

# Informática Gráfica.

## Sesión 8: Texturas, sombreado y materiales.

Carlos Ureña, Sept 2025.  
Dept. Lenguajes y Sistemas Informáticos.  
Universidad de Granada.

---

### Sección 1.

#### Texturas

1. Coordenadas de textura.
2. Asignación explícita de coords. de textura.
3. Asignación de coordenadas y consulta de texels.
4. Consulta de texels en texturas de imagen.
5. Problemas sobre texturas.

### Índice

---

Texturas .....	3
Sombreado ( <i>Shading</i> ) .....	37
Luces, materiales y texturas en Godot. ....	55

#### 1. Texturas.

### Detalles a pequeña escala

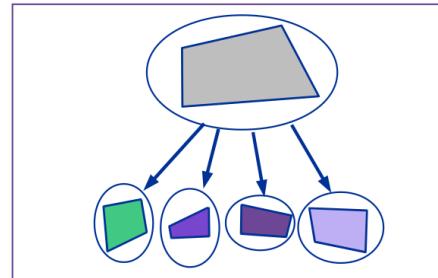
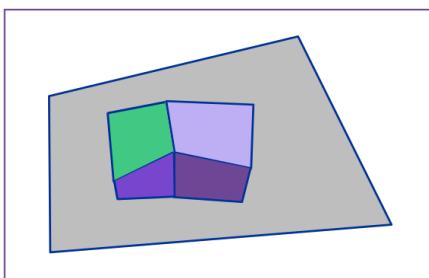
Los objetos reales presentan a veces detalles a pequeña escala, como son manchas, defectos, motivos ornamentales, rugosidades, o, en general, cambios en el espacio de los atributos que determinan su apariencia:



- Estas variaciones se pueden modelar como funciones que asignan a cada punto de la superficie de un objeto un valor diferente para algunos parámetros del MIL.
- Lo más usual es variar las reflectividades difusa y ambiente, pero se hace también con la normal (rugosidades), o a veces otros parámetros.

## Implementación de detalles: polígonos de detalle

Para reproducir detalles a pequeña escala se pueden usar **polígonos de detalle**, son polígonos pequeños adicionales a los que definen la geometría de la escena, pero con materiales y/o orientación distintos entre ellos:



La desventaja de esta opción es su enorme complejidad en espacio (necesario para almacenar muchos polígonos pequeños) y tiempo (empleado en su visualización).

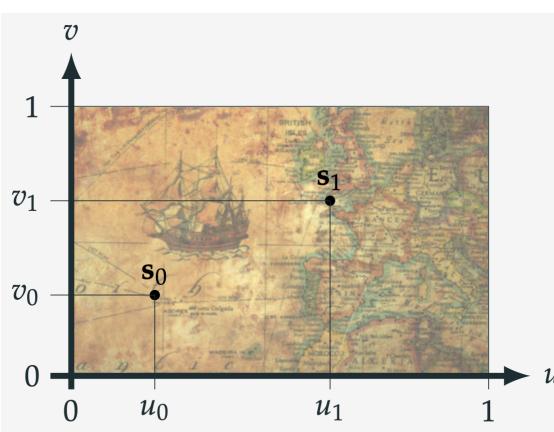
## Texturas

Una opción mejor (mucho más eficiente) es usar imágenes para representar las funciones antes citadas. A estas imágenes se les llama (en el contexto de la Informática Gráfica) **texturas**:

- Una textura se puede interpretar como una función  $T$  que asocia a cada punto  $s$  de un dominio  $D$  (usualmente  $[0, 1] \times [0, 1]$ ) un valor para un parámetro del MIL (típicamente el color  $C(p)$ ). La función  $T$  determina como varía el parámetro en el espacio.
- La función  $T$  puede estar representada en memoria como una matriz de pixels RGB (una imagen discretizada), a cuyos pixels se les llama **texels** (*texture elements*). A esta imagen se le llama **imagen de textura**.
- La función  $T$  puede también representarse como un subprograma que calcula los valores a partir de  $s$  (que se le pasa como parámetro). A este tipo de texturas le llamamos **texturas procedurales**.

## La textura como una función

En este ejemplo vemos una imagen de textura (bidimensional). El dominio  $D$  es  $[0, 1]^2$ . Cada punto del dominio es una par  $s = (u, v)$ . Los valores  $T(s) = T(u, v)$  son ternas RGB.



## Aplicación de texturas

Vemos varias formas de asignar colores a puntos del objeto:

- Evaluación del MIL con reflectividades blancas (izq. abajo)
- Eso directo de colores de la textura (izq. arriba)
- Evaluación del MIL con reflectividades obtenidas de la textura (der.)



## Coordenadas de textura

Para poder aplicar una textura a la superficie de un objeto, es necesario hacer corresponder cada punto  $\mathbf{p} = (x, y, z)$  de su superficie con un punto  $(u, v)$  del dominio de la textura:

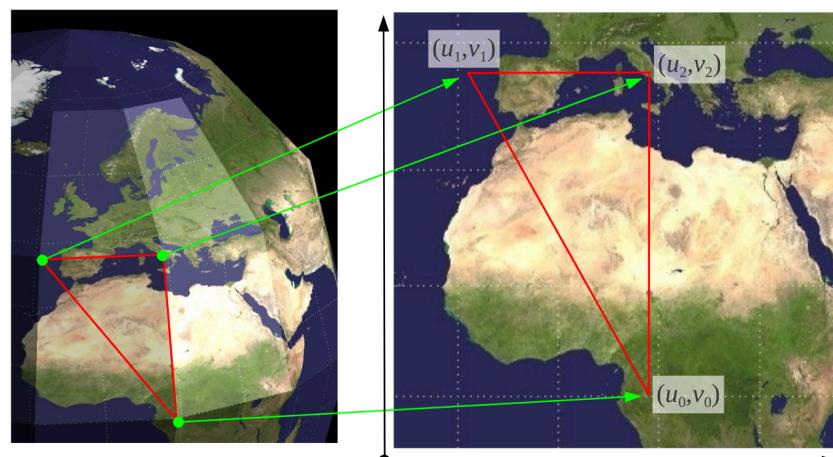
- Debe existir una función  $f$  tal que  $(u, v) = f(x, y, z)$
- Si  $(u, v) = f(x, y, z)$  entonces decimos que  $(u, v)$  son las **coordenadas de textura** del punto  $\mathbf{p} = (x, y, z)$ .
- Normalmente  $f$  se descompone en dos componentes  $f_u, f_v$ , de forma que  $u = f_{u(x,y,z)}$  y  $v = f_{v(x,y,z)}$
- La función  $f$  puede implementarse usando una tabla de coordenadas de textura de los vértices, o bien calcularse proceduralmente con un subprograma.

### Subsección 1.1.

#### Coordenadas de textura.

### Ejemplo de coordenadas de textura.

Vemos como a cada vértice de un triángulo del modelo se le asignan sus coordenadas de textura  $(u_i, v_i)$  (donde  $i$  es el índice del vértice en la tabla de vértices).



### Asignación explícita o procedural

La asignación de coord. de text. se puede hacer usando:

- **Asignación explícita a vértices:** las coordenadas forman parte de la definición del modelo de escena, y son un dato de entrada al cauce gráfico, en forma de un vector o tabla de coordenadas de textura de vértices  $(v_0, u_0), (v_1, u_1), \dots, (u_{n-1}, v_{n-1})$ .
  - ▶ se puede hacer manualmente en objetos sencillos, o bien
  - ▶ de forma asistida usando software para CAD.

Esto hace necesario realizar una interpolación de coords. de text. en el interior de los polígonos.

- **Asignación procedural:**  $f$  se implementa con un subprograma que se invoca como `CoordText(p)` y que calcula las coordenadas de textura (para un punto  $p$  devuelve el par  $(u, v) = f(p)$  con las coords. de textura de  $p$ ).

## Ejemplo de asignación explícita.

Esto es posible en objetos sencillos como este cubo construido con triángulos. En este ejemplo se busca una asignación de coordenadas de textura que sea continua en las aristas:

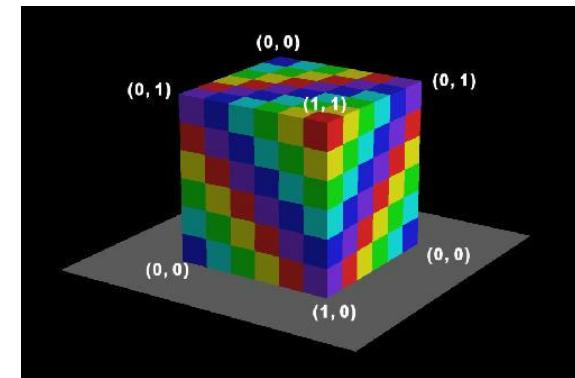


Imagen de Rosalee Wolfe, en [Teaching Texture Mapping Visually](#) (SIGGRAPH 97 Education Slide Set).

---

### Subsección 1.2.

#### Asignación explícita de coords. de textura.

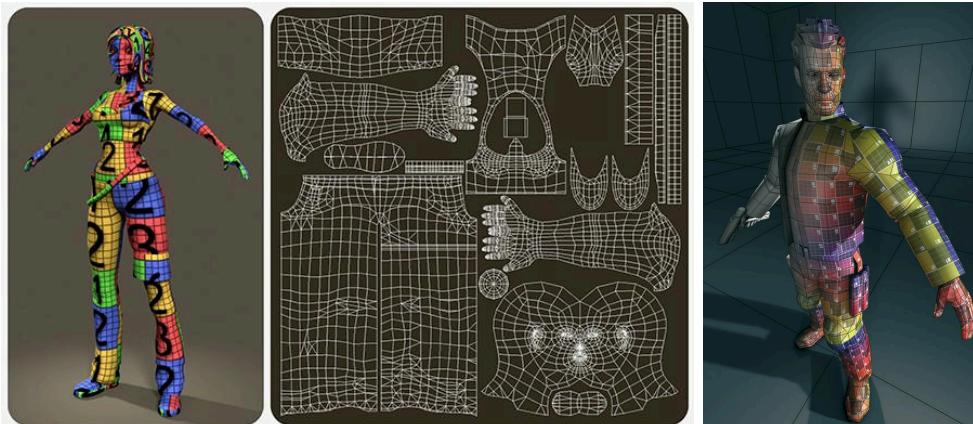
---

1. Texturas.

1.2. Asignación explícita de coords. de textura..

## Ejemplo de uso de herramientas CAD.

En objetos complejos es necesario el uso de herramientas CAD:



Imágenes de [Sean Dixon](#) (izquierda, centro) y [Mayan Escalante](#) (derecha).

---

### Subsección 1.3.

#### Asignación de coordenadas y consulta de texels.

---

## Tipos de asignación procedural

Hay dos opciones:

- **Asignación procedural a vértices:** se invoca `CoordText( $v_i$ )` para calcular las coordenadas de textura en cada vértice  $v_i$ , y las coordenadas obtenidas se almacenan y después se interpolan linealmente en el interior de los polígonos de la malla.
  - ▶ Funciona de forma totalmente correcta (exacta) solo cuando  $f$  es lineal, en otro caso es una aproximación lineal a trozos.
- **Asignación procedural a puntos:** se invoca `CoordText(p)` cada vez que sea necesario evaluar el MIL en un punto de la superficie  $p$ .
  - ▶ Permite exactitud incluso aunque  $f$  sea no lineal.
  - ▶ Esto requiere programación del cauce gráfico, invocando a `CoordText` en cada pixel desde el *fragment shader*.

## Otras opciones para asignación procedural

Otras opciones (no lineales) son estas dos:

- **Coordenadas polares** (proyección en una esfera): el punto  $p$  se expresa en coordenadas polares como una terna  $(\alpha, \beta, r)$ , los valores  $u$  y  $v$  se obtienen de  $\alpha$  y  $\beta$ .
- **Coordenadas cilíndricas** (proyección en un cilindro): el punto  $p$  se expresa en coordenadas cilíndricas como una terna  $(\alpha, y, r)$ , los valores  $u$  y  $v$  se obtienen de  $\alpha$  e  $y$ .

Es muy complicado usarlas con asignación a vértices ( $\alpha$  puede pasar de 360 a 0 en un triángulo, la textura se vería mal), y por tanto requieren usar asignación procedural a puntos (invocar `CoordText` desde los *fragment shaders*).

## Funciones para asignación procedural:

Los tipos de funciones  $f$  más frecuentes son:

- **Funciones lineales** de la posición (proyección en un plano): el punto  $p = (x, y, z)$  se proyecta sobre un plano y se expresa como un par  $(x', y')$  de coordenadas en dicho plano, que se interpretan como coordenadas de textura.
- **Coordenadas paramétricas:** se pueden usar si la malla aproxima una superficie paramétrica (p.ej. la tetera, hecha de superficies paramétricas tipo B-spline). Se usa asignación procedural a vértices. Se trata de funciones no lineales de la posición.

## Funciones lineales (proyección).

En este caso el punto  $p = (x, y, z)$  se proyecta en un plano, y se usan las coordenadas del punto proyectado (en el sistema de referencia del plano), como coordenadas de textura.

El plano estará definido por un punto por el que pasa ( $q$ ) y por dos vectores libres ( $e_u$  y  $e_v$ , de longitud unidad y perpendiculares entre sí). En estas condiciones:

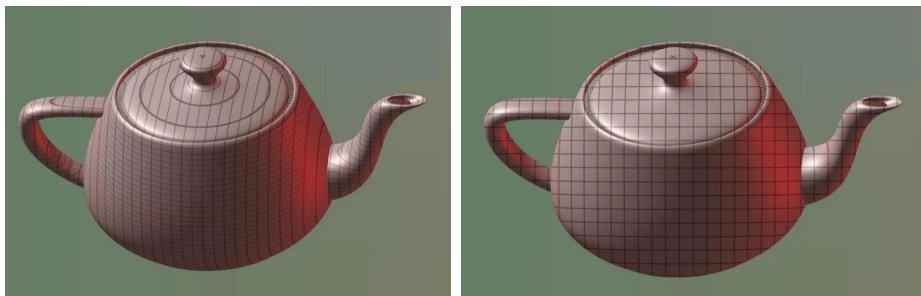
$$u = f_u(p) = (p - q) \cdot e_u \quad v = f_v(p) = (p - q) \cdot e_v$$

como casos particulares, y a modo de ejemplo, podemos hacer  $q$  igual al origen  $(0, 0, 0)$ ,  $e_u = x = (1, 0, 0)$  y  $e_v = y = (0, 1, 0)$ , y en este caso es una proyección paralela el eje Z, sobre el plano XY (descarta la Z)

$$\begin{aligned} u &= x = p \cdot x = (x, y, z) \cdot (1, 0, 0) \\ v &= y = p \cdot y = (x, y, z) \cdot (0, 1, 0) \end{aligned}$$

## Ejemplo de proyección paralela a Z.

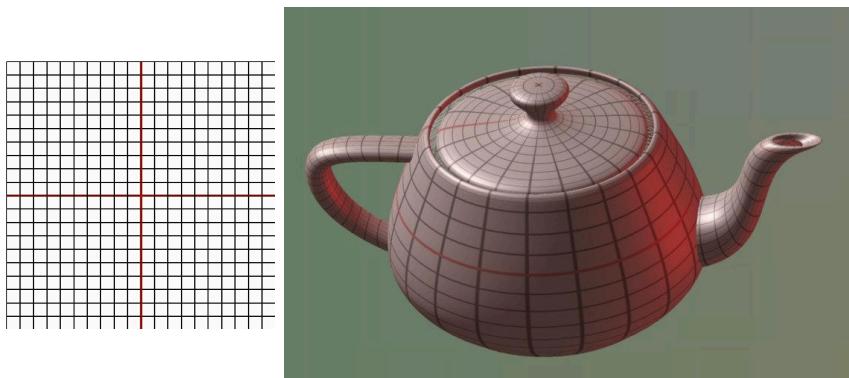
Las coordenadas de  $p$  que se usan en las funciones lineales pueden ser las coordenadas de objeto (izquierda) o bien o las coordenadas de mundo (derecha). Aquí vemos un ejemplo de una proyección paralela al eje Z:



Este método funciona mejor (menor deformación) cuando la normal es aproximadamente paralela a la dirección de proyección (parte frontal en el ejemplo de la izquierda).

## Ejemplo de coordenadas paramétricas

Vemos un ejemplo de textura (izq.) y su aplicación a la tetera (der.). Este tipo de superficies se denominan superficies paramétricas de Bézier o B-spline, y se usan mucho en aplicaciones de diseño (CAD).



Esta imagen se ha generado asignando explícitamente en el programa a cada vértice sus coordenadas de textura, usando para ello sus coordenadas paramétricas.

## Coordenadas paramétricas.

Una **superficie paramétrica** es una variedad plana de dos dimensiones (que puede ser abierta o cerrada), para la cual existe una función  $g$  (con dominio en  $[0, 1] \times [0, 1]$ ) tal que, si  $p$  es un punto de la superficie, entonces existe un tupla de reales  $(s, t)$  tales que  $p = g(s, t)$ :

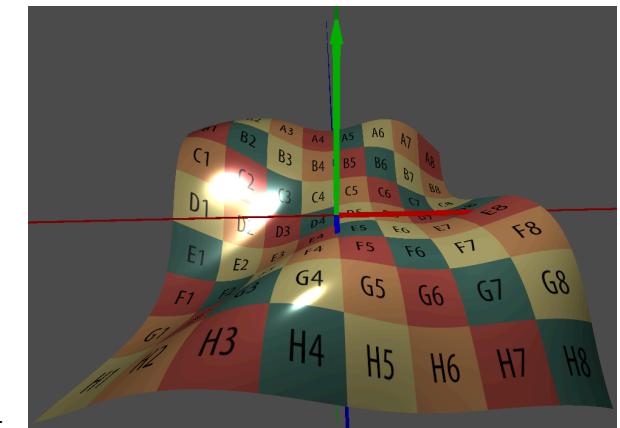
- En este caso, al par  $(s, t)$  se le llaman **coordenadas paramétricas** del punto  $p$ , y a la función  $g$  se le llama **función de parametrización** de la superficie.
- Usando la capacidad de evaluar  $g$ , podemos construir una malla que aproxima cualquier superficie paramétrica. La posición  $p_i$  del  $i$ -ésimo vértice se obtiene como  $g(s_i, t_i)$ , donde los  $(s_i, t_i)$  forman una rejilla en  $[0, 1] \times [0, 1]$ .
- En estas condiciones, podemos hacer  $(u, v) = f(p) = (s, t)$ , es decir, podemos usar las coordenadas paramétricas como coordenadas de textura.

## Coordenadas paramétricas: campos de alturas

Una opción bastante simple es usar un **campo de alturas (height field)**, en el cual el punto  $p = (x, y, z)$  se escribe en función de  $(u, v)$  así:

$$\begin{aligned} x &= u \\ y &= h(u, v) \\ z &= v \end{aligned}$$

donde  $h$  es una función que expresa la altura de cada vértice en el eje Y (podrían ser los otros).



## Coordenadas esféricas

Se basa en usar las coordenadas polares (longitud, latitud y radio) del punto p:

- Equivale a una proyección radial en una esfera.
- Las coordenadas  $(\alpha, \beta, r)$  se obtienen a partir de las coordenadas cartesianas  $(x, y, z)$  (normalmente coordenadas de objeto, con el origen en un punto central de dicho objeto). Hacemos:

$$\alpha = \text{atan2}(z, x) \quad \beta = \text{atan2}\left(y, \sqrt{x^2 + z^2}\right)$$

- Se obtiene  $\alpha$  en el rango  $[-\pi, \pi]$  y  $\beta$  en el rango  $[-\frac{\pi}{2}, \frac{\pi}{2}]$ . Por tanto, podemos calcular  $u$  y  $v$  como sigue:

$$u = \frac{1}{2} + \frac{\alpha}{2\pi} \quad v = \frac{1}{2} + \frac{\beta}{\pi}$$

el valor de  $r$  no se usa y por tanto no es necesario calcularlo.

## Coordenadas cilíndricas

Se usan las coordenadas polares (ángulo y altura) del punto p:

- Equivale a una proyección radial en un cilindro (cuyo eje es usualmente un eje vertical central al objeto).
- Las coordenadas  $(\alpha, h, r)$  se obtienen a partir de las coordenadas cartesianas  $(x, y, z)$  (también con origen en el centro del objeto). Hacemos:

$$\alpha = \text{atan2}(z, x) \quad h = y$$

- El valor de  $\alpha$  está en el rango  $[-\pi, \pi]$  y  $h$  en el rango  $[y_{\min}, y_{\max}]$  (el rango en Y del objeto). Por tanto, podemos calcular  $u$  y  $v$  como:

$$u = \frac{1}{2} + \frac{\alpha}{2\pi} \quad v = \frac{y - y_{\min}}{y_{\max} - y_{\min}}$$

tampoco el valor de  $r$  se usa ahora y por tanto no es necesario calcularlo.

## Ejemplo de coordenadas esféricas

Vemos un esquema de la proyección (izq.) y el resultado en varios objetos (der.)

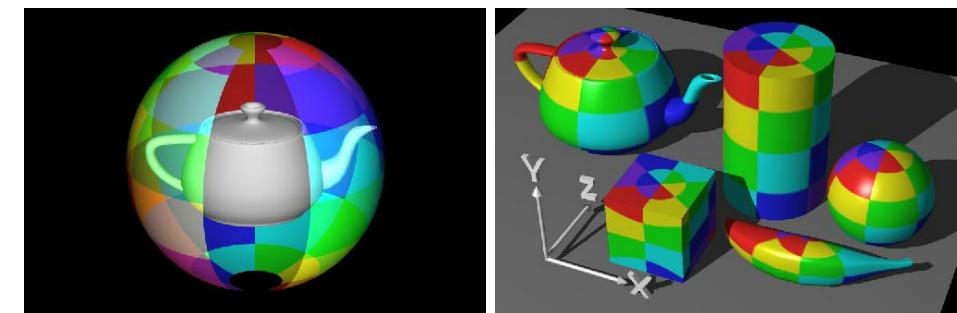


Imagen de Rosalee Wolfe, en [Teaching Texture Mapping Visually](#) (SIGGRAPH 97 Education Slide Set).

## Ejemplo de coordenadas cilíndricas

Vemos un esquema de la proyección (izq.) y el resultado en varios objetos (der.)

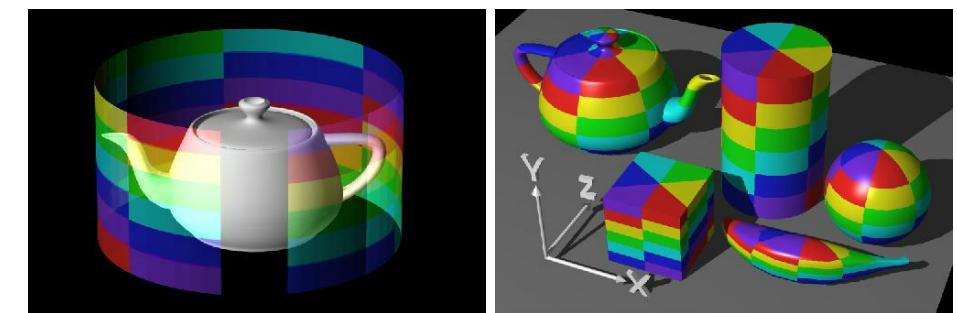


Imagen de Rosalee Wolfe, en [Teaching Texture Mapping Visually](#) (SIGGRAPH 97 Education Slide Set).

## Consulta de texels en texturas de imagen.

En una textura de imagen con  $n_x$  columnas de texels y  $n_y$  filas, podemos interpretar que cada texel tiene asociada un pequeño rectángulo contenido en  $[0, 1]^2$ . El texel en la columna  $i$ , fila  $j$  tendrá un área con centro en el punto  $(c_i, d_j)$

---

### Subsección 1.4.

#### Consulta de texels en texturas de imagen.

---

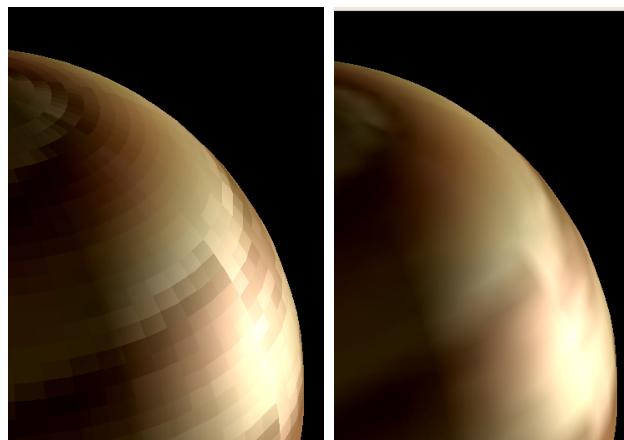
La consulta del color de la textura en un punto  $(u, v)$  puede hacerse de dos formas:

- **más cercano:** usar el color del texel cuyo centro sea más cercano a la posición  $(u, v)$ , es equivalente a seleccionar el texel cuya área contiene a  $(u, v)$ .
- **interpolación:** realizar un interpolación (bilineal) entre los colores de los cuatro texels con centros más cercanos al punto  $(u, v)$ .

Las diferencias entre ambos métodos son visibles cuando la proyección en la ventana de un texel ocupa muchos pixels.

## Interpolación bilineal

Aquí vemos una textura de baja resolución, vista de cerca, que se visualiza usando los dos métodos:



---

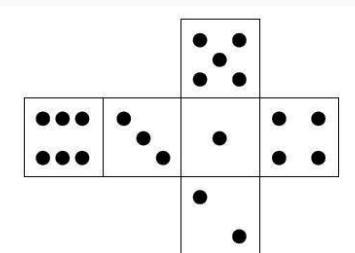
Subsección 1.5.  
Problemas sobre texturas.

---

## Problemas: coordenadas de textura (1/3)

### Problema 8.1:

Supongamos que se desea crear una malla indexada para un cubo, de forma que deseamos aplicar un textura que incluya las caras de un dado. Para ello disponemos de una imagen de textura que tiene una relación de aspecto 4:3. La imagen aparece aquí:



(continua en la siguiente transparencia)

## Problemas: coordenadas de textura (2/3)

### Problema 8.2:

Considera de nuevo el cubo y la textura del problema anterior. Ahora supón que queremos que el cubo se visualize iluminado, para lo cual debemos de asignar normales a los vértices. Responde a estas cuestiones

- Describe razonadamente si sería posible usar la misma tabla de vértices y la misma tabla de coordenadas de textura que has escrito en el problema anterior, o es necesario usar otra tabla distinta.
- Si has respondido que no es posible usar las mismas tablas, escribe la nueva tabla de vértices, la nueva tabla de coordenadas de textura. Asimismo, escribe como sería la tabla de normales.

## Problemas: coordenadas de textura (1/3, cont.)

### Problema 8.1 (continuación):

Responde a estas cuestiones:

1. Describe razonadamente cuantos vértices (como mínimo) tendrá el modelo.
2. Escribe la tabla de coordenadas de vértices, la tabla de coordenadas de textura, y la tabla de triángulos. Ten en cuenta que el cubo tiene lado unidad y su centro está en  $(\frac{1}{2}, \frac{1}{2}, \frac{1}{2})$ . Dibuja un esquema de la textura en la cual cada vértice del modelo aparezca etiquetado con su número de vértice más sus coordenadas de textura.

## Problemas: coordenadas de textura (3/3)

### Problema 8.3:

Considera un cubo (de nuevo de lado unidad, y con centro en  $(\frac{1}{2}, \frac{1}{2}, \frac{1}{2})$ ) que se quiere visualizar con una textura a partir de una única imagen (cuadrada) que se replicará en las 6 caras de dicho cubo.

Asume que no se va a usar iluminación (no es necesario calcular la tabla de normales). Escribe ahora la tabla de coordenadas de vértices y la tabla de coordenadas de textura.

---

## Sección 2.

### Sombreado (*Shading*)

---

1. Evaluación del MIL en rasterización.
2. Sombreado plano.
3. Sombreado en los vértices.
4. Sombreado en los píxeles.

2. Sombreado (*Shading*).

2.1. Evaluación del MIL en rasterización..

## Alternativas

En el algoritmo de Z-buffer, la evaluación del MIL puede hacerse en tres puntos distintos del cauce gráfico:

- **Sombreado plano:** (*flat shading*) una vez por cada polígono que forma el modelo, asignando el resultado (una terna RGB única) a todos los pixels donde se proyecta el polígono.
- **Sombreado de vértices:** (*smooth shading* o *Gouraud shading*) una vez por vértice, cada color RGB obtenido se usa para interpolar los colores de los pixels en cada polígono.
- **Sombreado de pixel:** (*pixel shading* o *Phong shading*) una vez por cada pixel donde se proyecta el polígono

---

#### Subsección 2.1.

##### Evaluación del MIL en rasterización.

---

---

#### Subsección 2.2.

##### Sombreado plano.

---

## Sombreado plano

Este método de sombreado es muy eficiente en tiempo si el modelo es sencillo (es decir, el número de polígonos es pequeño en comparación con el de pixels).

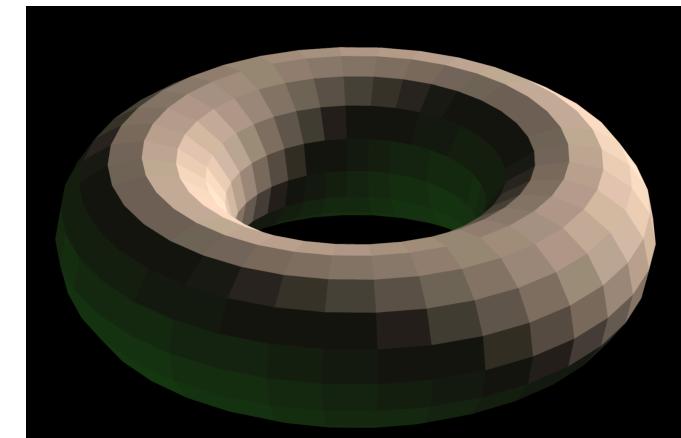
- Se debe seleccionar un punto cualquiera  $p$  de cada polígono, típicamente se usa un vértice, pero podría ser cualquier otro.
- Se usa la normal al polígono  $n_p$ .
- Se calcula el vector al observador  $v$  en  $p$ .

Las desventajas son:

- Puede no ser deseable que se aprecien los polígonos del modelo.
- Produce discontinuidades en el brillo de los pixels en las aristas.
- No es realista si el tamaño del polígono es grande en comparación con la distancia que lo separa al observador, en proyección perspectiva y/o brillos pseudo-especulares.

## Resultados del sombreado plano.

Aquí vemos un objeto curvo aproximado con caras planas y visualizado con sombreado plano (MIL difuso).



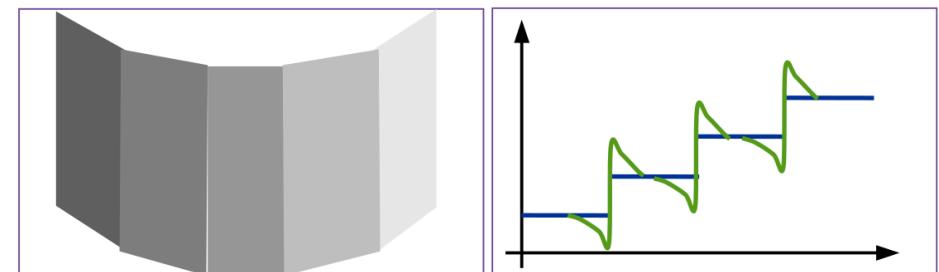
## Resultados del sombreado plano

Aquí se observa la tetera, con sombreado plano, a distintas resoluciones. En este caso el MIL tiene una componente pseudo-especular no nula.



## Bandas Mach

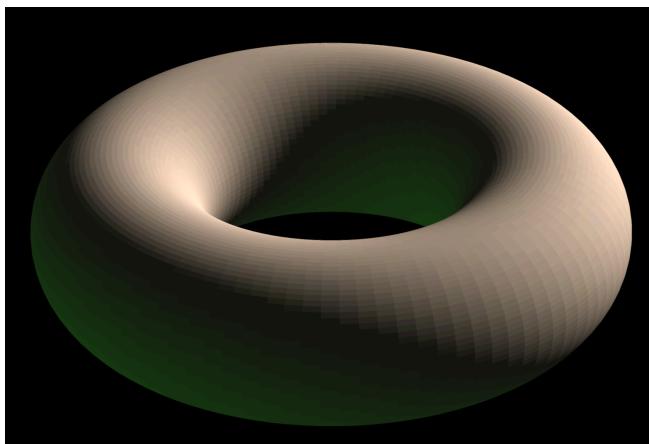
La **Bandas Mach** son una ilusión visual producida por la *inhibición lateral de las neuronas de la retina*, que es un mecanismo desarrollado evolutivamente para resaltar el contraste en aristas entre colores planos:



si no se quiere modelar un objeto formado realmente por caras planas, esta forma de visualizar produce resultados pobres. En algunos casos (objetos hechos de caras planas, iluminación puramente difusa) puede ser muy eficiente y realista.

## Ejemplo de bandas Mach

En este objeto las bandas Mach son fácilmente apreciables:



---

Subsección 2.3.  
Sombreado en los vértices.

---

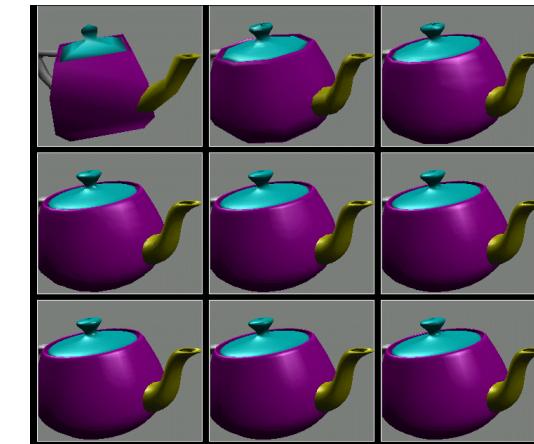
## Sombreado en los vértices

En esta modalidad (*vertex shading*), el MIL se evalúa una vez en cada vértice del modelo.

- La posición  $p$  coincide con la posición del vértice.
- Si la malla de polígonos aproxima un objeto curvo, la normal  $n_p$  puede calcularse como el promedio de las normales de los polígonos adyacentes al vértice.
- La evaluación del MIL produce un color único para cada vértice
- Los valores en los vértices se usan como valores extremos para interpolar los colores de los pixels donde se proyecta el polígono
- La eficiencia en tiempo es parecida al sombreado plano.
- Los resultados son muchas veces más realistas que con sombreado plano.
- Pueden persistir problemas de bandas Mach y poco realismo.

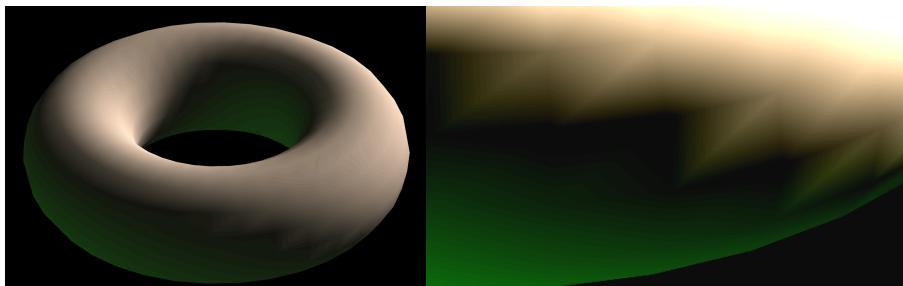
## Pérdida de zonas brillantes

Los resultados mejoran, pero puede haber problemas de pérdida de zonas brillantes (componente pseudo-especular), en modelos con pocos polígonos que aproximan objetos curvos:



## Discontinuidades en la derivada

A veces pueden aparecer problemas parecidos a las bandas Mach, en este caso por exageración en la retina de las discontinuidades de primer orden (cambios bruscos en la pendiente de la iluminación)



(a la derecha aparece una ampliación, con el brillo y contraste aumentado)

### Sombreado en los píxeles.

En esta modalidad (*pixel shading*), el MIL se evalúa en cada pixel del viewport en el que se proyecta un polígono

- Requiere interpolar las normales asociadas a los vértices.
- Es computacionalmente más costoso que los anteriores, pero no cuando el número de polígonos visibles es del orden del número de pixels del viewport (o superior).
- Produce resultados de más calidad, hay muchos menos defectos por discontinuidades.
- Los resultados son más realistas incluso con pocos polígonos.
- La evaluación del MIL es la última etapa del cauce. Es decir, la evaluación del MIL se hace en cada instancia del *fragment shader*

---

Subsección 2.4.  
Sombreado en los píxeles.

---

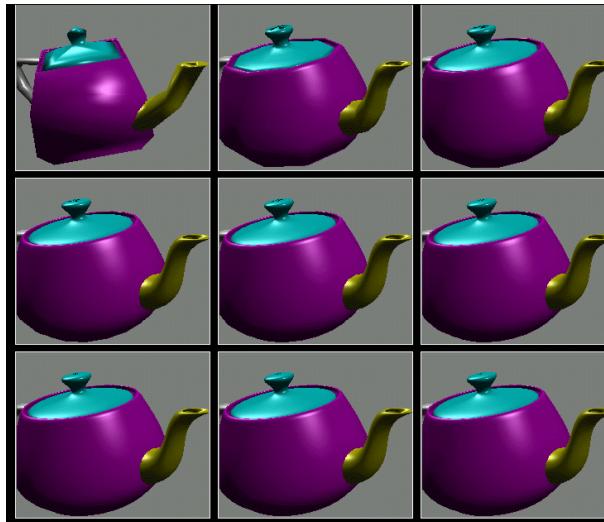
### Ejemplo de sombreado

Esta imagen se ha creado con sombreado en los pixels:



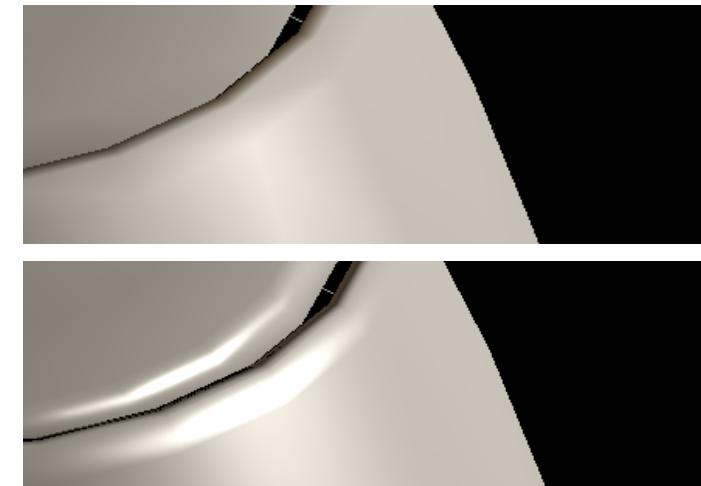
## Reproducción de zonas de brillo

Con este sombreado se reproducen los brillos incluso a baja resolución:



## Comparación de sombreado vértices y píxeles

Sombreado de vértices (arriba) y de píxeles (abajo), en iguales condiciones de iluminación, observador y atributos material:




---

### Sección 3.

#### **Luces, materiales y texturas en Godot.**

1. Luces en Godot.
2. Materiales y texturas en Godot.

---

##### Subsección 3.1.

#### **Luces en Godot.**

## Tipos de fuentes de luz en Godot

En Godot, las fuentes de luz emiten luz que se refleja en las superficies (con materiales con iluminación activada) y determina el color de los pixels. La luz puede provenir de varios tipos de fuentes en una escena:

1. Nodos del arbol de escena de tipo *fuente de luz*: son nodos de las clases `DirectionalLight3D`, `OmniLight3D` y `SpotLight3D`.
2. Luz ambiental en proveniente del entorno o fondo de la escena (un objeto de tipo `Environment`)
3. Luz reflejada indirecta, precalculada en una serie de puntos (usando la clase `ReflectionProbe`).
4. Luz reflejada indirecta, calculada usando técnica de *Iluminación global* (algoritmos implementados en las clases `LightmapGI`, o `VoxelGI` o `SDFGI`).
5. Desde la superficie de los objetos, cuando tienen definido un material con *color de emisión* y además están habilitada la iluminación indirecta en el espacio de pantalla (una aproximación a la iluminación global)

Veremos los dos primeros tipos de fuentes de luz.

## Propiedades de las luces (clase `Light3D`)

Las propiedades comunes a todos los tipos de fuentes de luz son:

**`light_color`**: color RGB de la luz emitida por la fuente (tipo `Color`).

**`light_energy`**: es un valor `float` que multiplica a `light_color` (1.0 por defecto)

**`shadow_enabled`**: si es `true`, la luz produce sombras arrojadas (es decir, no ilumina puntos desde los que la luz no es visible). Si vale `false`, todos los puntos de las superficies (cuya normal está de cara a la luz) son iluminados.

**`distance_fade_enabled`**: para luces puntuales o spot, indica si hay una distancia a partir de la cual la luz no ilumina los puntos. Por defecto es `false`. Si se pone a `true`, hay otras propiedades `float` que controlan el efecto:

**`distance_fade_begin`**: distancia a partir de la cual la luz empieza a disminuir.

**`distance_fade_length`**: distancia adicional (a partir de `distance_fade_begin`) en la que la luz se atenúa hasta cero.

## Nodos del árbol de tipo *fuente de luz*

En Godot se pueden situar, en un árbol de escena, nodos de tipo *fuente de luz*. La clase para esto es `Light3D`, derivada de `VisualInstance3D`, a su vez derivada de `Node3D`, es decir,

- Cada fuente tiene asociado su propio marco de coordenadas.
- Para usar una fuente de luz es necesario añadir un nodo `Light3D` a un árbol de escena.

Para cada tipo de nodo fuente de luz, existe una clase derivada de `Light3D`:

**Clase `DirectionalLight3D`**: luz direccional, como la luz del sol. La dirección de la fuente de luz es siempre el eje Z de su marco de coordenadas.

**Clase `OmniLight3D`**: luz puntual, que emite luz en todas las direcciones desde un punto, en concreto el origen de su marco de coordenadas.

**Clase `SpotLight3D`**: luz de tipo foco, que emite desde un punto (el origen del marco), con una distribución simétrica alrededor del eje Z del marco.

## Luces direccionales (clase `DirectionalLight3D`)

Un nodo de tipo `DirectionalLight3D` representa una luz **direccional**, la cual ilumina cualquier punto de una superficie desde una dirección igual para todos los puntos:

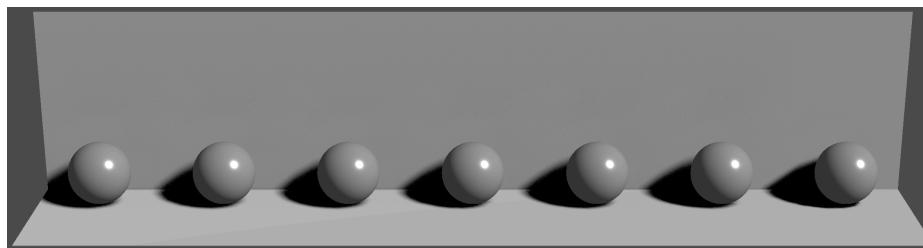
- Por defecto dicha dirección es la rama positiva del eje Z (el vector hacia la fuente de luz, `l`, es  $(0, 0, 1)$  para cualquier punto de la escena)
- Se puede modificar la dirección aplicando transformaciones de rotación al nodo (su posición es irrelevante).
- Por ejemplo, para apuntar a una dirección con coordenadas esféricas  $(\theta, \varphi)$ , se puede usar la rotación:

1. Rotar  $\theta$  radianes alrededor del eje horizontal X ( $\theta$  es el ángulo de `l` con el plano horizontal, negativo si es hacia arriba).
2. Rotar  $\varphi$  radianes alrededor del eje vertical Y.

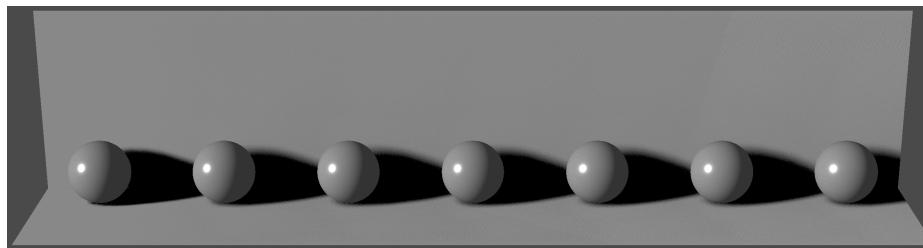
Se puede hacer directamente en el editor, o bien con GDScript usando los métodos `rotate_x` y `rotate_y` de la clase `Node3D`.

## Ejemplo de luz direccional

Fuente de luz direccional, rotada con  $\theta = -40^\circ$  y  $\varphi = 60^\circ$

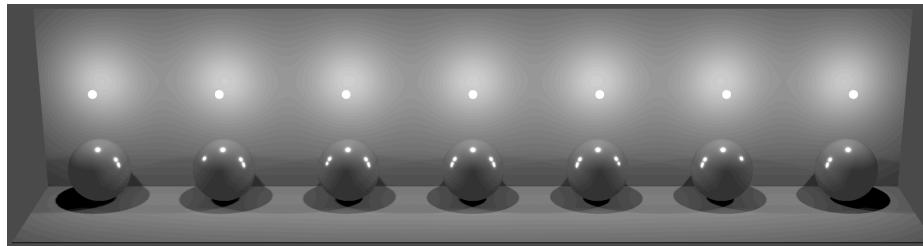


Fuente de luz direccional, rotada con  $\theta = -20^\circ$  y  $\varphi = -65^\circ$

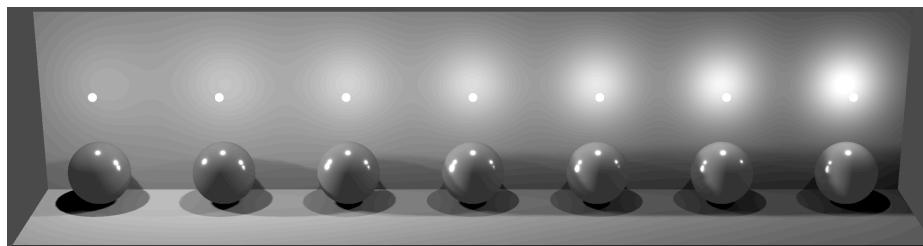


## Ejemplo de luces puntuales

Ejemplo de luces puntuales, con atenuación fija  $e = 1.5$ ,



Atenuación desde  $e = 0$  (izquierda) hasta  $e = 3.5$  (derecha):



## Luces puntuales (clase `OmniLight3D`)

Un nodo de tipo `OmniLight3D` representa una **luz puntual**, la cual ilumina un punto  $\hat{p}$  en una superficie desde un punto  $\dot{q}$  fijo del espacio, asociado a la fuente.

- El punto  $\dot{q}$  es el **origen del marco de coordenadas del nodo**. Se puede cambiar asignando `position`, en un script o en el editor.
- La intensidad no depende de la dirección  $\vec{r} = \dot{q} - \hat{p}$  pero sí puede depender de la distancia  $r = \|\dot{q} - \hat{p}\|$ . Se usa una propiedad `float`, con identificador `omni_attenuation` y llamada **atenuación**, si su valor es  $e$ , la intensidad será proporcional a:

$$g(r) = \frac{1}{r^e}$$

- Un valor  $e = 0$  hace que la luz no se atenúa con la distancia. El valor  $e = 2$  es **físicamente realista**, ya que en la naturaleza la luz se atenúa con el cuadrado de la distancia al punto de donde emana. A mayores valores del exponente  $e$ , más brillantes son las superficies cercanas a la fuente, y más oscuras las lejanas.

## Luces de tipo foco (spot) (clase `SpotLight3D`)

Las luces de tipo **foco** (o **spot**) son una clase de luces puntuales que emiten luz desde un punto  $\dot{q}$  fijo hacia cualquier otro punto  $\hat{p}$ , con una intensidad que:

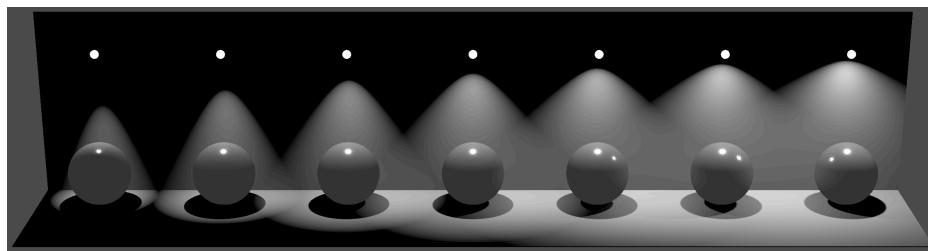
- Al igual que las luces puntuales, puede depender de una potencia de la distancia  $r = \|\dot{p} - \dot{q}\|$ , en este caso la propiedad se llama `spot_attenuation`, y se interpreta igual que `omni_attenuation` en `OmniLight3D`.
- Además, la intensidad depende del ángulo  $\theta = \arccos(\hat{e} \cdot \hat{r})$  entre:
  - ▶  $\hat{e} \equiv$  versor Z negativo del marco de coordenadas de la fuente, llamado **eje del foco**, es decir, el vector con coordenadas locales  $(0, 0, -1)$
  - ▶  $\hat{r} \equiv (\dot{p} - \dot{q})/r$ , el versor desde  $\dot{q}$  hacia  $\dot{p}$ .

en concreto, la intensidad será nula cuando  $\theta > \theta_{\max}$ , donde el ángulo  $\theta_{\max}$  se define con la propiedad `spot_angle` (llamada **apertura del foco**), que es un `float` con unidades en grados.

## Ejemplo de luces de tipo foco

Aquí vemos un ejemplo de varias luces de tipo foco, en las cuales el eje  $\hat{e}$  tiene coordenadas de mundo  $(0, -1, 0)$  (apunta en vertical hacia abajo), para ello se ha rotado cada nodo fuente un ángulo de  $-90^\circ$  entorno al eje  $X$  (lo cual lleva el versor  $Z$  negativo hasta el versor  $Y$  negativo).

El ángulo de apertura va desde  $20^\circ$  a la izquierda hasta  $60^\circ$  a la derecha, y la atenuación es 1.0:



## Mapa de entorno usando un panorama

Los mapas de entorno se pueden asociar a una escena mediante código o en el editor. Aquí vemos un ejemplo en el cual se asocia una imagen con un *panorama equirectangular* a una escena.

Para ello se añade un nodo **WorldEnvironment** y se usa un archivo de imagen **.jpg**. En GDScript se puede hacer asociando este script al nodo raíz de la escena:

```
func _ready() :  
  
    ## crear el objeto 'world environment' con el fondo en 'imagen.jpg'  
    var we := WorldEnvironment.new()  
    we.environment = Environment.new()  
    we.environment.background_mode = Environment.BG_SKY  
    we.environment.sky = Sky.new()  
    we.environment.sky.sky_material = PanoramaSkyMaterial.new()  
    we.environment.sky.sky_material.panorama = CargarTextura("imagen.jpg")  
  
    ## añadir el 'world environment' al árbol de escena  
    add_child( we )
```

## Iluminación desde mapas de entorno

En Godot los objetos pueden recibir iluminación ambiental desde un mapa de entorno (un objeto de tipo **Environment** asociado a la escena).

- El árbol de escena puede tener un nodo de tipo **WorldEnvironment** con diversos parámetros de configuración, entre ellos está la propiedad **environment** que referencia a un objeto de tipo **Environment**.
- Los objetos de tipo **Environment** pueden incluir un modo en el cual el entorno es de tipo **cielo**, referenciado en la propiedad **sky** (de la clase **Sky**).
- Un objeto de tipo **Sky** tiene la propiedad **sky\_material** de la clase **Material**, la propiedad es una referencia a un objeto de uno de estos tipos:
  - ▶ **PanoramaSkyMaterial**: usa una imagen panorámica de  $360^\circ \times 180^\circ$ . Se almacena en la propiedad **panorama** del material, de tipo **Texture2D**.
  - ▶ **ProceduralSkyMaterial**: degradados de color
  - ▶ **PhysicalSkyMaterial** : basado en física
  - ▶ **ShaderMaterial**: implementado en un shader.

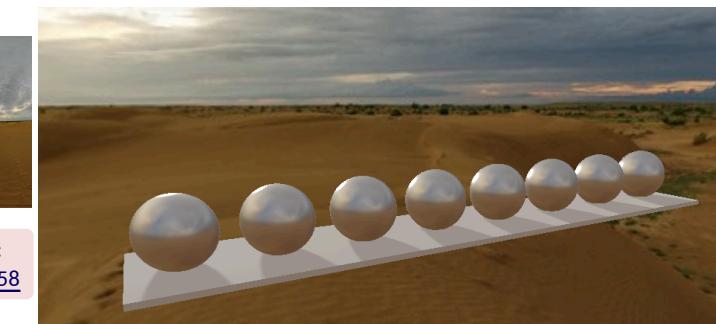
## Mapa de entorno usando un panorama

Los mapas de entorno iluminan las superficies de los objetos en la escena, su luz se refleja en las superficies tanto difusas como especulares.

Aquí vemos un ejemplo (a la derecha) de una escena con varias esferas, iluminadas con una **fuente direccional** (ver las sombras arrojadas) y además con un mapa de entorno basado en un *panorama equirectangular* (es la imagen a la izquierda).

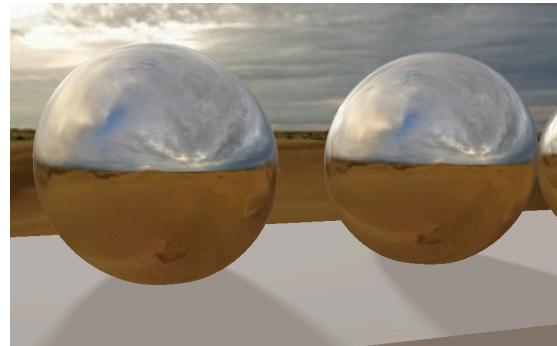


Imagen de Alexandre Duret-Lutz:  
[flickr.com/photos/gadl/595353758](https://flickr.com/photos/gadl/595353758)



La reflexión pseudo-especular o especular desde una superficie (iluminada por un mapa de entorno) únicamente incluye el mapa en sí, no los otros objetos.

En este ejemplo, las esferas especulares reflejan el mapa de entorno, pero no reflejan las otras esferas ni el suelo:



Para tener en cuenta estas reflexiones tendríamos que usar *Ray tracing*.

## Materiales

En los *engines* de videojuegos y las aplicaciones gráficas 3D en general, un **material** es un objeto con una serie de parámetros que determinan como refleja la luz la superficie de un objeto (3D usualmente, pero también puede ser 2D). En concreto, un *material* encapsula:

- Los **tipos de BRDFs** asociadas al material, incluyendo los **pesos** con los que se combinan cuando hay varias (difusa, especular perfecta, metálica, etc)
- Los **parámetros de cada BRDF**, por ejemplo, el **exponente de brillo** del modelo de Blinn-Phong, las **rugosidades** del modelo GGX, el **índice de refracción** para refracción y reflexión en dieléctricos, etc...
- El color base de la superficie (a menudo llamado *albedo*), que puede ser un **color plano único** o puede ser una **textura de color** (ya sea almacenada en un archivo o generada proceduralmente).
- Texturas que controlan parámetros adicionales, por ejemplo texturas para controlar los pesos citados arriba (que así pueden variar punto a punto), o bien texturas para **mapas de perturbación de la normal**.

---

### Subsección 3.2. Materiales y texturas en Godot.

---

## Materiales en Godot

En Godot, los materiales se representan con objetos de la clase **Material**, que es una clase base abstracta. Existen diversas clases derivadas de **Material**, entre las cuales las más usadas son:

- **BaseMaterial3D**: clase base abstracta para los materiales de las superficies de los objetos de tipo malla de triángulos. Define la mayoría de las propiedades relevantes. No se puede usar directamente (es una clase abstracta), sino que se usan instancias de su clase derivada **StandardMaterial3D**.
- **ShaderMaterial**: material definido mediante un shader personalizado (escrito en el lenguaje de shaders de Godot). Permite un control total sobre la apariencia del material.
- Los materiales para entornos de tipo *cielo* (clase **PanoramaSkyMaterial**, **ProceduralSkyMaterial**, **PhysicalSkyMaterial**), no se usan para mallas de triángulos.

## La clase StandardMaterial3D. Propiedades (1/2)

La clase **StandardMaterial3D** sirve para definir características de los materiales, y se implementa mediante un *fragment shader* interno específico. Las propiedades de esta clase son heredadas de su clase base abstracta **BaseMaterial3D**. Destacamos:

- **albedo\_color**: color base del material (tipo **Color**).
- **albedo\_texture**: textura con colores que multiplican a **albedo\_color** por distintos factores en cada punto, permite variar el color base del material de un punto a otro (tipo **Texture2D**).
- **metallic**: (tipo **float**), entre 0.0 y 1.0 determina define el carácter metálico del material.(*metallicidad*).
- **metallic\_specular**: (tipo **float**) entre 0 y 1 que define la intensidad del brillo pseudo-especular en materiales no metálicos.
- **roughness**: (tipo **float**), entre 0.0 y 1.0 que define la rugosidad del material

(continúa en la siguiente diapositiva)

## El modelo de iluminación de Godot

El color reflejado  $I$  (antes de mult. por el color de la luz) puede escribirse así:

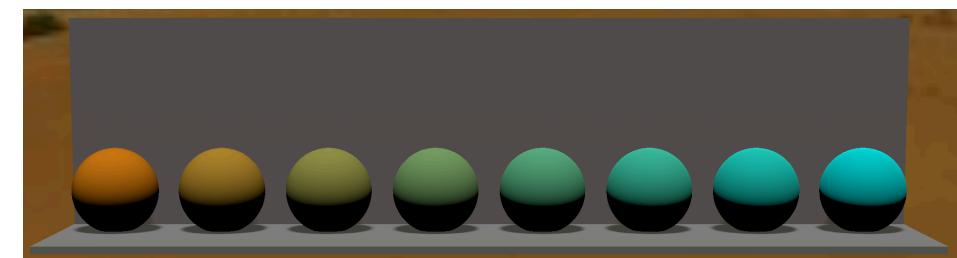
$$I = e + a + (1 - m)(b \cdot d + s \cdot p_\alpha) + m \cdot b \cdot p_\alpha$$

- $e$  ≡ color en **emission** (**color emitido**)
- $a$  ≡ **color ambiental** reflejado, depende del mapa de entorno.
- $m$  ≡ factor real en **metallic** (**factor de metalicidad**)
- $s$  ≡ factor real en **metallic\_specular** (**factor pseudo-especular**)
- $b$  ≡ **color base**, es **albedo\_color**, multiplicado por **albedo\_texture** (si existe).
- $d$  ≡ **reflectividad difusa** según el modelo de Burley (o de Lambert).
- $p_\alpha$  ≡ **componente pseudo-especular**, es la suma de la reflectividad de una BRDF pseudo-especular (por defecto la BRDF GGX anisotrópica), más el color reflejado del mapa de entorno (si existe). Está afectada por la rugosidad  $\alpha$
- $\alpha$  ≡ valor real en **roughness** (**rugosidad**), va desde 0 (especular perfecto, como un espejo), hasta 1 (superficie muy rugosa, casi mate).

## La clase StandardMaterial3D. Propiedades (2/2)

Más propiedades relevantes de la clase **StandardMaterial3D** (**BaseMaterial3D**):

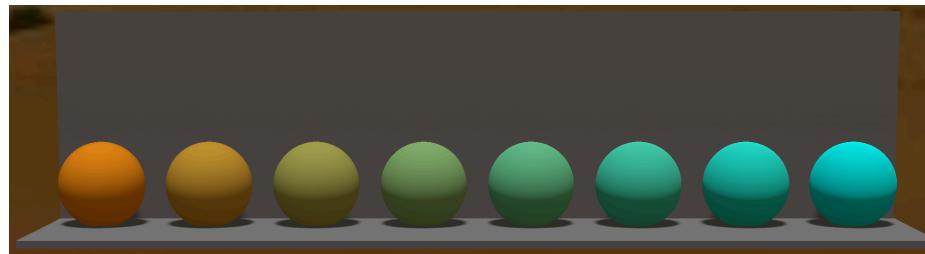
- **emission\_enabled**: si es **true**, la superficie del objeto emite luz por sí mismo, si es **false** no emite luz. En la práctica este color se suma al reflejado, pero no afecta a otros objetos si no se activa algún método avanzado de iluminación.
- **emission**: valor **Color** con la luz emitida, cuando **emission\_enabled** es **true**.
- **transparency**: modo de transparencia, permite desactivar transparencias (si se usa el valor **TRANSPARENCY\_DISABLED**) o activarlas, si se activa, entre otras opciones, se puede ligar a la componente **alpha** del color base (usando el modo **TRANSPARENCY\_ALPHA**).
- **disable\_ambient\_light**: si es **true**, el material no recibe luz ambiental del mapa de entorno.



Aquí vemos una serie de esferas puramente difusas, con el color base  $b$  (**albedo\_color**) variando desde naranja (1.0, 0.6, 0.1) hasta verde (0.0, 1.0, 1.0), todo iluminado con una luz direccional, y con  $a$  a cero (**disable\_ambient\_light** puesto a **true**).

## La componente ambiental y la difusa

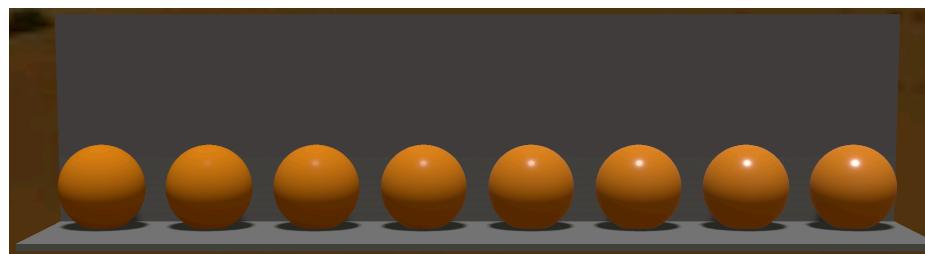
La componente ambiental  $a$  puede habilitarse si hay un mapa de entorno en la escena, en ese caso, las mismas esferas de la imagen anterior tendrán una cantidad de luz reflejada desde el entorno, lo cual es relevante en las partes que no dan a la fuente de luz (y donde  $d$ , por tanto, es nulo).



Aquí vemos la misma escena con `disable_ambient_light` a `false`, de modo que las esferas reciben luz ambiental desde el mapa de entorno.

## Combinación difusa y pseudo-especular

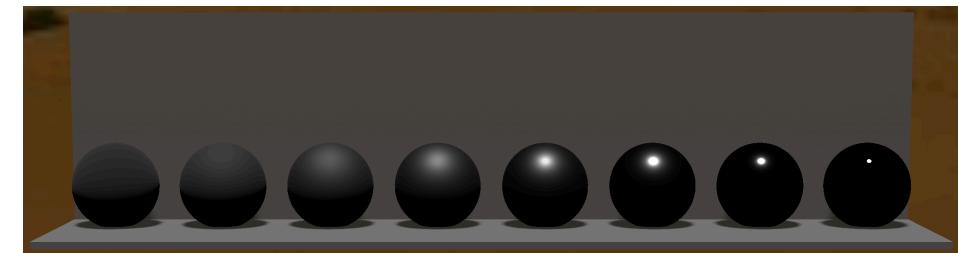
La componente pseudo-especular puede combinarse con la difusa para simular un material brillante (no afectado por el color base  $b$ ) sobre un substrato difuso (afectado por  $b$ ). En esta imagen variamos  $s$  desde 0 a la izquierda, hasta 0.6, con rugosidad  $\alpha = 0.3$ . El color base  $b$  se mantiene a naranja ( $1.0, 0.6, 0.1$ )



Se ha habilitado la luz ambiental.

## La componente pseudo-especular

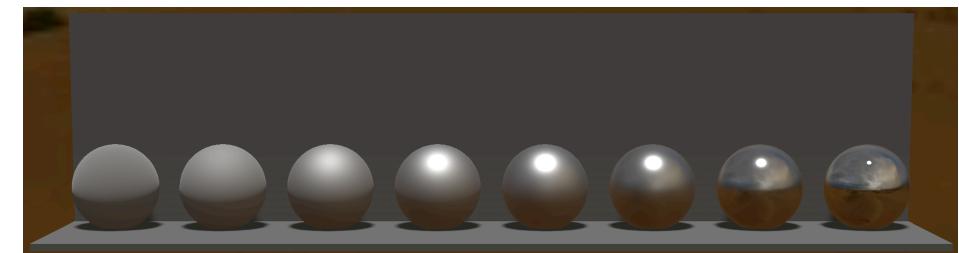
El factor  $s$  (propiedad `metallic_specular`) controla la cantidad de luz reflejada por la componente pseudo-especular. Este producto no está multiplicada por el color base, así que solo depende del color de la fuente de luz. Se puede observar haciendo  $m = 0$  y  $b = e = a = (0, 0, 0)$ . Puesto que  $I = s \cdot p_\alpha$ , esta componente depende de la rugosidad  $\alpha$ .



En esta serie vemos la componente pesudo-especular  $s \cdot p_\alpha$ , con  $s$  puesto a 1 y la rugosidad  $\alpha$  bajando desde 1 hasta 0 (de izquierda a derecha). No hay luz ambiental,  $a = (0, 0, 0)$ .

## Materiales metálicos

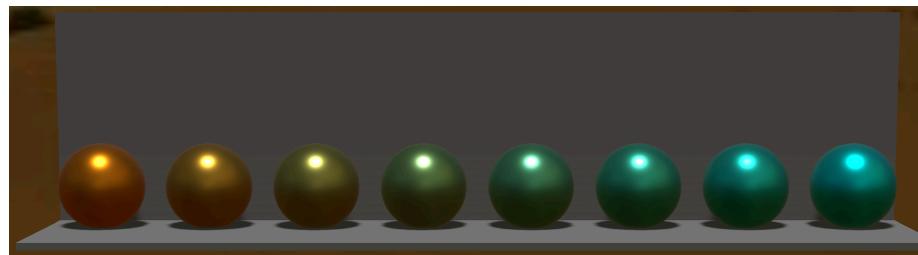
Para lograr que la componente pseudo-especular esté afectada por el color base  $b$ , podemos hacer  $m = 1$  y entonces no se usa el valor  $s$ , ya que  $I = m \cdot b \cdot p_\alpha$ . Se consigue un aspecto metálico (a partir de cierto  $\alpha$ ), más o menos pulido, según  $\alpha$ . Para  $\alpha = 0$ , se tiene un material reflectante como un espejo.



En esta serie hacemos  $b = (1, 1, 1)$  y variamos la rugosidad desde  $\alpha = 1$  a la izquierda hasta 0 a la derecha. Se tiene en cuenta la luz del entorno. El aspecto resultante es un aspecto metálico

## Materiales metálicos afectados del color base

En esta serie se ha mantenido  $m = 1$  y  $\alpha = 0.3$ . Aquí de nuevo se varía el color base  $b$  desde naranja a verde.



Este tipo de materiales puede simular metales como el oro u otros metales tintados.

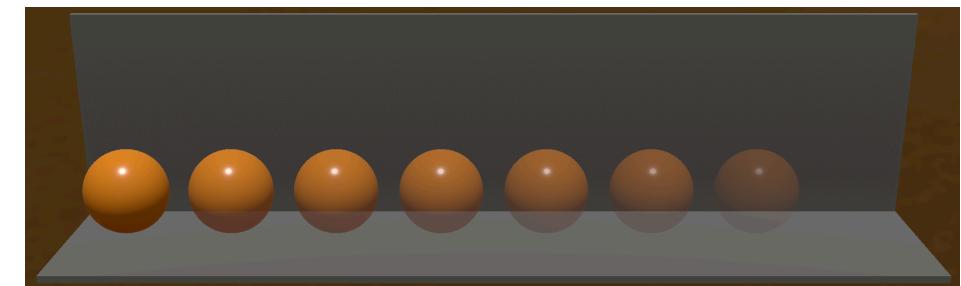
## Objetos parcialmente transparentes

Además de los materiales, los nodos (de clases derivadas de `GeometryInstance3D`, es decir, todas las mallas, como `MeshInstance3D` u otras):

- Tienen una propiedad llamada `transparency`, es un `float` entre 0 y 1.
- Define la transparencia del objeto, pero se interpreta al revés que la componente `alpha` del albedo, ya que el valor 0 en `transparency` implica **totalmente opaco** y el valor 1 **totalmente transparente**.
- Este valor se compone con la transparencia del albedo.
- Los nodos con un valor de `transparency` mayor que 0 (es decir, parcialmente transparentes) **sí arrojan sombras**.
- Esta transparencia se ignora si se genera la aplicación en el modo de render de `compatibilidad` o para dispositivos móviles (solo se tiene en cuenta con el modo de render `Forward +`)

## Materiales parcialmente transparentes

En Godot se puede hacer que un material sea parcialmente transparente, para ello se usa la propiedad `transparency` de la clase `StandardMaterial3D`. Si se activa la transparencia, el material puede usar la componente `alpha` del color base (`b` debe ser una tupla RGBA con 4 `float`) para definir el grado de transparencia, es un valor entre 0.0 (totalmente transparente) y 1.0 (totalmente opaco).



Aquí se la componente `alpha` de `b` varía desde 1 a la izquierda hasta 0 a la derecha. Con este modo, **no se tiene en cuenta la refracción, ni se proyectan sombras**.

## Texturas en Godot. La clase `Texture2D`

En Godot, la clase `Texture2D` representa una imagen, con un rectángulo bidimensional con un color (albedo) asociado cada punto del mismo. Godot incorpora numerosas clases derivadas de `Texture2D`, podemos destacar estas:

- **ImageTexture**: usa una imagen cargada desde un archivo (PNG, JPG, etc) o creada en memoria. Es el tipo de textura **más frecuentemente usado**.
- **ViewportTexture**: se usan los pixels de un nodo `Viewport` (una ventana o una parte de una ventana sobre la que se visualiza una imagen).
- **GradientTexture2D**: degradado de color.
- **NoiseTexture2D**: colores calculados usando algoritmos de generación de ruido (ruido procedural) con un objeto tipo `Noise` (ruido Perlin, Cellular, Simplex).
- **CameraTexture**: usa la imagen capturada por una cámara del dispositivo que ejecuta la aplicación.
- **AtlasTexture**: textura que usa una región rectangular de otra textura (útil para optimizar el uso de memoria).

## Asociación de materiales a objetos

Cualquier nodo derivado de **GeometryInstance3D** (por ejemplo, **MeshInstance3D**):

- tiene una propiedad llamada **material\_override**, que es una referencia a un objeto de la clase **Material** (o a una clase derivada).
- adicionalmente, se puede definir su propiedad **material\_overlay**, es un material que se *superpone* al material definido en **material\_override**

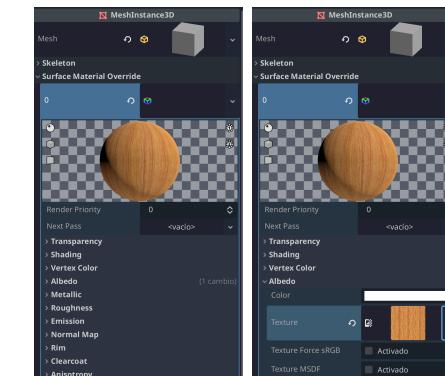
Si un nodo no tiene por defecto ningún material asignado, en ese caso se usa el material asociado a las mallas del nodo. En Godot,

- Los objetos de tipo **Mesh** (**ArrayMesh**, **IndirectMesh** y **PrimitiveMesh**) pueden tener una o varias *superficies* (mallas), y cada una puede tener asignado un material distinto.
- Ese material se puede consultar y cambiar usando los métodos **surface\_get\_material(i)** y **surface\_set\_material(i, m)** de la clase **Mesh**

Fin de transparencias.

## Asignación de materiales y texturas en el editor

El editor de Godot puede usarse para crear un material específico para un nodo (de tipo derivado de **GeometryInstance3D**), y asignar texturas a ese material.



Aquí vemos dos capturas del panel de propiedades de un nodo tipo **MeshInstance3D**, en la primera se ven las propiedades dentro del **Surface Material Override**, y en la segunda igual, pero con las propiedades de textura desplegadas (en el apartado **Albedo** ⇒ **Texture**)