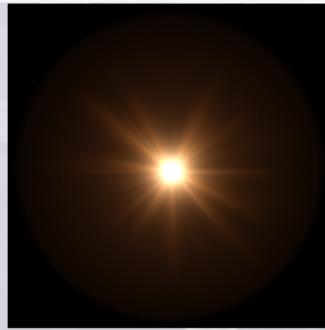
## Ejemplo: Disco del sol y reflejo en la lente

```
var textureSun = loader.load ("lensflareSun.png");
var textureFlare = loader.load ("lensflare.png");

var flareColor = new THREE.Color (0xffaaee);
var lensFlare = new THREE.LensFlare(textureSun, 350, 0.0,
    THREE.AdditiveBlending, flareColor);

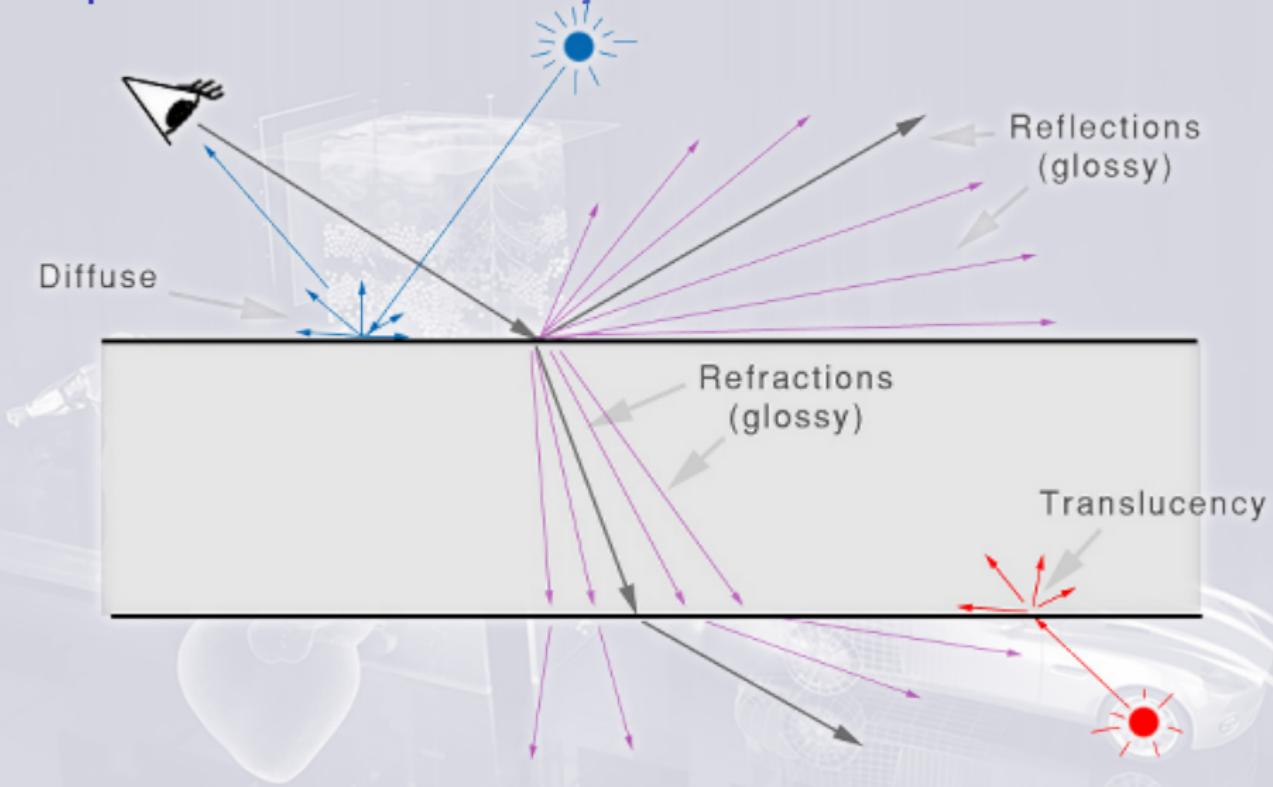
lensFlare.add(textureFlare, 60, 0.6, THREE.AdditiveBlending);
lensFlare.add(textureFlare, 70, 0.7, THREE.AdditiveBlending);

lensFlare.position.copy (directionalLight.position);
```

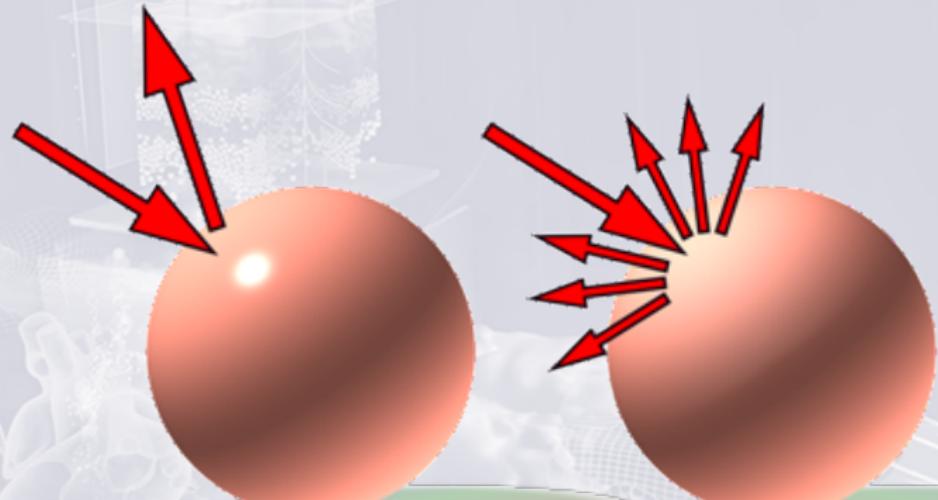


# Materiales

## Comportamiento de la luz en los objetos



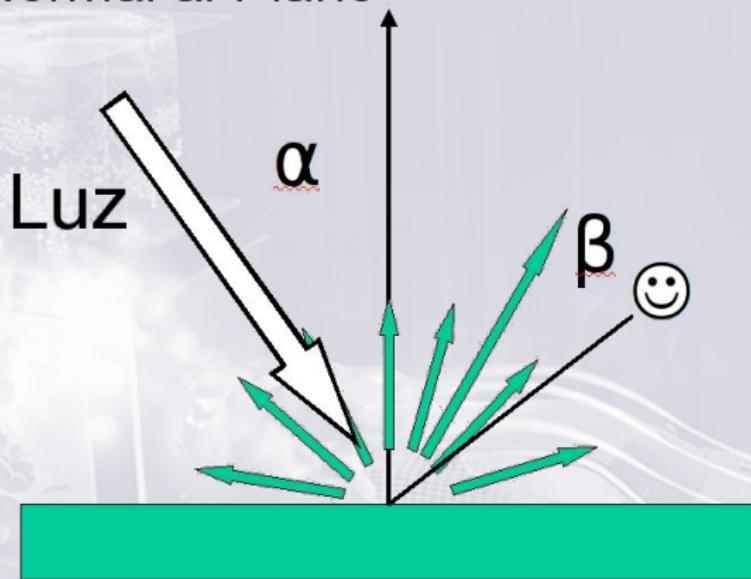
# Componente Especular vs. Difusa



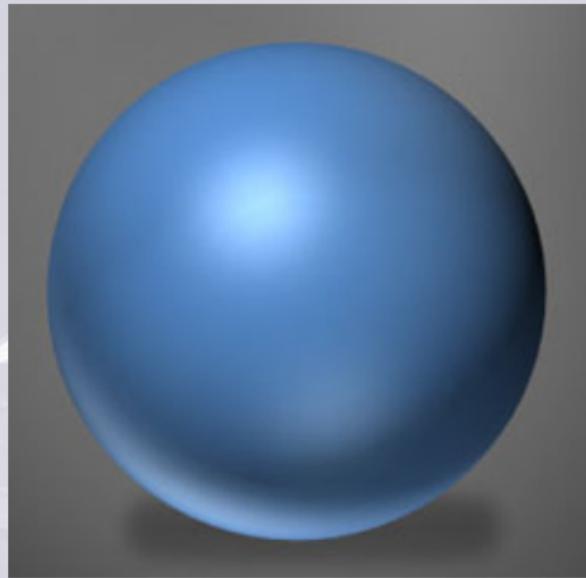
# Componente Especular vs. Difusa

Cálculo del color en un punto

Vector Normal al Plano



# Componentes *ambiental*, difusa y especular



- Componente Ambiental
- Componente Difusa
- Componente Especular



# Materiales en Three.js

- Three.js es una capa sobre WebGL
- El uso de programas shaders es obligatorio
- Cada shader implementa un modelo de iluminación
  - ▶ Un shader recibe:
    - ★ Los vértices de cada polígono
    - ★ Otra información necesaria según el modelo de iluminación a aplicar:
      - Vector normal
      - Coeficiente de reflexión difusa
      - Características de las luces, etc.
  - ▶ Calcula el color que le corresponde a un determinado “pixel”
- En Three.js, cada material predefinido contiene su propio shader
- Pueden programarse shaders propios

## Clase Material

## Three.js

- Clase base para el resto de materiales
- Atributos comunes
  - ▶ name: Permite identificar el material con un nombre
  - ▶ transparent: Boolean.
    - ★ Si false, la figura será opaca.
    - ★ Si true, la figura será transparente según el atributo de opacidad
  - ▶ opacity: Indica el nivel de transparencia.  
Un valor entre 0.0 (invisible) y 1.0 (opaco)
  - ▶ visible: Boolean. Permite hacer invisible una figura (false)
  - ▶ side: Indica en qué lados de una cara se aplica el material
    - ★ THREE.FrontSide, THREE.BackSide, THREE.DoubleSide

# Materiales básicos

## Clase MeshBasicMaterial

- No tiene en cuenta la iluminación
- Muestra toda la figura de un color plano
- Permite mostrar la figura en modo alambre
- Constructor:

`MeshBasicMaterial ( parámetros )`

- ▶ `color`, se especifica en hexadecimal
- ▶ `wireframe`, boolean
- ▶ `wireframeLineWidth`

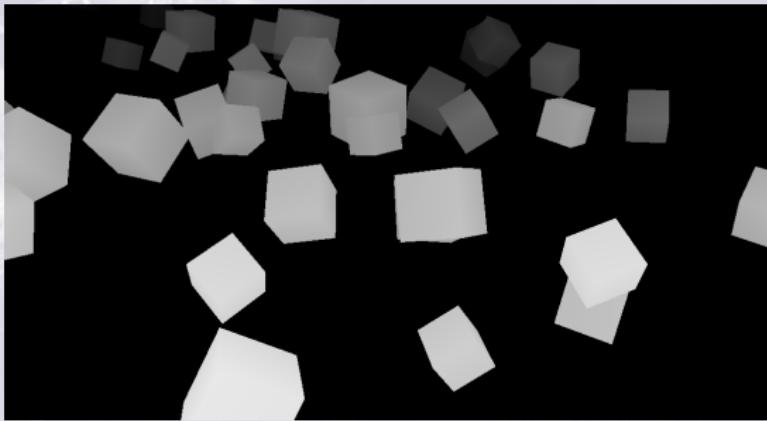


## Ejemplo: Definición de un material *modo alambre*

```
var parameters = { color: 0x0000FF,
                   wireframe: true, wireframeLineWidth: 2 };
var wireframeMat = new THREE.MeshBasicMaterial (parameters);
```

## Clase MeshDepthMaterial

- No tiene en cuenta la iluminación ni otros atributos
- Colorea la figura en un gris plano según la distancia a la cámara
- El efecto se controla mediante 2 atributos de la cámara
  - ▶ near y far determinan el rango de aplicación
- Se usa combinado con otros materiales

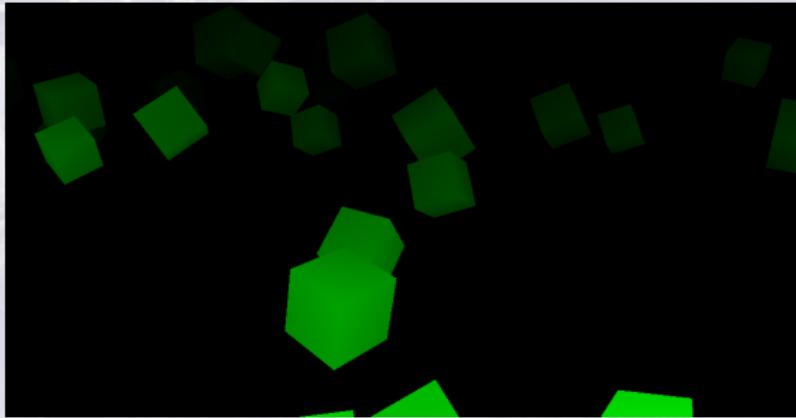


# Clase MeshDepthMaterial

Combinación con otro material

Ejemplo: Combinando MeshDepthMaterial con otro material

```
var colorMat = new THREE.MeshBasicMaterial ( { color: 0x00ff00 ,  
    transparent: true , blending: THREE.MultiplyBlending } );  
var depthMat = new THREE.MeshDepthMaterial ();  
var cube = new THREE.SceneUtils.createMultiMaterialObject ( [  
    cubeGeometry , [ colorMat , depthMat ] ]);  
cube.children[1].scale.set ( 0.99, 0.99, 0.99 );
```



## Clase MeshNormalMaterial

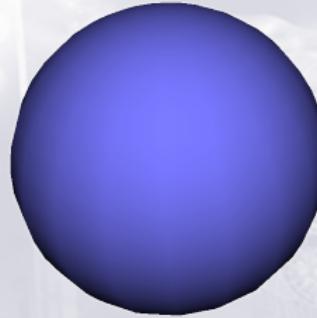
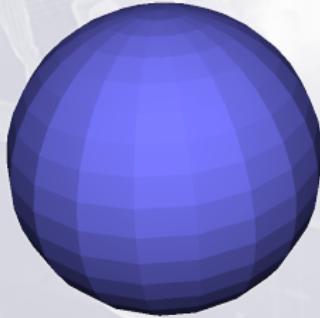
- Colorea cada vértice según su normal
- Permite configurar el sombreado con el atributo `shading`
  - ▶ `THREE.FlatShading`
  - `THREE.SmoothShading`



# Materiales avanzados

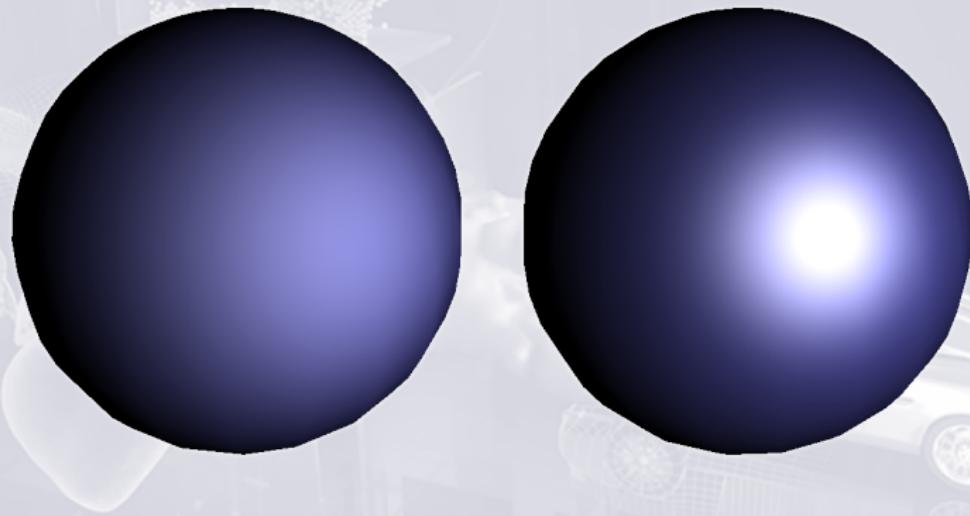
## Clase MeshLambertMaterial

- Material basado en Lambert: componentes difusa y emisiva
- Atributos
  - ▶ `color`: Define el color difuso de la figura
  - ▶ `emissive`: Permite simular que la figura emite luz.  
Por defecto es negro.
- También dispone de los otros atributos vistos anteriormente:  
`shading`, `blending`, `opaque`, `transparent`, etc.



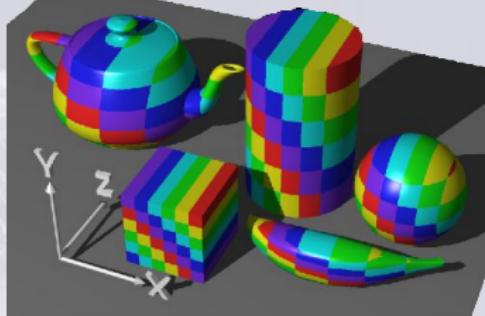
## Clase MeshPhongMaterial

- Material basado en Phong: Lambert + componente especular
- Atributos añadidos
  - ▶ specular: Define el color de los brillos
  - ▶ shininess: La intensidad del brillo. Por defecto, 30.

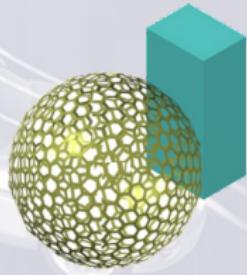
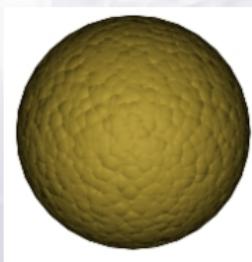
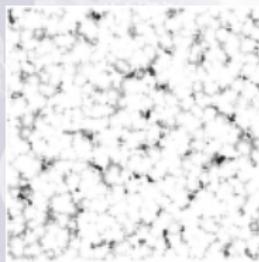


# Texturas

- Permiten usar una imagen para colorear un objeto



- O modificar otro tipo de características



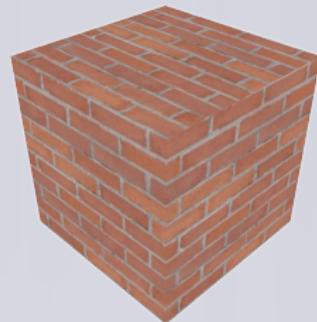
# Texturas

## Three.js

- Se cargan con la siguiente clase

```
var loader = new THREE.TextureLoader();
var textura = loader.load ("imagen.jpg");
```

- Se pueden usar imágenes png, gif, o jpg
- Los mejores resultados se obtienen con imágenes cuadradas y resolución potencia de 2 (256x256, 512x512, etc.)
  - ▶ Las texturas mipmap se generan automáticamente
- Una vez cargada, se asigna al canal del material correspondiente
  - ▶ El canal difuso se denomina map



### Texturas: Uso de una textura en el canal difuso

```
var loader = new THREE.TextureLoader ();
var textura = loader.load ("imagen.jpg");
var material = new THREE.MeshPhongMaterial ();
material.map = textura;
```

# Texturas

## Configuración de las texturas

- Repeticiones

- ▶ Se indican en los atributos `wrapS` y `wrapT`
- ▶ Puede tener dos modos
  - ★ `THREE.RepeatWrapping` El número de repeticiones se indica con el método `repeat.set (n,m)`
  - ★ `THREE.ClampToEdgeWrapping`
- ▶ Un cambio de modo exige solicitar una actualización  
`material.map.needsUpdate = true;`
- ▶ Un cambio en el número de repeticiones se actualiza automáticamente



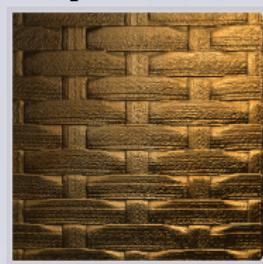
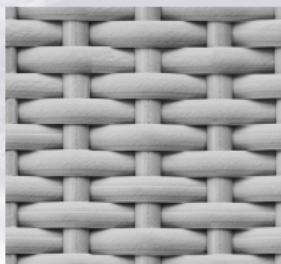
## Texturas: Ajuste de las repeticiones

```
material.map.wrapS = material.map.wrapT = THREE.RepeatWrapping;  
material.map.repeat.set (5, 3);  
material.map.offset.set (-0.2, 0.3); // Para reencuadrar la textura
```

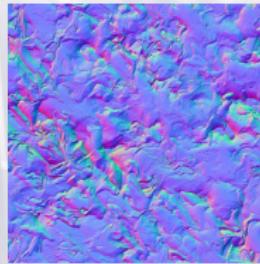
# Texturas en otros canales

## Relieve

- Se puede simular relieve mediante 2 canales
  - ▶ bumpMap, se usa una imagen en tonos de gris
    - ★ Se ajusta la magnitud con el escalar `bumpScale`



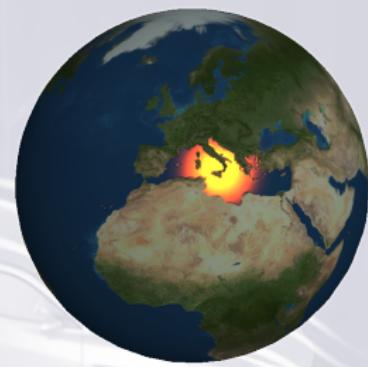
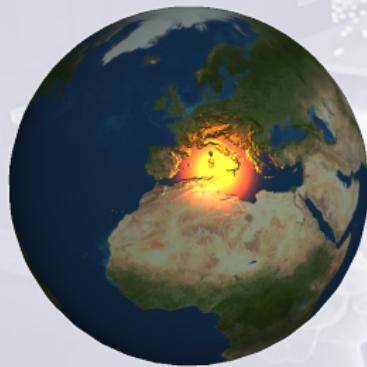
- ▶ normalMap, se usa una imagen que almacena vectores
  - ★ Se ajusta la magnitud con el vector2D `normalScale`



# Texturas en otros canales

## Especular

- Permite que se aplique la componente especular de manera distinta en cada parte de la geometría
- Se asigna un mapa en tonos de gris al canal `specularMap`



# Texturas en otros canales

## Transparencia

- Se asigna un mapa en tonos de gris al canal alphaMap



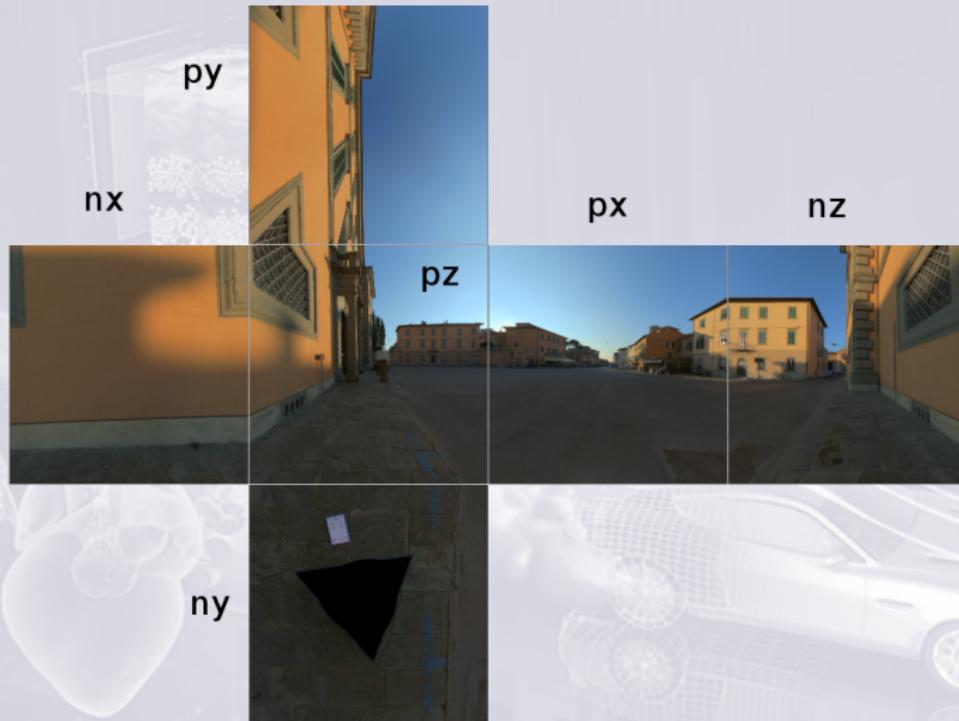
## Ejemplo: Visualización de interior y exterior con transparencias

```
var matExt = new THREE.MeshPhongMaterial ();
// se configura de la manera habitual y además ...
matExt.alphaMap = unaTexturaAlfa;
matExt.transparent = true;    matExt.side = THREE.FrontSide;
// ahora se hace el material para el interior
matInt = matExt.clone();    matInt.side = THREE.BackSide;
// Se crea una figura multimaterial, el interior primero
var figura = THREE.SceneUtils.createMultiMaterialObject (geometria,
    [matInt, matExt]);
```

# Texturas en otros canales

## Entorno

- Se usa una caja con 6 texturas en el interior, una por cara



# Texturas como entorno

## Carga de las texturas

- Las texturas se cargan en una clase específica para ello

### Ejemplo: Carga de texturas para el entorno

```
var path = "textures/cube/pisa/";
var format = '.png';
var urls = [
    path + 'px' + format, path + 'nx' + format,
    path + 'py' + format, path + 'ny' + format,
    path + 'pz' + format, path + 'nz' + format
];
var textureCube = new THREE.CubeTextureLoader().load( urls );
```

# Texturas como entorno

## Shader para el entorno

- Hay que usar un Shader específico

### Ejemplo: Shader para un entorno con texturas

```
var shader = THREE.ShaderLib[ "cube" ];
shader.uniforms[ "tCube" ].value = textureCube;

var material = new THREE.ShaderMaterial( {
    fragmentShader: shader.fragmentShader,
    vertexShader: shader.vertexShader,
    uniforms: shader.uniforms,
    depthWrite: false,
    side: THREE.BackSide
} );

environmentMesh = new THREE.Mesh (
    new THREE.BoxGeometry( 100, 100, 100 ), material );
```

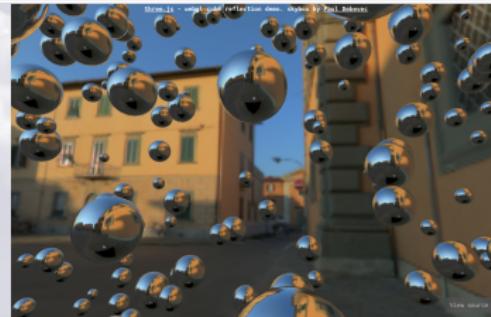
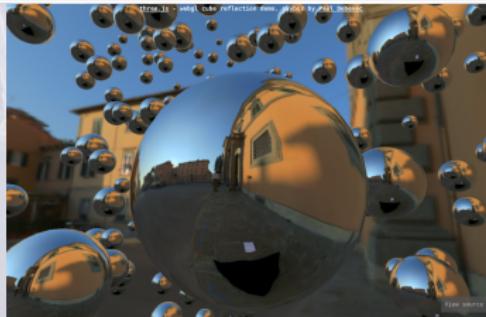
# Texturas en otros canales

## Reflexión del Entorno

- Se debe usar la CubeTexture del entorno
- No hace falta un shader específico
- Pero sí poner dicha textura en el canal envMap

### Ejemplo: Reflejo del entorno

```
var material = new THREE.MeshBasicMaterial( { color: 0xffffffff ,  
envMap: textureCube } );
```



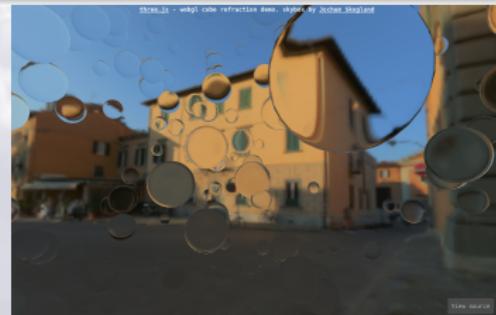
# Texturas en otros canales

## Refracción del Entorno

- Se usa también la CubeTexture del entorno
- Hay que activar el mapeo de refracción en la textura
- E indicar el índice de refracción

## Ejemplo: Reflejo del entorno

```
var textureCube = new THREE.CubeTextureLoader().load( urls );
textureCube.mapping = THREE.CubeRefractionMapping;
var material = new THREE.MeshBasicMaterial( { color: 0xffffffff,
    envMap: textureCube, refractionRatio: 0.95 } );
```



# Usar un vídeo como textura

- Se añade el vídeo en un elemento HTML
- Se toma desde ahí para asignarlo como textura a un material

## Ejemplo: Usar un vídeo como textura

```
// La parte en HTML
<video id="video" style="display: none;"  
       src="ruta/y/fichero" autoplay="true">  
</video>  
  
// La parte en javascript  
var video = document.getElementById ('video');  
texture = new THREE.Texture (video);  
texture.generateMipmaps = false; // si el vídeo no es cuadrado  
// Se usa de la manera habitual
```

# Ejemplo

## El planeta Tierra



# El planeta Tierra

## Fuentes de luz

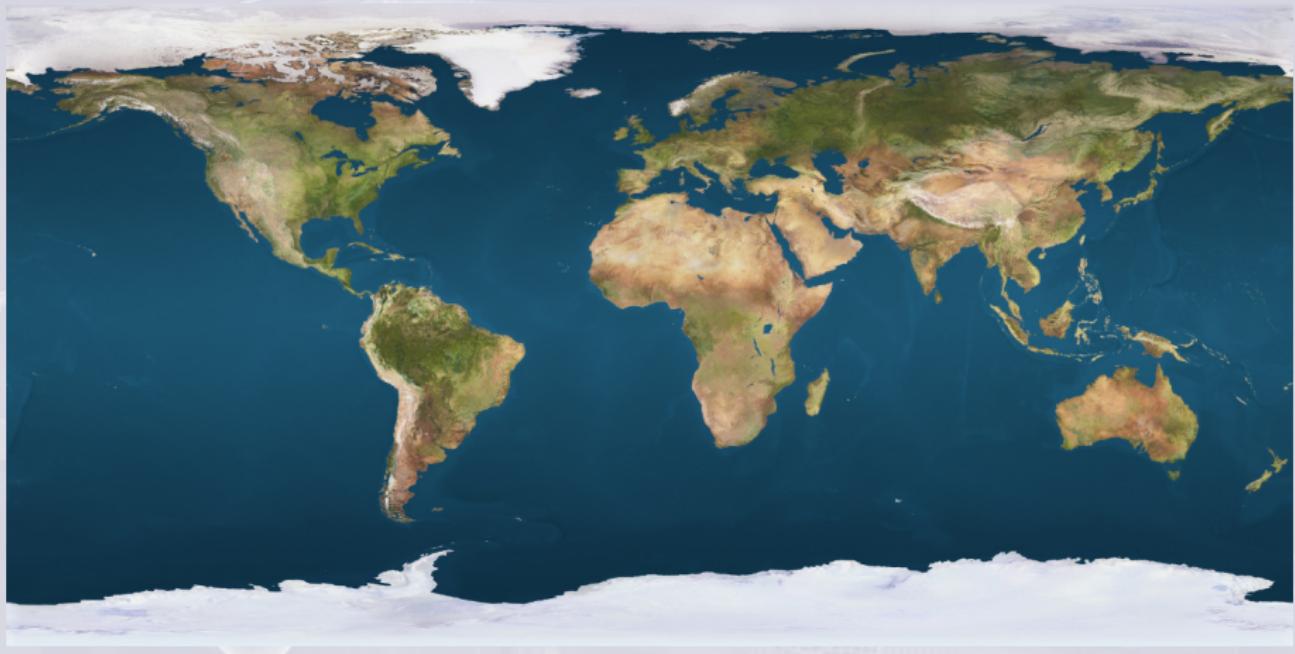
- Una ambiental y una direccional



# El planeta Tierra

## Textura difusa

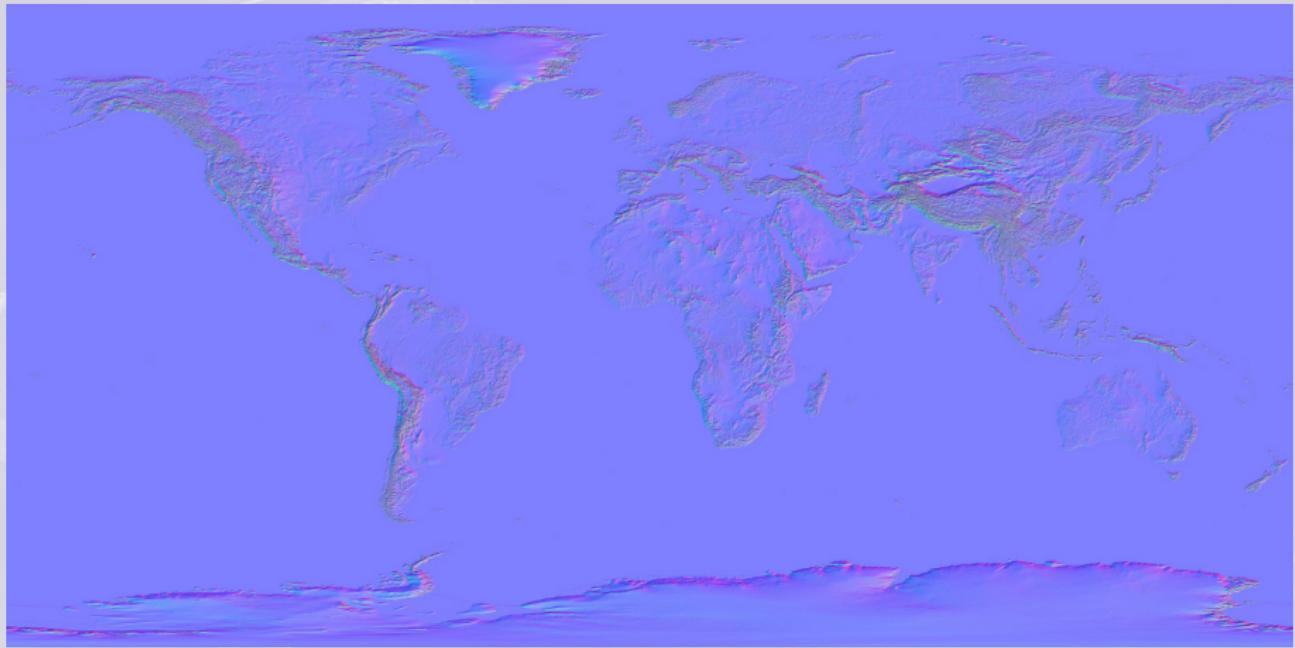
- Asignada en el atributo map



# El planeta Tierra

## Mapa de normales para el relieve

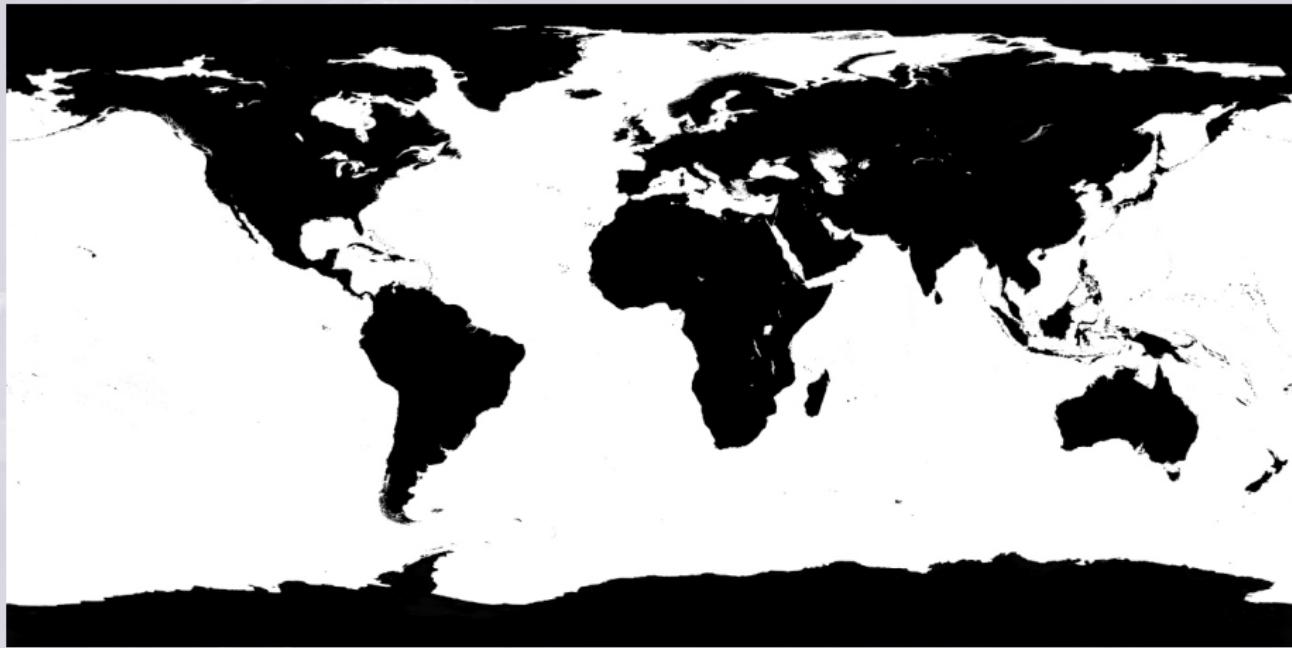
- Asignada en el atributo `normalMap`



# El planeta Tierra

Mapa en tonos de gris para la reflectividad

- Asignada en el atributo specularMap



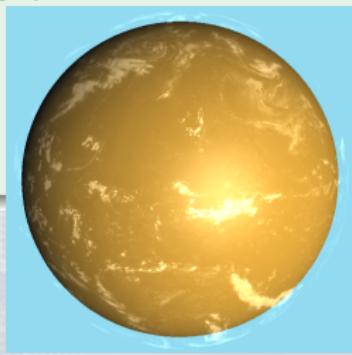
# El planeta Tierra

## Añadido de las nubes

- Se ha usado otra esfera
  - ▶ Con un radio un 15 % mayor
  - ▶ Con una velocidad de rotación un 10 % mayor
  - ▶ Con un textura difusa de nubes sobre un fondo transparente

## El planeta Tierra: Nubes

```
var cloudTexture = loader.load("ruta/nubes.png");
var cloud = new THREE.MeshPhongMaterial();
cloud.map = cloudTexture;
cloud.transparent = true;
cloud.opacity = 0.5;
cloud.blending = THREE.AdditiveBlending;
```



# El planeta Tierra



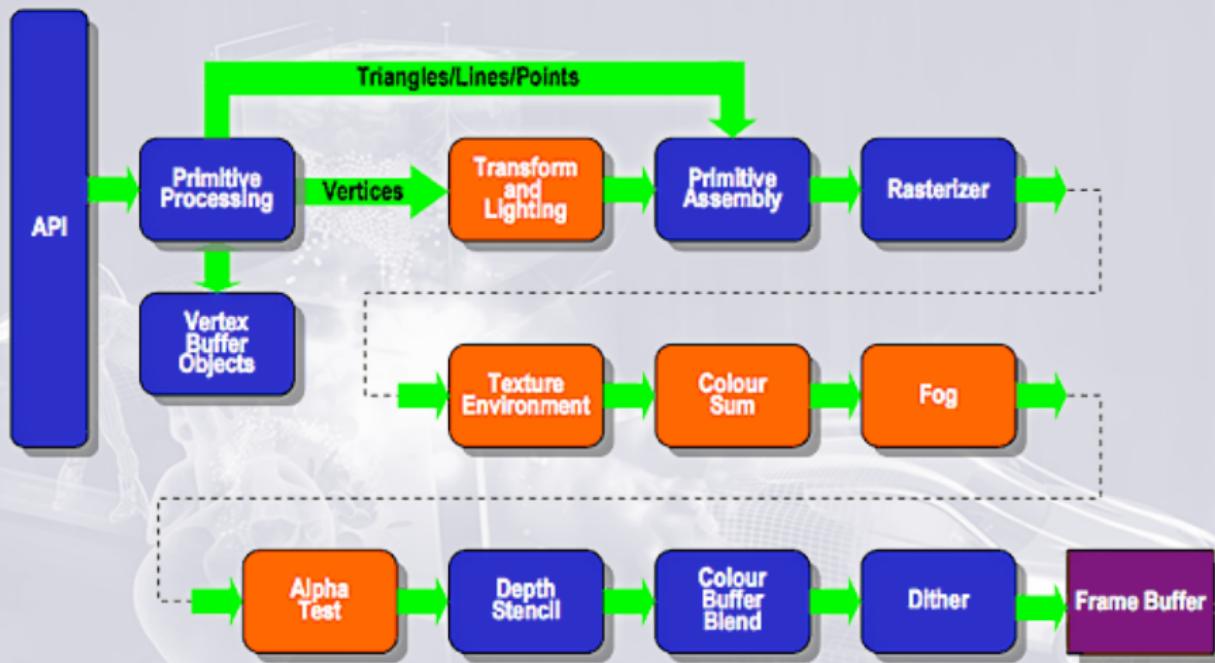
# Introducción

- GPU: Graphics Processing Unit
  - ▶ Coprocesador dedicado al procesamiento de gráficos
  - ▶ No confundir con *Tarjeta Gráfica*, la placa que aloja a la GPU así como memoria y otra circuitería
  - ▶ Aligera de trabajo a la CPU
    - ★ La CPU se encarga de la gestión de la escena
    - ★ La GPU se encarga de la visualización de la misma



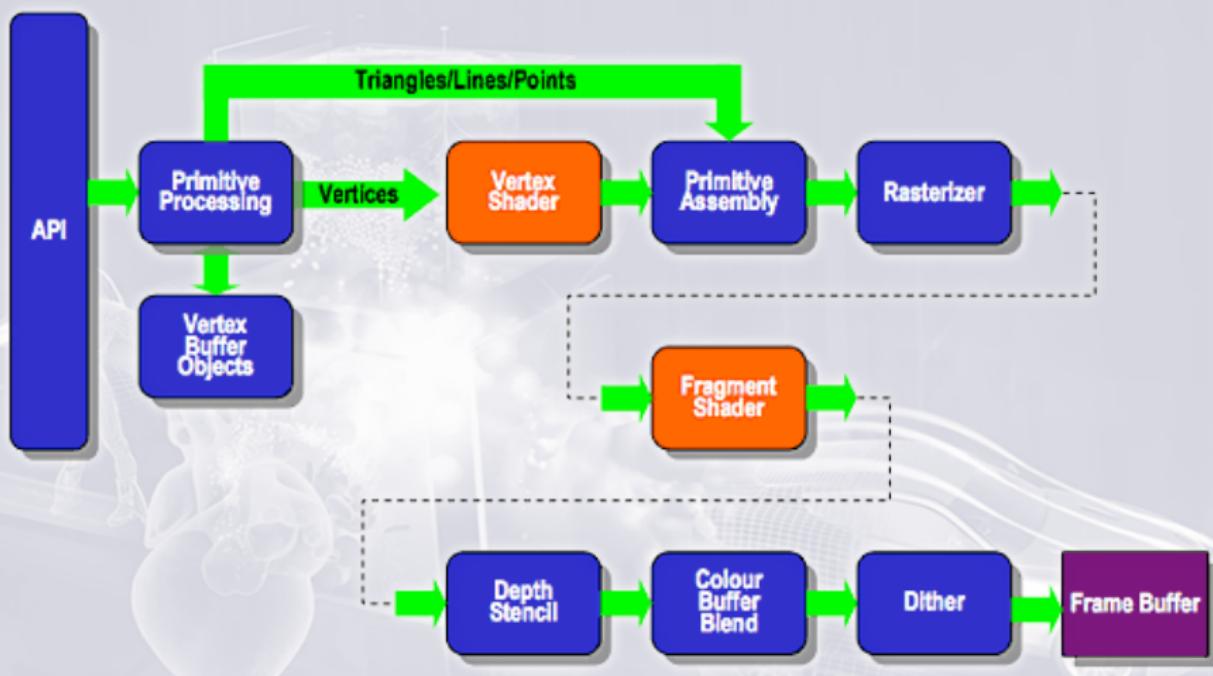
# Optimización del rendering

## Existing Fixed Function Pipeline



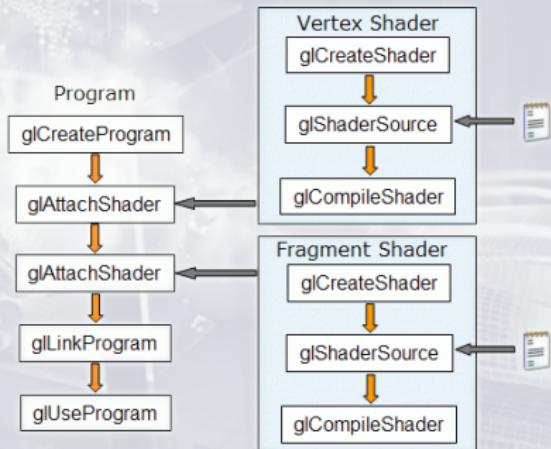
# Optimización del rendering

## ES2.0 Programmable Pipeline



# Estructura de un programa shader

- Los Shaders se escriben en GLSL (OpenGL Shading Language)
  - ▶ Lenguaje inspirado en C
- Modo de uso
  - ▶ El código fuente de cada shader es compilado por OpenGL
  - ▶ Un programa objeto es creado que enlaza todos los shaders

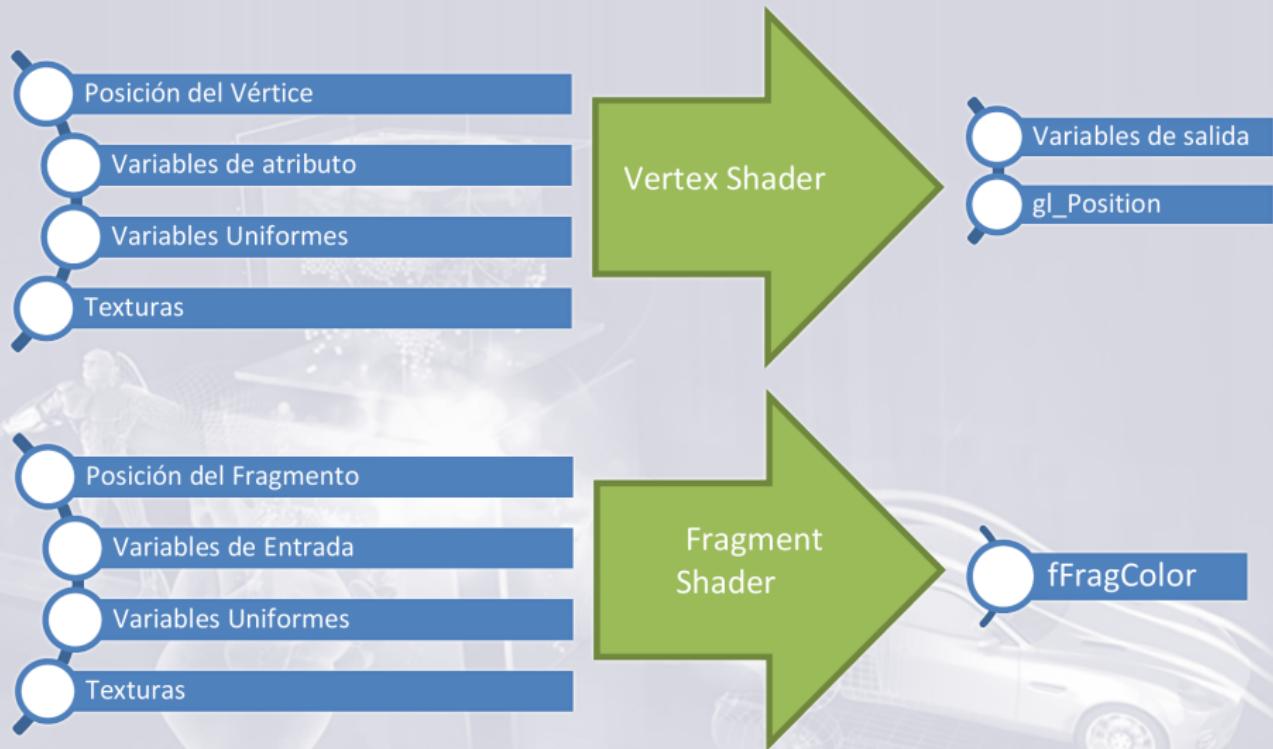


# Vertex y fragment shaders

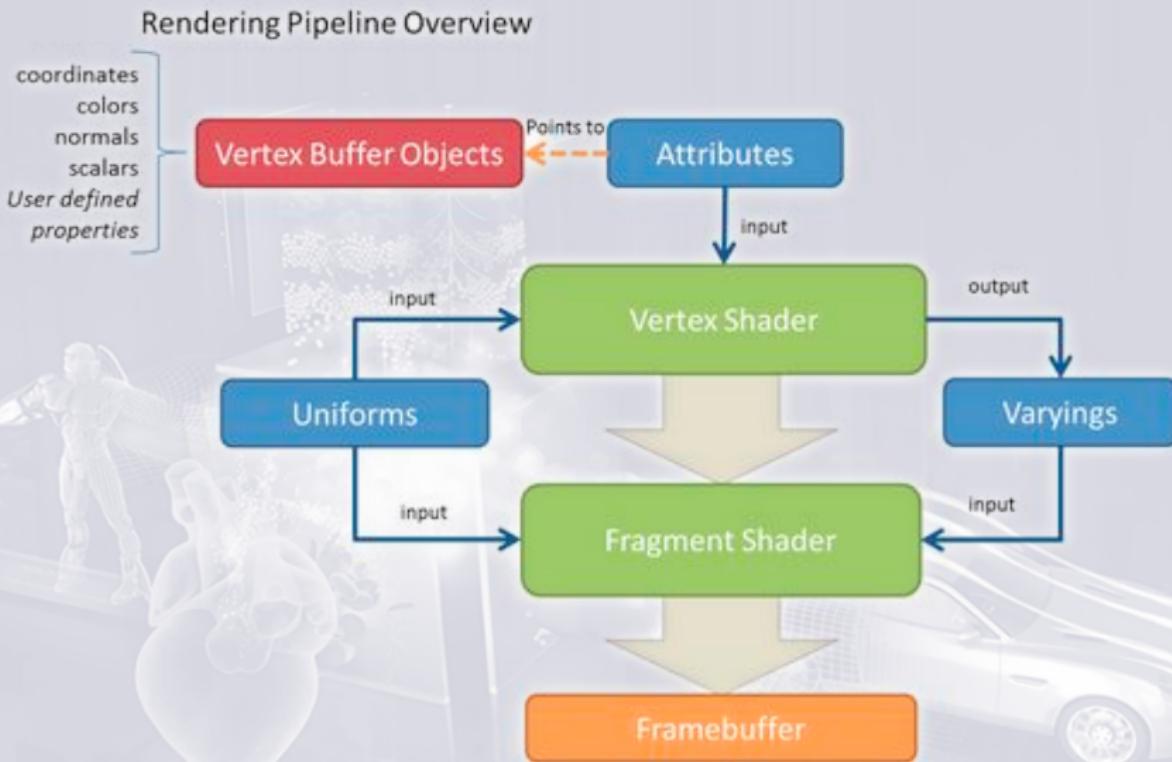
- El vertex shader es llamado para cada vértice
- Realiza cálculos asociados a los vértices
- En etapas posteriores se realiza la rasterización
- El fragment shader es llamado para cada píxel
- Realiza cálculos con el objetivo de obtener el color del píxel



# Flujo de datos en el vertex y fragment shader (1)



# Flujo de datos en el vertex y fragment shader (y 2)

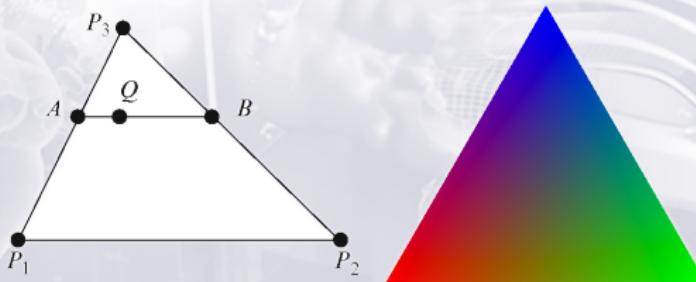


# Vertex shader

- Se ejecuta sobre cada vértice de la escena
- Información de entrada
  - ▶ Vértices: Coordenadas, normales, color, etc.
  - ▶ Iluminación: Vectores a las fuentes, intensidades, etc.
- Operaciones que se pueden programar
  - ▶ Transformaciones de vértices y normales
  - ▶ Cálculo de la iluminación por vértice
  - ▶ Gestión de las coordenadas de textura
- Salidas
  - ▶ El vértice en coordenadas de dispositivo, `gl_Position`, obligatorio
  - ▶ Intensidad lumínica o color en los vértices
  - ▶ Normales, coordenadas de textura, etc.

# Fragment shader

- Se ejecuta sobre cada píxel
- Información de entrada
  - ▶ Información *interpolada*
  - ▶ También se tiene acceso a información constante
- Operaciones que se pueden realizar
  - ▶ Principalmente, calcular el color del píxel
  - ▶ Mediante iluminación, texturas, etc.
- Información de salida: El color del píxel, `gl_FragColor`, obligatorio



# Parámetros de entrada / salida

GLSL

## ● uniform

- ▶ De solo lectura, constantes en todo el cauce de procesamiento de una primitiva
- ▶ Accesibles por ambos shaders
- ▶ Se establecen en la aplicación OpenGL
- ▶ Pueden especificar valores como:
  - ★ Matrices de transformación:  
Modelado, vista, proyección, una composición de varias, etc.
  - ★ Planos de recorte
  - ★ Propiedades del material: componente ambiental, difuso, etc.
  - ★ Propiedades de las luces: color, posición, dirección, etc.
  - ★ Parámetros de efectos: niebla, densidad, etc.

# Parámetros de entrada / salida (y 2)

## ● attribute

- ▶ De solo lectura
- ▶ Accesibles solo por el vertex shader
- ▶ Se establecen en la aplicación OpenGL
- ▶ Especifican información asociada a los vértices
  - ★ Coordenadas
  - ★ Color
  - ★ Normal
  - ★ Coordenadas de textura

## ● varying

- ▶ Variables de paso de información entre shaders
- ▶ De escritura/salida en el vertex y lectura/entrada en el fragment
- ▶ Deben estar en ambos

# Otros tipos de datos en GLSL

- GLSL se parece mucho a C
  - ▶ Se pueden usar tipos como float, int, bool con los operadores habituales
- GLSL incluye nuevos tipos vectoriales y matriciales
  - ▶ Enteros: ivec2, ivec3, ivec4
  - ▶ Reales: vec2, vec3, vec4
  - ▶ Matrices cuadradas reales: mat2, mat3, mat4
  - ▶ Tienen los operadores + y \* sobrecargados
  - ▶ Pueden accederse
    - ★ Con el operador tradicional [ ]
    - ★ Con nombres significativos, por ejemplo, color.r, punto.x
- GLSL incluye funciones como
  - ▶ abs, max, sqrt, pow, log, cos, normalize, dot, cross, etc.

# Variables

- Convenciones para nombrar variables
  - ▶ a\_nombre, para nombrar atributos de vértices
  - ▶ u\_nombre, para nombrar variables uniformes
  - ▶ v\_nombre, para las salidas del vertex shader
  - ▶ f\_nombre, para las salidas del fragment shader
- Nombres usados habitualmente, prácticamente un estándar
  - ▶ a\_vertex, el vértice actual, en coordenadas del modelo
  - ▶ a\_normal, la normal del vértice
  - ▶ a\_color, el color del vértice
- Se disponen en ambos shaders de diversas variables uniformes
  - ▶ u\_ModelViewMatrix, la matriz de modelado y vista
  - ▶ u\_ProjectionMatrix, la matriz de proyección, etc.

# Ejemplos: Gouraud (1)

# GLSL

## Gouraud: Vertex shader: Parámetros de entrada / salida

```
// ----- Constantes -----
// Transformaciones para puntos y normales
uniform mat4 u_mvp_matrix;
uniform mat3 u_normalMatrix;

// Parámetros de un material Lambertiano
uniform vec4 u_ambient;      uniform vec4 u_diffuse;
uniform vec4 u_specular;     uniform float u_shininess;

// Una luz direccional
uniform vec3 u_light_direction;   uniform vec4 u_light_color;

// El vector al observador
uniform vec3 u_observer

// ----- Atributos de los vértices -----
attribute vec4 a_position;
attribute vec3 a_normal;

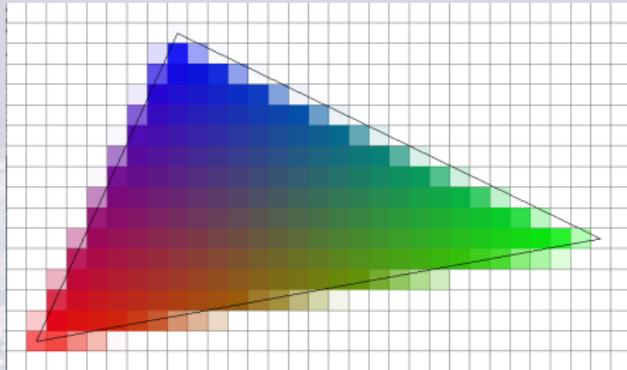
// ----- Salida para el fragment shader -----
varying vec4 v_color;
```

# Gouraud (2)

## Gouraud: Vertex shader: main

```
void main() {  
    // Variables locales para cálculos intermedios  
    float NdotL, NdotHV;  
    vec3 HV, normal;  
  
    // Cálculo del vértice en coordenadas de dispositivo  
    gl_Position = u_mvp_matrix * a_position;  
    // Cálculo de la normal en coordenadas de vista  
    normal = u_normalMatrix * a_normal;  
  
    // Cálculo del color por Lambert, cálculos intermedios  
    NdotL = max (dot (normal, u_light_direction), 0.0);  
    HV = normalize (u_observer + u_light_direction);  
    NdotHV = max (dot (normal, HV), 0.0);  
  
    // Cálculo del color por Lambert  
    v_color = min (u_ambient +  
        NdotL * u_diffuse * u_light_color +  
        pow (NdotHV, u_shininess) * u_specular * u_light_color,  
        vec4(1,1,1,1));  
}
```

# Gouraud (y 3)

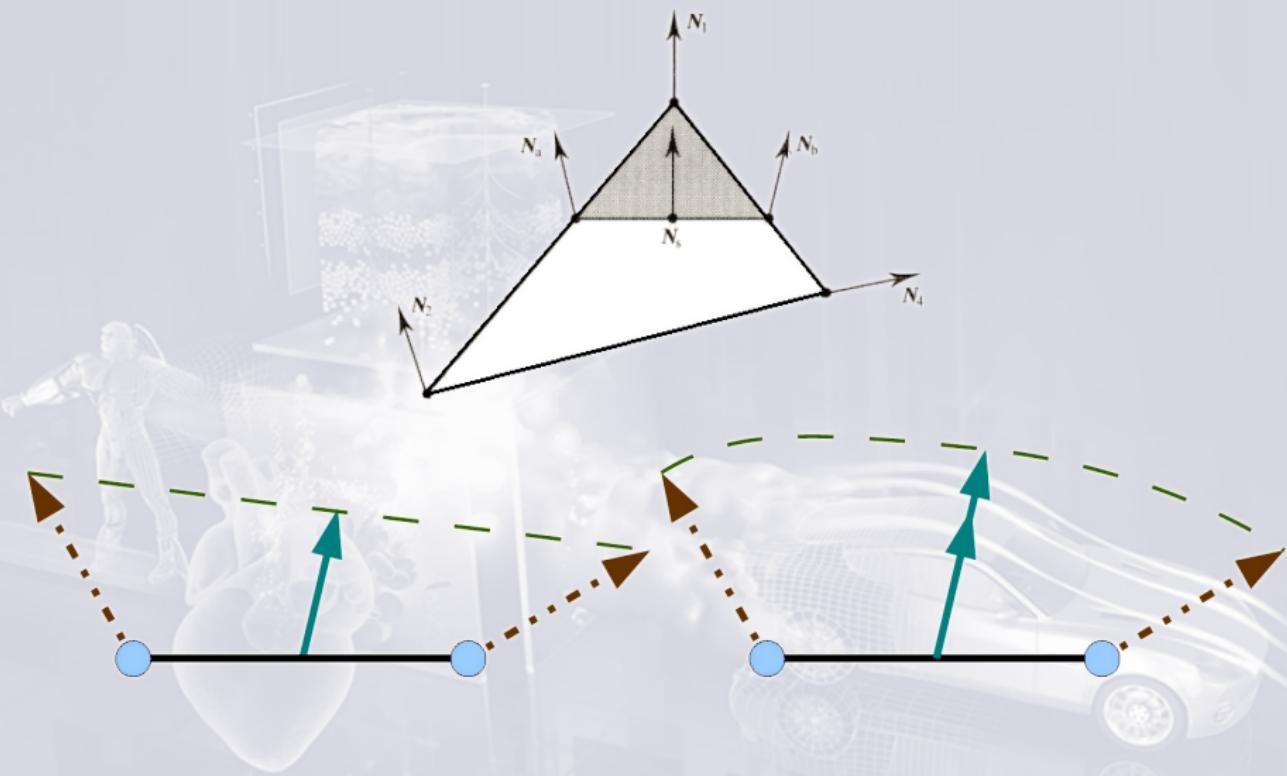


## Fragment shader: Parámetros y main

```
// Entrada: Un color interpolado
varying vec4 v_color;

// En el caso de Gouraud no son necesarios más cálculos
void main()
{
    gl_FragColor = v_color;
}
```

# Ejemplos: Phong (1)



# Phong (2)

## Phong: Vertex shader: Parámetros y main

```
// ----- Constantes -----
// Transformación del modelo al dispositivo
uniform mat4 u_mvp_matrix;
uniform mat3 u_normalMatrix;

// ----- Atributos de los vértices -----
attribute vec4 a_position;
attribute vec3 a_normal;

// ----- Salida para el fragment shader -----
varying vec3 v_normal;

// ----- Programa -----
void main()
{
    // Cálculo del vértice en coordenadas de dispositivo
    gl_Position = u_mvp_matrix * a_position;

    // Las normales son transmitidas para su interpolación
    v_normal = u_normalMatrix * a_normal;
}
```

# Phong (3)

## Phong: Fragment shader: Parámetros

```
// ----- Constantes -----  
  
// Parámetros de un material Lambertiano  
uniform vec4 u_ambient;      uniform vec4 u_diffuse;  
uniform vec4 u_specular;     uniform float u_shininess;  
  
// Una luz direccional  
uniform vec3 u_light_direction;   uniform vec4 u_light_color;  
  
// El vector al observador  
uniform vec3 u_observer  
  
// ----- Entrada desde el vertex shader -----  
varying vec3 v_normal;
```

# Phong (y 4)

## Phong: Fragment shader: main

```
void main()
{
    float NdotL;           // Para cálculos intermedios
    float NdotHV;
    vec3 HV;
    vec3 normal;          // Para normalizar la normal de entrada

    // Cálculo del color por Lambert, cálculos intermedios
    normal = normalize (v_normal);
    NdotL = max (dot (normal, u_light_direction), 0.0);
    HV = normalize (u_observer + u_light_direction);
    NdotHV = max (dot (normal, HV), 0.0);

    // Cálculo del color por Lambert
    gl_FragColor = min (u_ambient +
        NdotL * u_diffuse * u_light_color +
        pow (NdotHV, u_shininess) * u_specular * u_light_color,
        vec4(1,1,1,1));
}
```

# Texturas (1)

## Texturas: Vertex shader: Parámetros y main

```

// _____ Constantes _____
// Transformación del modelo al dispositivo
uniform mat4 u_mvp_matrix;

// _____ Atributos de los vértices _____
attribute vec4 a_position;
attribute vec2 a_texcoord;

// _____ Salida para el fragment shader _____
varying vec2 v_texcoord;

void main()
{
    // Cálculo del vértice en coordenadas del dispositivo
    gl_Position = u_mvp_matrix * a_position;

    // Las coords de textura son transmitidas para su interpolación
    v_texcoord = a_texcoord;
}

```



# Texturas (y 2)

## Texturas: Fragment shader: Parámetros y main

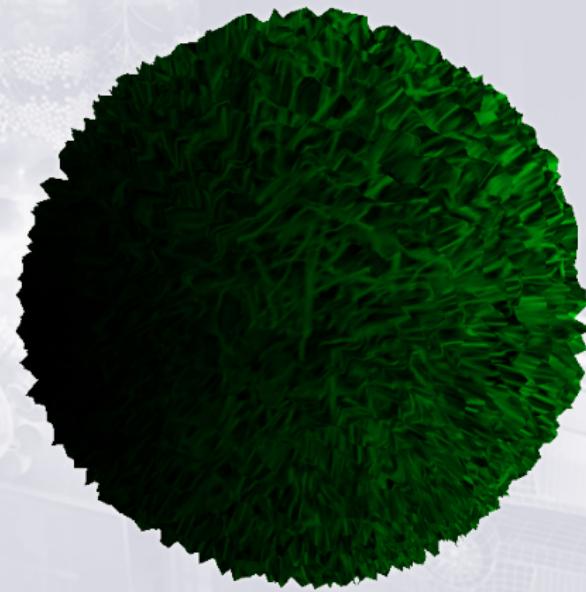
```
// ----- Constantes -----
// La textura
uniform sampler2D u_texture;

// ----- Entrada desde el vertex shader -----
// Las coordenadas de textura concretas de este píxel
varying vec2 v_texcoord;

void main()
{
    // Cálculo del color a partir de la textura
    gl_FragColor = texture2D (u_texture, v_texcoord);
}
```

# Uso de Shaders en Three.js

- Cada material predefinido lleva asociado su shader
- Para usar shaders personalizados debemos usar la clase `ShaderMaterial`



# Clase ShaderMaterial

## Código fuente de los shaders

### Shader: Código fuente del vertex shader

```
<script type="x-shader/x-vertex" id="vertexS">
    uniform float amplitude;
    attribute float displacement;
    varying vec3 v_normal;
    varying vec4 v_position;

    void main() {
        vec3 newPosition = position + normal * displacement * amplitude;
        v_position = modelViewMatrix * vec4 (newPosition, 1.0);
        gl_Position = projectionMatrix * v_position;

        // Cálculo de la normal en coordenadas de vista
        v_normal = normalize (vec3 (normalMatrix * normal));
    }
</script>
```

# Clase ShaderMaterial

uniform y attribute disponibles

## ShaderMaterial: uniform y attribute disponibles

```
// Datos globales
uniform mat4 modelMatrix;
uniform mat4 viewMatrix;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
uniform mat3 normalMatrix;
uniform vec3 cameraPosition;

// Atributos para cada vértice
attribute vec3 position;
attribute vec3 normal;
attribute vec2 uv;
attribute vec3 color;
```

# Clase ShaderMaterial

Añadido de más uniform y attribute y define

## ShaderMaterial: Añadido de más uniform y attribute

```
uniforms = {  
    amplitude: { type: "f", value: 1.0 }  
};  
// Se puede modificar cuando se desee  
uniforms.amplitude.value = 2.5;  
  
// Los atributos se asignan a la geometría, debe ser BufferGeometry  
var nVertices = model.geometry.attributes.position.count;  
var displacement = new Float32Array (nVertices);  
for (var v = 0; v < nVertices; v++) {  
    displacement[v] = Math.random();  
}  
model.geometry.addAttribute( 'displacement' ,  
    new THREE.BufferAttribute (displacement,1));  
  
// Cuando se cambian atributos hay que solicitar una actualización  
model.geometry.attributes.displacement.needsUpdate = true;
```

# Clase ShaderMaterial

## Tipos de datos

- Correspondencia entre tipos de datos de GLSL y Three.js

GLSL	type:	value
int	“i”	un entero
float	“f”	un real
vec2	“v2”	THREE.Vector2
vec3	“v3”	THREE.Vector3
vec3	“c”	THREE.Color
vec4	“v4”	THREE.Vector4
mat3	“m3”	THREE.Matrix3
mat4	“m4”	THREE.Matrix4
sampler2D	“t”	THREE.Texture

# Clase ShaderMaterial

## Creación del material

# Three.js

## ShaderMaterial: Creación del material

```
var shaderMat = new THREE.ShaderMaterial ( {  
    uniforms: uniforms,  
    vertexShader: document.getElementById ('vertexS').textContent,  
    fragmentShader: document.getElementById ('fragmentS').textContent,  
  
    // Se le pueden poner otros campos opcionales  
    wireframe: true, // y se mostraría en modo alambre  
    transparent: true, // obligatorio si se manejan transparencias  
    lights: true // si se van a usar luces definidas en THREE  
});
```

# Uso de luces en un shader personalizado

- Al definir el `ShaderMaterial` se debe
  - ▶ Poner el atributo `lights` a `true`
  - ▶ Añadir a nuestros `uniforms` los `uniforms` de las luces
- En el shader se debe declarar:
  - ▶ Una estructura con los campos adecuados según el tipo de luz que se desea usar
    - ★ Esa información se obtiene de los fuentes de la biblioteca
    - ★ En concreto de  
`src/renderer/shaders/ShaderChunk/lights_pars.glsl`
  - ▶ Una variable `uniform` que sea un array de elementos de dicha estructura
  - Se pueden copiar y pegar esas declaraciones desde ese archivo en nuestro Shader

# Uso de luces en un shader personalizado

Ejemplo: Definición del `ShaderMaterial`

**Luces en un shader personalizado:** En la aplicación js

```
var shaderMat = new THREE.ShaderMaterial ({  
    // se mezclan los uniforms de las luces con otros que hayamos  
    // podido definir nosotros, en el ejemplo, amplitud  
    uniforms : THREE.UniformsUtils.merge ([  
        THREE.UniformsLib['lights'],  
        {  
            amplitude : {type : 'f', value : 5.0 }  
        }  
    ]),  
    vertexShader: document.getElementById ('vertexS').textContent,  
    fragmentShader: document.getElementById ('fragmentS').textContent,  
    // se activa el atributo lights  
    lights : true  
});
```

# Uso de luces en un shader personalizado

Ejemplo: Definición y uso en el Shader

Archivo **lights\_pars.glsl**: Fragmento relativo a las luces direccionales

```
struct DirectionalLight {  
    vec3 direction;  
    vec3 color;  
    int shadow;  
    float shadowBias;  
    float shadowRadius;  
    vec2 shadowMapSize;  
};  
uniform DirectionalLight directionalLights[ NUM_DIR_LIGHTS ];
```

- Se copian las declaraciones en nuestro Shader y se usa el array de luces de la manera habitual

Nuestro shader: Uso de las luces

```
for ( int i = 0; i < NUM_DIR_LIGHTS; i++ ) {  
    esteColor = directionalLights[i].color;  
    // se hacen los cálculos necesarios  
}
```

# Visualización

Francisco Velasco Anguita

Dpto. Lenguajes y Sistemas Informáticos  
Universidad de Granada

Sistemas Gráficos

Grado en Ingeniería Informática  
Curso 2017-2018