

Informática Gráfica

Iluminación

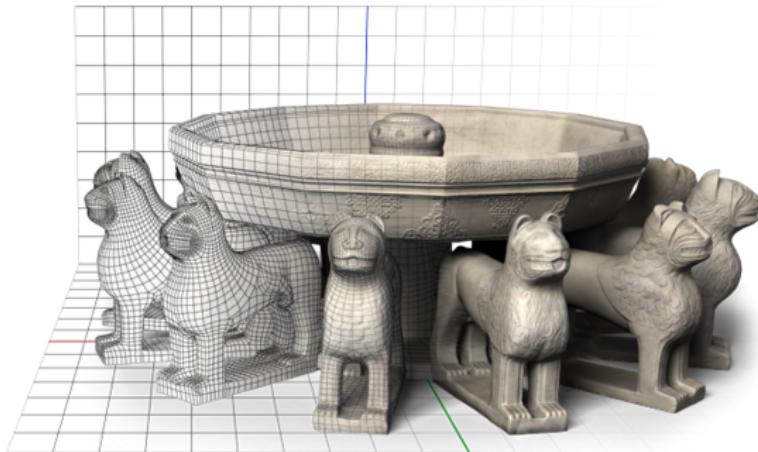
Juan Carlos Torres
Grupos C y D

Dpt. Lenguajes y Sistemas Informáticos
ETSI Informática y de Telecomunicación
Universidad de Granada

Curso 2024-25

Introducción

Introducción



- Para generar una imagen de una escena es necesario tener
 - Los objetos que forman la escena: Geometría, color y textura
 - La cámara que ve la escena
 - Luz que ilumina la escena
- En este tema veremos cómo especificar el color, la textura y las luces, y como se calcula el color en la imagen en función de ellos.

Introducción

El cálculo de iluminación depende de las fuentes de luz, de los materiales y de la posición del observador.

- Dicho cálculo se puede hacer emulando la iluminación real que ocurre en los objetos de la naturaleza.
- Para ello es necesario diseñar un **modelo de iluminación**, un modelo formal que incluya las características relevantes del color de los polígonos.
- OpenGL (Legacy) incorpora un modelo sencillo y computacionalmente eficiente para esto.

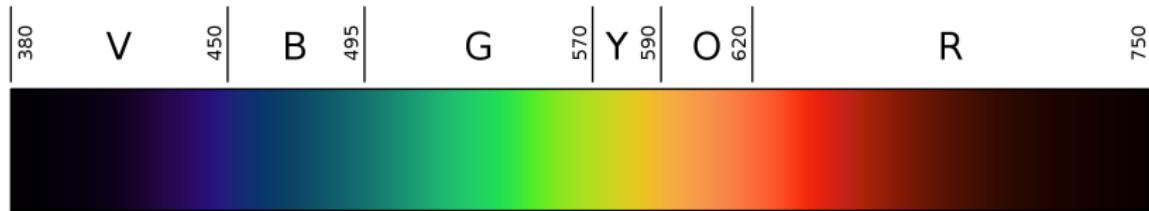
Antes de estudiar las cámaras, las luces y las propiedades de los objetos vamos a estudiar la luz y el color.

Luz y color

La luz como radiación electromagnética

La luz que observamos es radiación electromagnética (variaciones periódicas del campo eléctrico y magnético) de naturaleza similar a las ondas que se usan para los móviles, wifi, radio y televisión:

- El sistema visual humano ha evolucionado para percibir esa radiación solo cuando su longitud de onda λ está aprox. entre 390 y 750 nanómetros (\equiv *espectro visible*).
- Físicamente, la radiación se describen como algo que tiene características de onda y de corpúsculo a la vez (modelos complementarios).



(figura obtenida de: http://en.wikipedia.org/wiki/Visible_spectrum)

Brillo y color de la radiancia

Desde un punto **p** en una dirección **v** pueden emitirse o reflejarse una gran cantidad de fotones con longitudes de onda distintas.

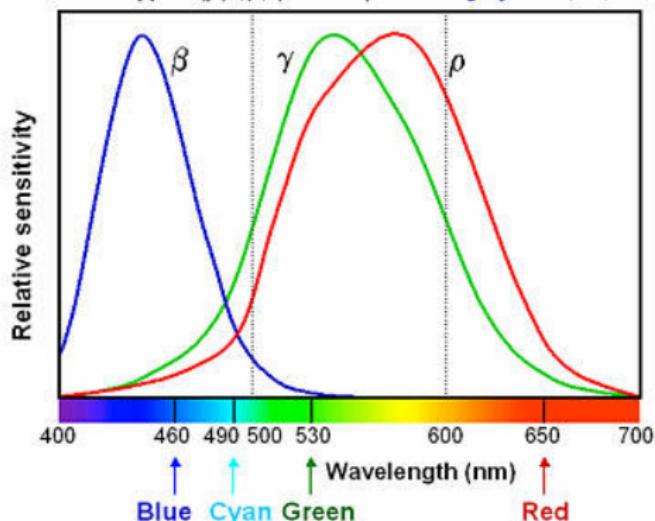
- La intensidad o brillo de la luz depende de la cantidad de fotones (es decir, de la radiancia total sumada en todas las longitudes de onda)
- El color con el que percibimos la luz depende de las distribución de las longitudes de onda de los fotones en el espectro visible.
- Nuestra retina tiene dos tipos de celulas fotoreceptoras: bastones (visión periférica y nocturna sin color) y conos (visión de detalle, sensibles al color).
- Tenemos tres tipos de conos, cada uno es mas sensible a una longitud de onda distinta (rojo, verde y azul).

Percepción del color

- Percibimos un color como una combinación de estímulos de los tres sensores.
- Hay muchas combinaciones de radiación que producen los mismos estímulos (colores metámeros).

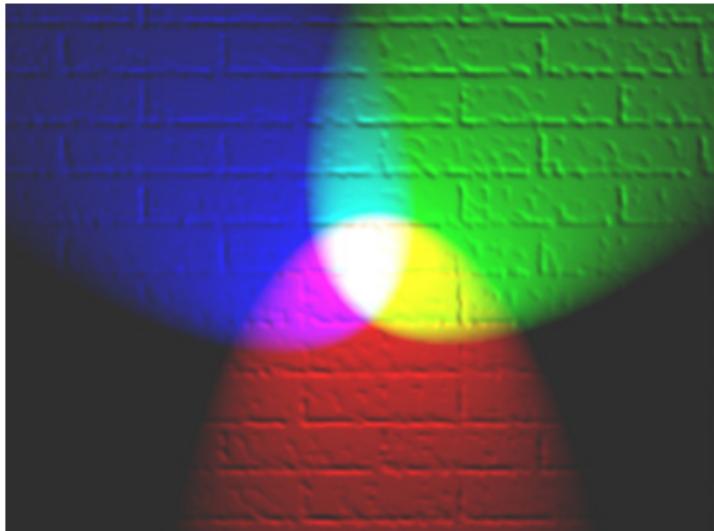
Human spectral sensitivity to color

Three cone types (ρ , γ , β) correspond roughly to R, G, B.



Mezcla aditiva de primarios

Podemos usar una *mezcla aditiva* (suma ponderada) de tres primarios RGB para producir una señal arbitraria.



☞ http://en.wikipedia.org/wiki/RGB_color_model

Reproducción de ternas RGB

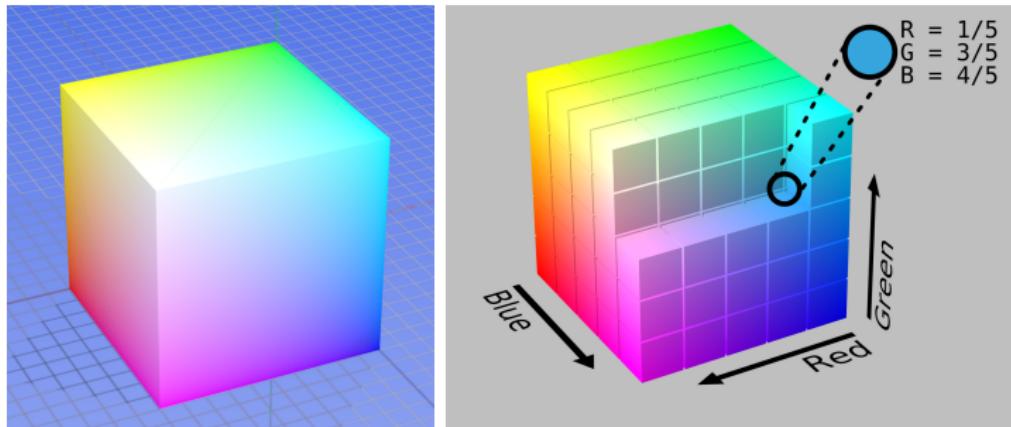
Un dispositivo de salida de color (monitor o proyector) tiene asociados tres primarios RGB

- Como consecuencia, cualquier color reproducible en un dispositivo se puede representar por una terna (r, g, b) , con $0 \leq r, g, b \leq 1$.
- El valor 0 indica que el correspondiente primario no aparece.
- El valor 1 representa la máxima potencia del dispositivo para cada primario.
- Una misma terna (r, g, b) produce tonos de color ligeramente distintos en dispositivos distintos.
- Con frecuencia se utilizan cuatro componentes (r, g, b, α) , en la que α es la opacidad.

¿Y las impresoras?

El espacio RGB

Al conjunto de todas las ternas RGB con componentes entre 0 y 1 se le llama **espacio de color RGB**, y se puede visualizar como un cubo 3D con colores asociados a cada punto del mismo.

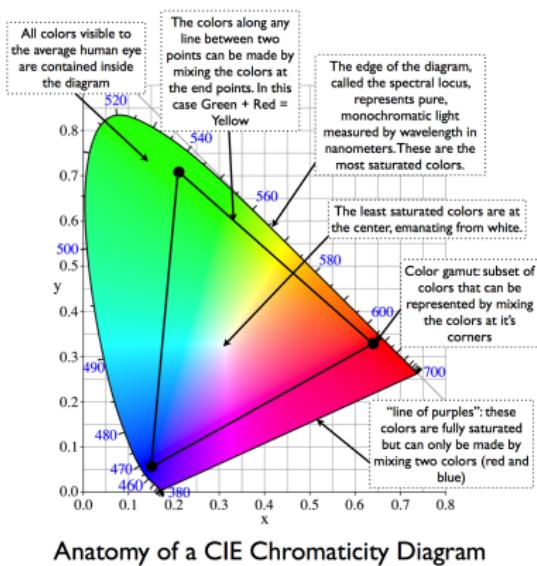


☞ http://en.wikipedia.org/wiki/RGB_color_model

El espacio RGB no es el único esquema para representar computacionalmente los colores, pero sí el más usado hoy en día.

Diaggrama CIE

Espacio de color CIE separa brillo y cromaticidad. El Diagrama CIE es una representación 2D de la cromaticidad. Los colores representables en un dispositivo son los incluidos en el triángulo formado por sus tres primarios.



Dispositivos

El color que se obtiene con una terna RGB en un dispositivo de salida depende de los primarios RGB que se usen en dicho dispositivo y del brillo máximo que pueda alcanzar:

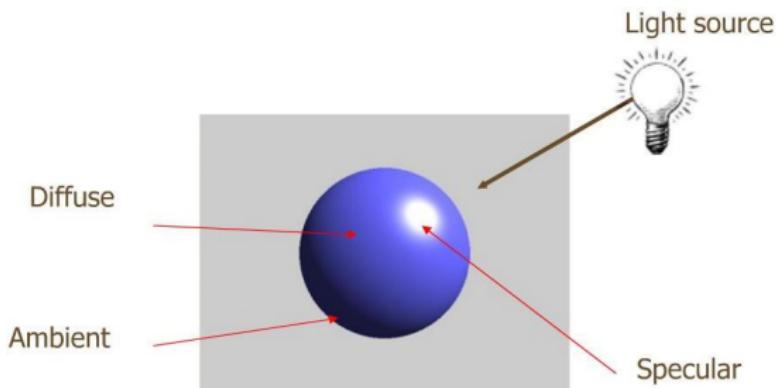


Illuminación y materiales

Modelo de Iluminación Local

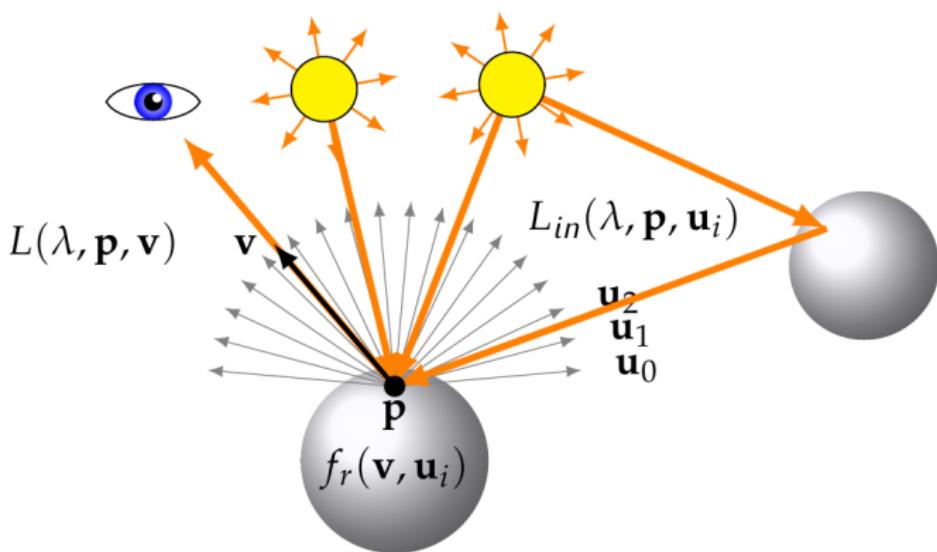
El color y la intensidad con la que vemos un punto depende de la radiancia emitida, que depende de las fuentes de luz y de la reflectividad de la superficie, a la que contribuyen todos los componentes:

$$L(\mathbf{p}) = L_a(\mathbf{p}) + L_d(\mathbf{p}) + L_s(\mathbf{p}) + L_e(\mathbf{p}) \quad (1)$$



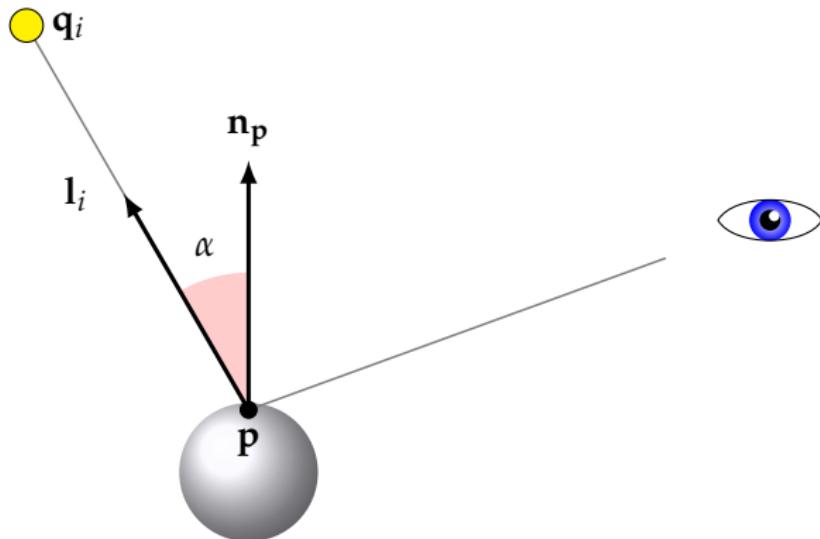
Reflexión local en un punto

Para generar la imagen es necesario determinar la luz que llega al observador desde cada punto de la escena. Para realizar el cálculo de forma interactiva es necesario simplificar el problema.



Componente difusa

La componente difusa depende del ángulo (α) entre el vector a la fuente de luz \mathbf{l}_i y el vector normal \mathbf{n}_p . Concretamente del $\cos(\alpha)$. Si los dos vectores están normalizados $\cos(\alpha) = \mathbf{l}_i \cdot \mathbf{n}$.



Componente difusa

La **componente difusa** modela como se refleja la luz en los objetos mate o difusos:

- La componente **depende** de la posición y orientación de la fuente de luz (es distinta según como esté orientada la fuente respecto de la superficie en \mathbf{p} , es decir, depende de \mathbf{n}_p y \mathbf{l}_i),
- **no depende** de la dirección en la que miramos (el punto se ve de un color igual desde cualquier dirección que lo miremos).

La expresión concreta de f_d es:

$$f_d(\mathbf{p}, i) = S_{iD}(i) \max(0, \mathbf{n}_p \cdot \mathbf{l}_i) \quad (2)$$

Siendo $S_{iD}(i)$ la intensidad con la que contribuye la luz i a la iluminación difusa.

Orientación de la superficie

Aquí se ilustran tres posibles casos:

$$90^\circ < \alpha$$

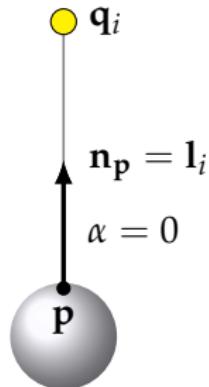
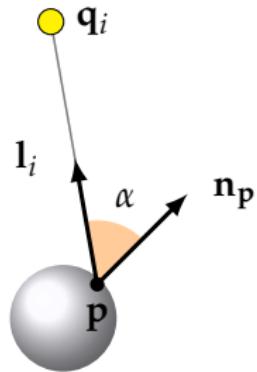
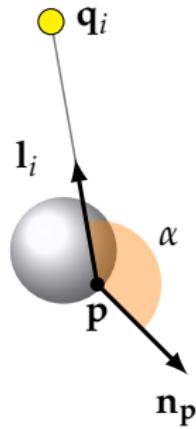
$$0 > \cos(\alpha)$$

$$0^\circ < \alpha < 90^\circ$$

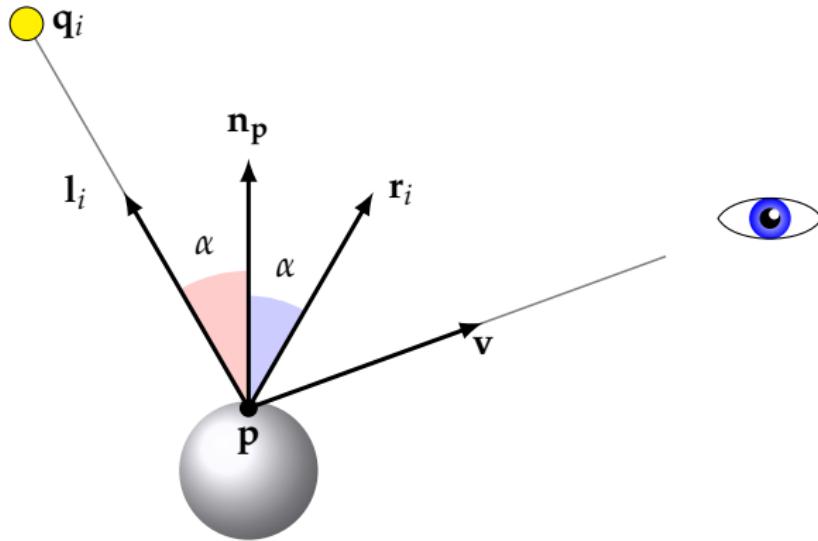
$$1 > \cos(\alpha) > 0$$

$$\alpha = 0^\circ$$

$$\cos(\alpha) = 1$$



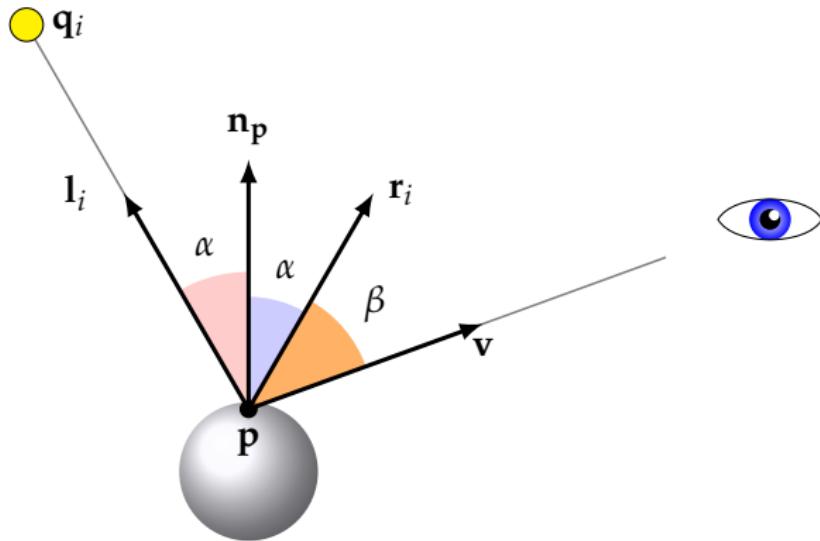
Componente especular



La componente **especular** modela como se refleja la luz en los objetos brillantes, en los cuales dichas zonas brillantes dependen de la posición del observador:

- La componente **depende** de la posición y orientación de la fuente de luz
- **también depende** de la dirección en la que miramos p (el punto p se ve de un color diferente según la dirección en la que lo veamos).

Componente especular



r_i es la dirección simétrica a l_i respecto a n_p . El valor $r_i \cdot v$ es el coseno del ángulo β que hay entre la dirección de máximo brillo r_i y la dirección v hacia el observador. Cuando $r_i = v$ entonces $\beta = 0^\circ$, $\cos(\beta) = 1$, y el brillo es máximo.

Componente especular

La expresión de f_s es

$$f_s(\mathbf{p}, \mathbf{v}, i) = S_{iS}(i)(\max(0, \mathbf{r}_i \cdot \mathbf{v}))^e \quad (3)$$

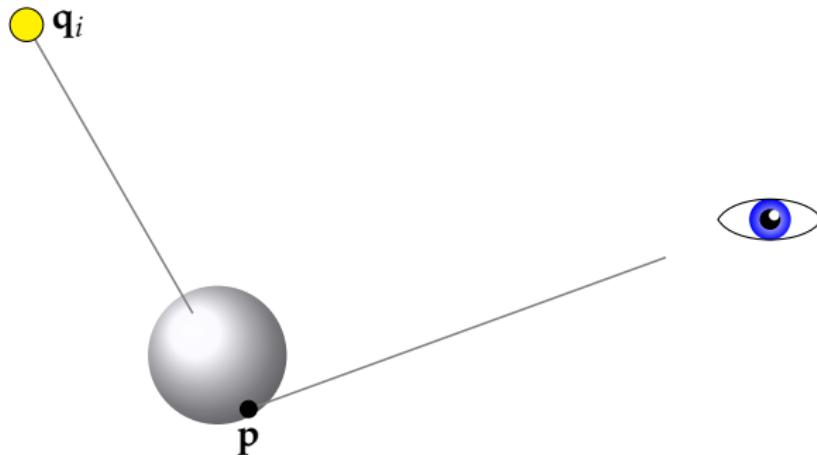
si $(\mathbf{n}_p \cdot \mathbf{l}_i) > 0$ (superficie iluminada) y 0 en caso contrario.

donde $S_{iS}(i)$ es la intensidad con la que contribuye la luz i a la componente especular y e es el exponente del brillo (que modela la amplitud de la mancha de brillo).

$$r_i = 2(l_i n_p)n_p - l_i \quad (4)$$

Componente ambiental

La componente ambiente es constante, simula la iluminación de fondo y evita que las partes no iluminadas directamente se vean negras



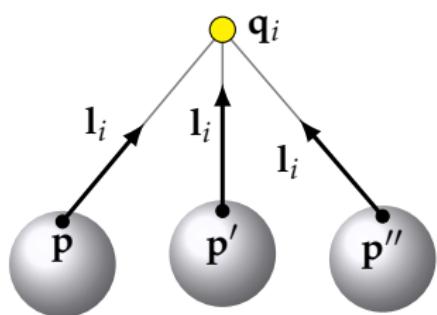
La expresión de f_a es

$$f_a(\mathbf{p}, i) = S_{iA}(i) \quad (5)$$

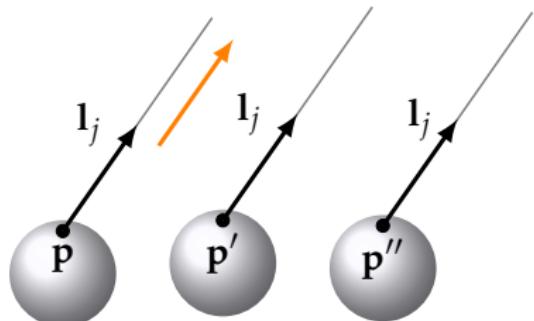
Siendo $S_{iA}(i)$ la luminosidad con la que contribuye la luz i a la componente ambiental.

Posición o dirección de las luminarias

Fuente posicional (i)



Fuente direccional (j)



- Posicional: la dirección l_i es distinta para cada punto p considerado. Es necesario recalcularla cada vez que se evalua el MIL.
- Direccional: La dirección l_j es igual para todos los puntos p considerados. Es una constante.

Sombreado

Sombreado

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. En este caso se interpolan los atributos de los vértices (p.e. la normal) para obtener los atributos del fragmento.

Sombreado plano

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

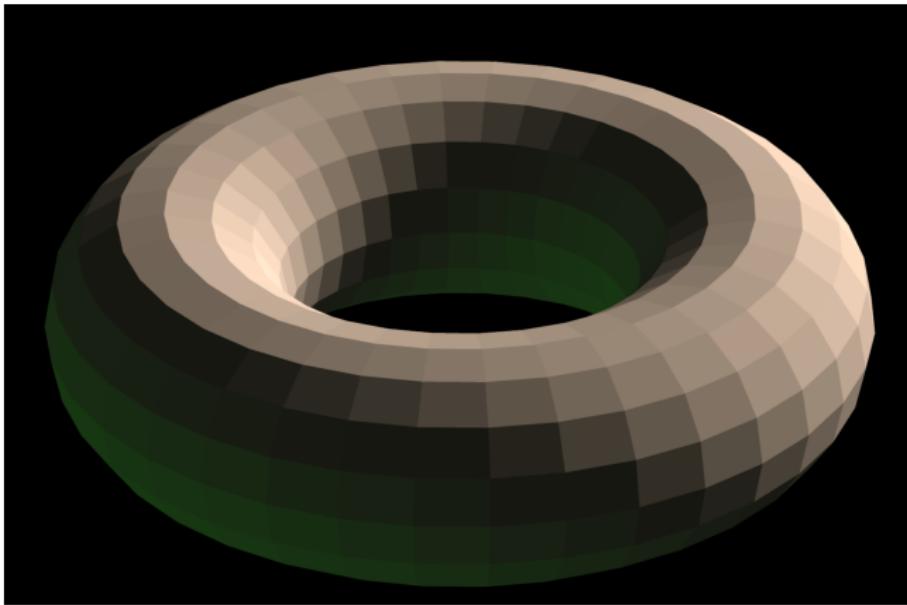
- Se selecciona un punto cualquiera p de cada polígono, típicamente se usa un vértice, pero podría ser cualquier otro.
- Se calcula el vector al observador v en p .
- Se calcula el vector a las fuentes de luz l_i en p .
- Se usa la normal al polígono n_p .

Este método solo es apropiado para objetos poliédricos.

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.



Sombreado en los vértices

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

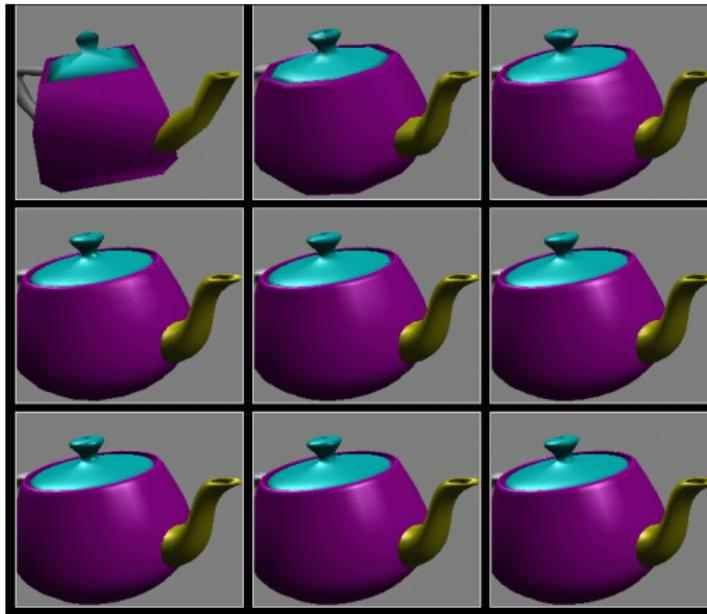
- La evaluación del MIL produce un color para cada vértice.
- La posición p coincide con la posición del vértice.
- Los valores en los vértices se usan para interpolar los colores de los pixels del polígono.

Para objetos no poliédrico los resultados son más realistas que con sombreado plano. En estos casos la normal en los vértices n_p se calcula como el promedio de las normales de los polígonos adyacentes al vértice.

La eficiencia en tiempo es parecida al sombreado plano.

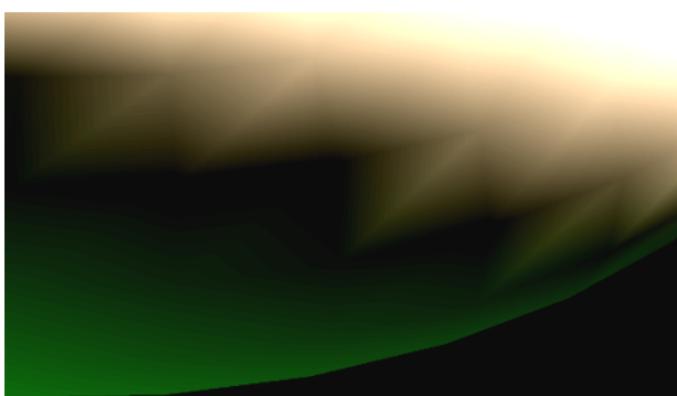
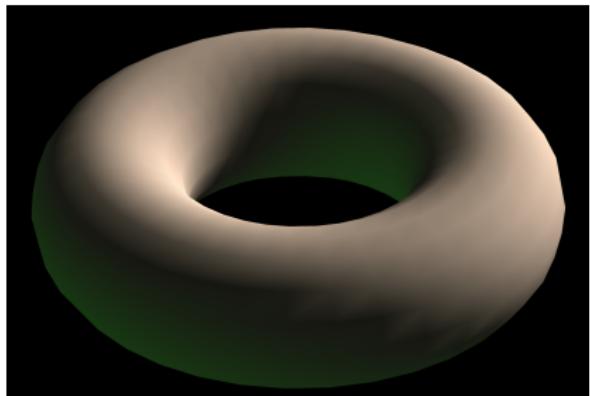
Pérdida de zonas brillantes

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 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 evalua en cada pixel del viewport en el que se proyecta un polígono

- La normal en los pixel se calcula interpolando las normales asociadas a los vértices.

La evaluación del MIL es la última etapa del cauce. Es computacionalmente más costoso que los anteriores, salvo que el número de polígonos visibles sea del orden del número de pixels del viewport o superior.

Produce resultados más realistas incluso con pocos polígonos.

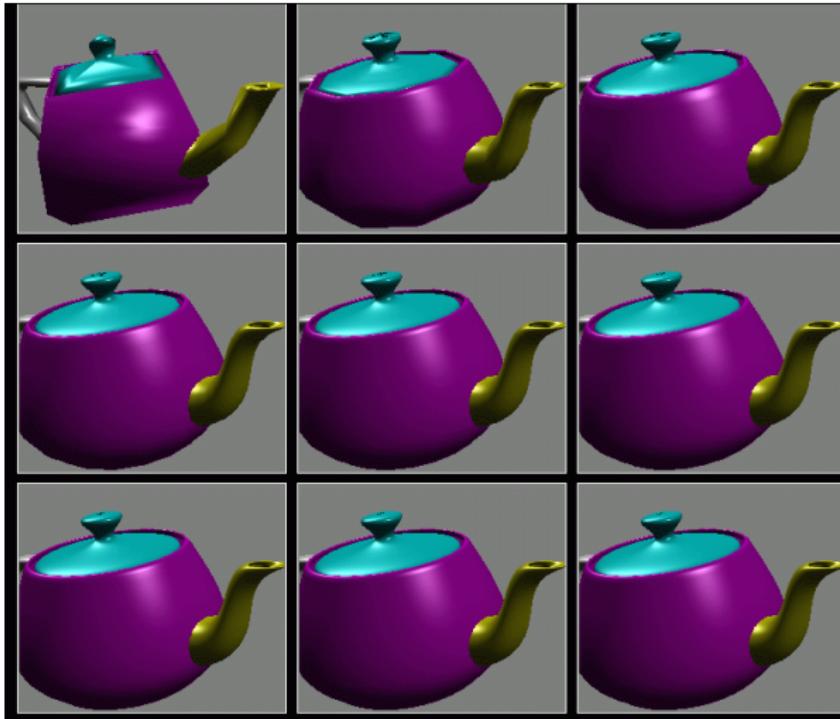
Ejemplo de sombreado

Esta imagen se ha creado con sombreado en los pixels:



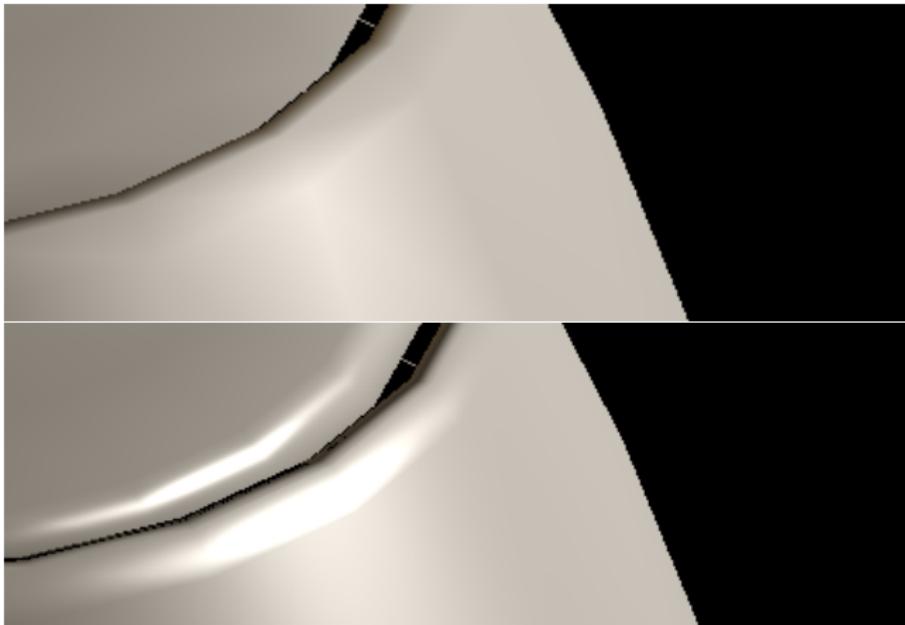
Reproducción de zonas de brillo

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:



Illuminación en OpenGL Legacy

Modelo de Iluminación Local de OpenGL Legacy

La librería OpenGL como parte de la funcionalidad fija (pre-programada), incluye una implementación de un modelo de iluminación simple, que tiene en cuenta la iluminación directa de las fuentes de luz, calculando:

- Reflexión difusa (producida en superficies mates)
- Reflexión especular (producida en superficies reflectantes)
- Iluminación ambiente (promedio de iluminación indirecta)
- Emisividad (luz emitida por la superficie del objeto)

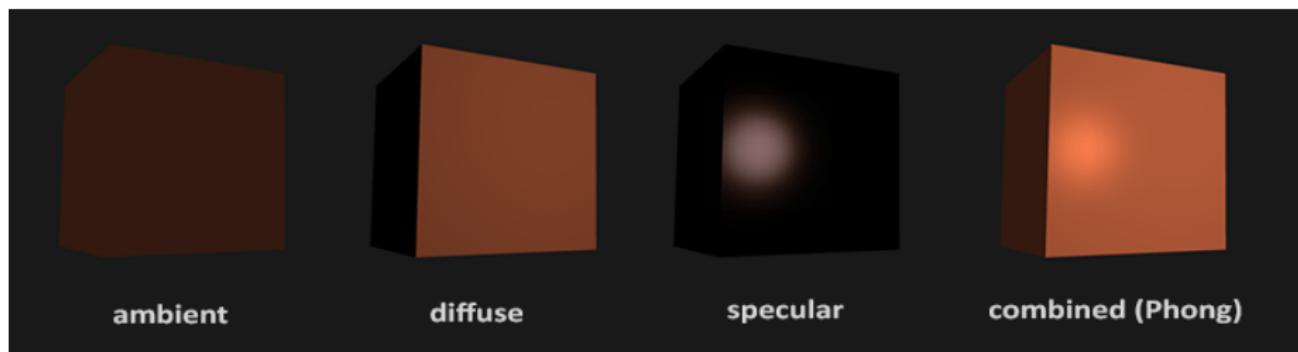


Figura: <https://learnopengl.com/Lighting/Basic-Lighting>

Iluminación con OpenGL

El comportamiento de OpenGL depende de si la evaluación de la iluminación está activada o no lo está:

- Con la iluminación desactivada, el color de las primitivas dibujadas depende de una terna RGB del estado interno, que se modifica con `glColor`.
- Con la iluminación activada la iluminación se evalúa usando un modelo de iluminación local (MIL) y unos parámetros incluidos en el estado de OpenGL, y el color resultante obtenido se usa en lugar del especificado con `glColor`.
- Los colores, materiales e intensidades se representan en `rgb` o `rgba` con valores normalizados entre 0 y 1.

El modelo de iluminación supone que:

- la reflexión es difusa y/o especular puras
- La iluminación incidente de reflejos múltiples se puede aproximar por una radiación de fondo.

Efecto de las simplificaciones

Aquí se observa una escena con iluminación compleja (izquierda) y simplificada usando OpenGL Legacy (derecha)

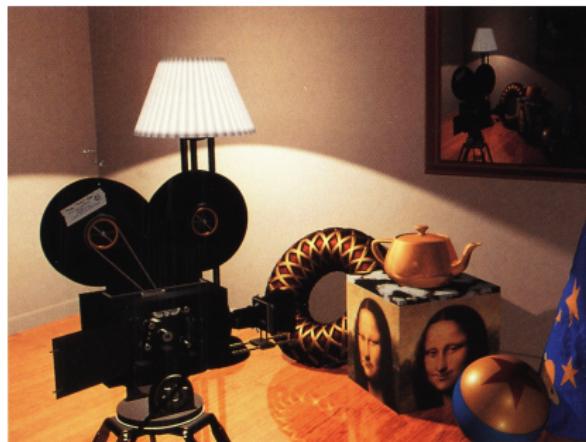


Imagen obtenida de:  Computer Graphics: Principles and Practice in C (2nd Edition)
Foley, van Dam, Feiner, Hughes.

Modelo de Iluminación Local de OpenGL legacy

De cada fuente de luz (i) se define su contribución a la iluminación difusa (S_{iD}), especular (S_{iS}) y ambiente (S_{iA}). De igual modo cada material tiene asociadas tres reflectividades (M_D, M_S, M_A):

El color del punto es el resultado de la suma de las distintas componentes para todas las fuentes de luz y la emisividad:

$$L(\mathbf{p}) = \sum_i M_A(\mathbf{p}) \cdot S_{iA} + \sum_i M_D(\mathbf{p}) \cdot S_{iD} + \sum_i M_S(\mathbf{p}) \cdot S_{iS} + M_E(\mathbf{p}) \quad (6)$$

Tanto las intensidades de las fuentes (S) como las reflectividades y emisividades (M) se indican con valores rgb o $rgba$, con componentes en el rango 0-1.

Modelo de Iluminación Local de OpenGL Legacy

Con este modelo de cada fuente de luz i se puede diferenciar su contribución a la iluminación ambiente, difusa y especular.

Las reflectividades m indican la fracción de luz incidente que se refleja. Este valor es normalmente diferente para cada longitud de onda. Por este motivo las reflectividades, intensidades y radiancias se representan en RGB.

Si la suma en un punto para alguna componente es mayor que uno se trunca el valor de esa componente.

Parámetros del Modelo de Iluminación Local (MIL)

En su estado interno, OpenGL mantiene un conjunto de ternas RGB que constituyen los parámetros más importantes del MIL. Son los siguientes:

M_E emisividad del material.

A_G término ambiente global.

r_A, r_D, r_S reflectividad difusa, ambiente y especular del material.

e exponente de la componente especular.

S_{iA}, S_{iD}, S_{iS} luminosidad de cada fuente de luz (para las componentes ambiental, difusa o especular).

$\mathbf{q}_i, \mathbf{l}_i$ posición o dirección de cada fuente de luz.

estos parámetros se pueden modificar en cualquier momento, afectando su nuevo valor a las evaluaciones del MIL posteriores.

Activación de iluminación

La librería OpenGL como parte de la funcionalidad fija (pre-programada), incluye una implementación de este modelo de iluminación.

- Es necesario usar la orden **glEnable/glDisable** para activar o desactivar la funcionalidad de iluminación:

```
glEnable(GL_LIGHTING); // activa evaluacion del MIL  
glDisable(GL_LIGHTING); // desactiva evaluacion del MIL
```

- Esta funcionalidad está por defecto desactivada en el estado inicial de OpenGL.

Nota toda esta funcionalidad solo está disponible hasta la versión 2.7 (OpenGL Legacy mode), a partir de la versión 3.0 (OpenGL Core Profile) es necesario programar el cálculo de iluminación usando shaders).

Activación y desactivación de las luces

Las implementaciones de OpenGL están obligadas a gestionar un número mínimo de 8 fuentes de luz.

- Cada una de ellas se referencia por un valor entero, el valor de las constantes **GL_LIGHT0**, **GL_LIGHT1**, ..., **GL_LIGHT8** (tienen valores consecutivos).
- Cada una de estas fuentes de luz puede activarse y desactivarse de forma individual, con:

```
glEnable(GL_LIGHTi) ; // enciende la i-esima fuente de luz  
glDisable(GL_LIGHTi) ; // apaga la i-esima fuente de luz
```

- Solo las fuentes activas intervienen en el MIL (por defecto están todas apagadas)

Posición de las fuentes

La posición o dirección de una fuente se puede especificar en varios sistemas de coordenadas, en función de cuando se haga la llamada a

```
GLfloat pos[4] = { 20.0, 10.0, 20.0, 0.0 };  
  
glLightfv(GL_LIGHTi, GL_POSITION, pos);
```

Las luces están integradas en el grafo de escena. Le afectan las transformaciones geométricas activas cuando se ubican en la escena usando la función **glLightf**.

Si la coordenada homogénea (w) es cero, la fuente está en el infinito, y la luz incide en la dirección del vector (x, y, z) .

Ejemplo: Posición en coordenadas polares

Por ejemplo, para establecer la dirección de una fuente de luz usando coordenadas polares (dos ángulo α y β de longitud y latitud, respectivamente, en grados), podríamos hacer:

```
GLfloat pos[4] = { 20.0, 10.0, 20.0, 0.0 };

glPushMatrix() ;

glRotatef( α, 0.0, 1.0, 0.0 ); // rotación α grados en torno a eje Y

glRotatef( β, 1.0, 0.0, 0.0 ); // rotación β grados en torno al eje X

glLightf(GL_LIGHTi, GL_POSITION, pos[0]);

glPopMatrix() ;
```

Si la luz se introduce después que la transformación de cámara su posición estará en coordenadas del modelo, si se introduce antes estará dada en coordenadas de cámara.

Configuración de las fuentes de luz

Se hace con la función **glLightf** (en todos los casos, el primer parámetro identifica la fuente de luz cuyos atributos queremos modificar)

Los valores de S_{iA}, S_{iD}, S_{iS} se fijan con estas llamadas

```
float color[4] = { 0.8, 0.0, 1, 1 };

glLightfv(GL_LIGHTi, GL_AMBIENT, color); // asigna  $S_{iA} := color$ 
glLightfv(GL_LIGHTi, GL_DIFFUSE, color); // hace  $S_{iD} := color$ 
glLightfv(GL_LIGHTi, GL_SPECULAR, color); // hace  $S_{iS} := color$ 
```

Las intensidades se especifican en (r, g, b) con componentes normalizadas entre 0 y 1.

Atributos de material

El estado interno de OpenGL contiene dos juegos de reflectividades del material: uno para polígonos **delanteros** (*front-facing polygons*), y otro para polígonos **traseros** (*back-facing polygons*).

- Por defecto, se consideran polígonos delanteros aquellos en cuya proyección los vértices aparecen en sentido anti-horario al recorrerlos en el orden en el que se proporcionan a OpenGL con **glVertex**. El resto son traseras (este comportamiento es configurable).
- Todas las llamadas que permiten cambiar el material tienen un primer parámetro que permite discriminar sobre que juego de ternas RGB se está actuando. Los valores son:

```
GL_FRONT           // atrib. del material de caras delanteras  
GL_BACK           // atrib. del material de caras traseras  
GL_FRONT_AND_BACK // ambos juegos de atributos
```

Ejemplo de material delantero/trasero

Aquí vemos un ejemplo de diferencias entre los atributos del material para las caras traseras y las delanteras, en un objeto no cerrado:

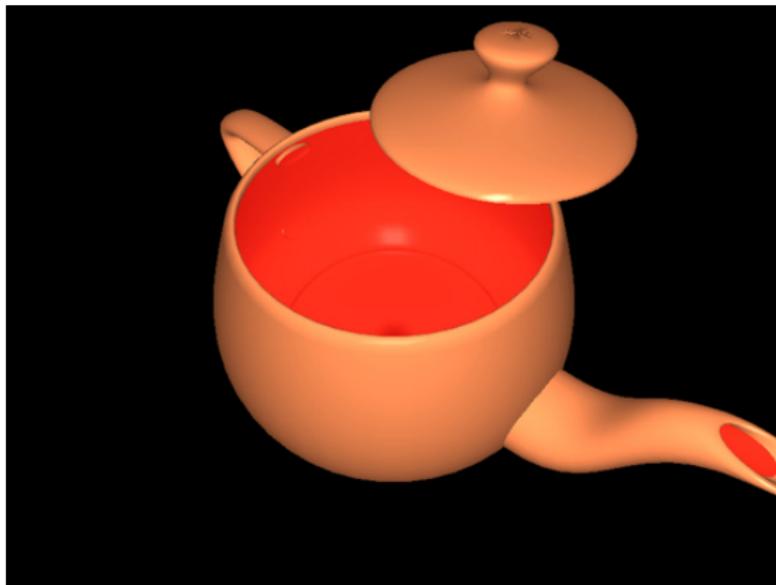


Imagen obtenida de: editorial Packt

Materiales

Las propiedades del material también forman parte del estado y se modifican con llamadas a la función **glMaterial**. Las reflectividades (r_A, r_D, r_S) y el exponente de brillo e se cambian con **glMaterial**.

Las reflectividades se indican en $rgb\alpha$ con valores normalizados. Un valor de α de 1 indica que el objeto es poco. Valores menores de 1 especifican objetos semitransparentes.

```
GLfloat color[4] = { r, g, b, 1.0 } ;  
  
// hace  $r_A := (r,g,b)$ , inicialmente  $(0.2, 0.2, 0.2)$   
glMaterialfv( GL_FRONT, GL_AMBIENT, color ) ;  
  
// hace  $r_D := (r,g,b)$ , inicialmente  $(0.8, 0.8, 0.8)$   
glMaterialfv( GL_FRONT, GL_DIFFUSE, color ) ;  
  
// hace  $r_S := (r,g,b)$ , inicialmente  $(0, 0, 0)$   
glMaterialfv( GL_FRONT, GL_SPECULAR, color ) ;  
  
// Asigna el exponente especular, inicialmente 0.0 (entre 0.0 y 128.0)  
glMaterialf( GL_FRONT, GL_SHININESS, e ) ;
```

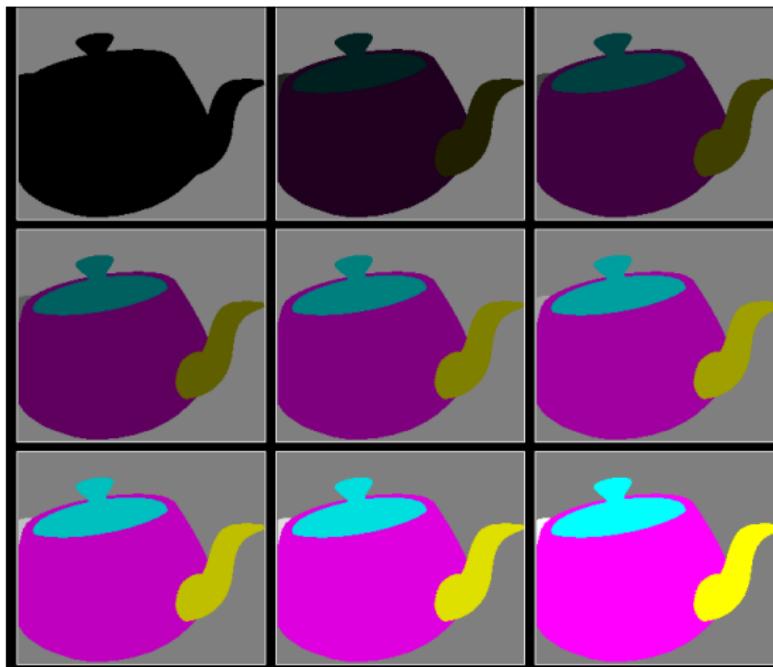
Emisividad

Tambien modificamos con la función **glMaterial** la emisividad del material(r_E):

```
GLfloat color[4] = { r, g, b, 1.0 } ;  
  
// hace  $r_E := (r,g,b)$ , inicialmente es (0,0,0)  
glMaterialf( GL_FRONT, GL_EMISSION, color[0] ) ;
```

Reflectividad ambiental

En este caso, la reflectividad ambiental $r_A(\mathbf{p})$ depende de en que parte de la tetera este el punto \mathbf{p} :



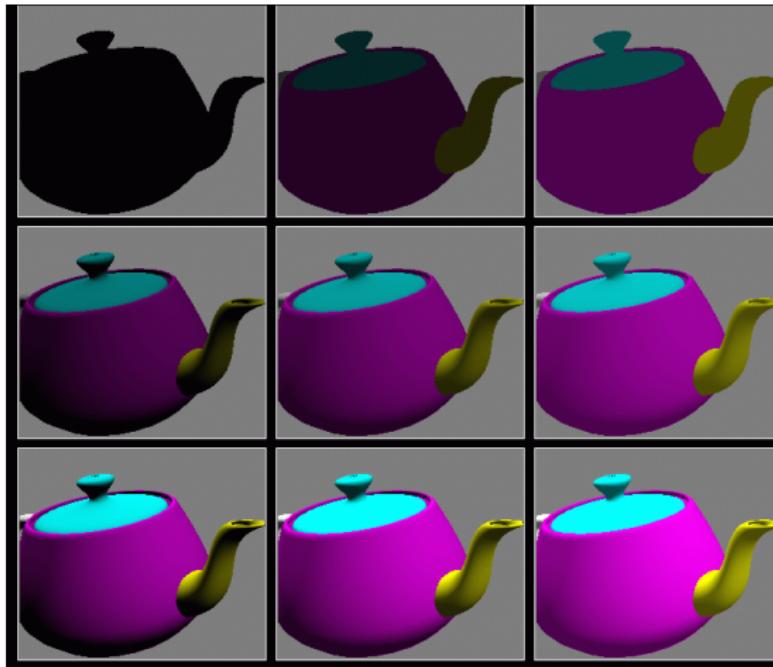
Material difuso

Ejemplo con dos fuentes de luz direccionales, $r_A(\mathbf{p}) = r_S(\mathbf{p}) = (0,0,0)$ (solo hay componente difusa)



Material difuso+ambiental

Aquí r_A crece de izquierda a derecha, y r_D de arriba abajo, $r_S = 0$:



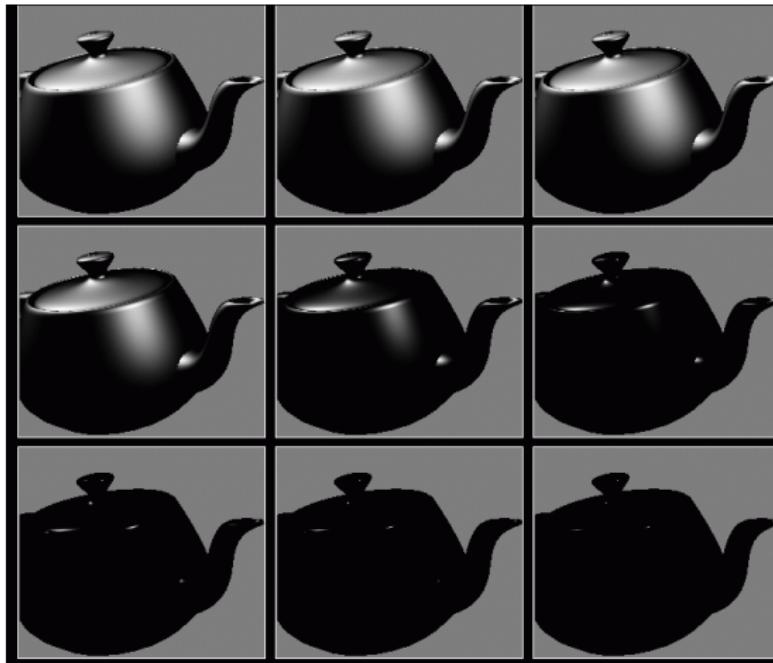
Ejemplo de material especular

Aquí $r_A(\mathbf{p}) = r_D(\mathbf{p}) = 0$, y $e = 5.0$:



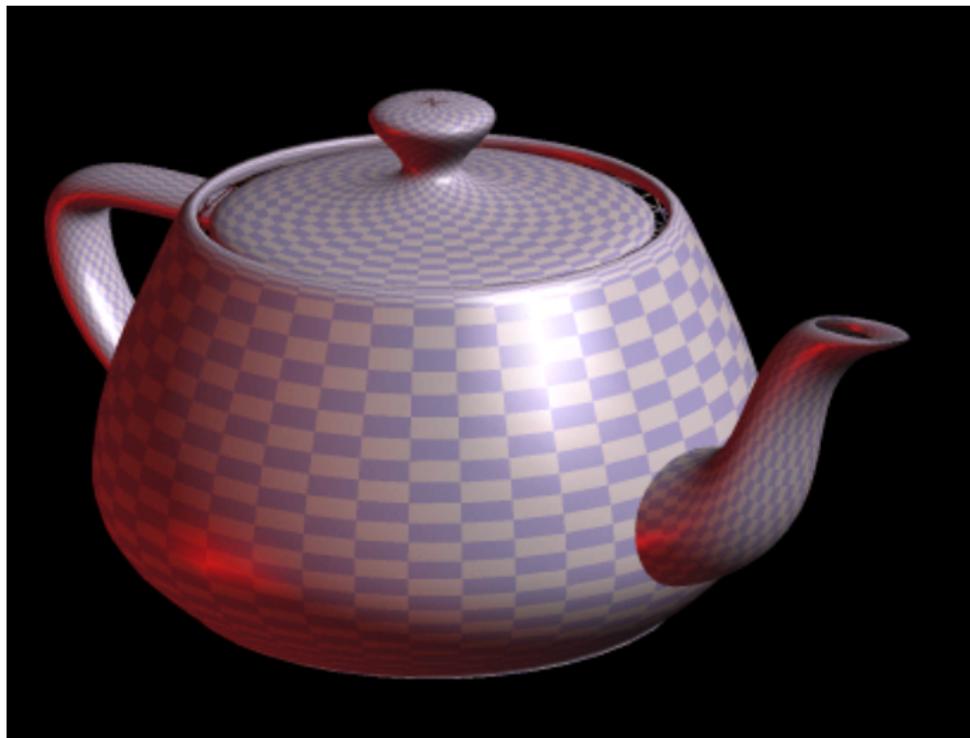
Efecto del exponente de brillo

Aquí crece de izquierda a derecha y de arriba abajo:



Ejemplo de material combinado

Combinación ambiental, más difusa, más pseudo.especular:



Tipos de sombreado en OpenGL

La función **glShadeModel** permite seleccionar el método de sombreado activo en cada momento (afecta a todas las primitivas posteriores)

- Sombreado plano se usa el color del último vértice para todo el polígono:

```
glShadeModel(GL_FLAT); // activa sombreado plano
```

- Sombreado de vértices el color de cada vértice es interpolado en el interior de los polígonos:

```
glShadeModel(GL_SMOOTH); // activa sombreado de vértices
```

- Sombreado de píxeles: no está disponible en OpenGL si no se hace programación del cauce gráfico.

inicialmente, el método de sombreado es el sombreado de vértices (**GL_SMOOTH**).

Sombreado e iluminación

El método de sombreado en OpenGL puede cambiarse incluso si la iluminación está desactivada. En cualquier caso (con ilum. activada o desactivada), OpenGL siempre asocia un color a cada vértice:

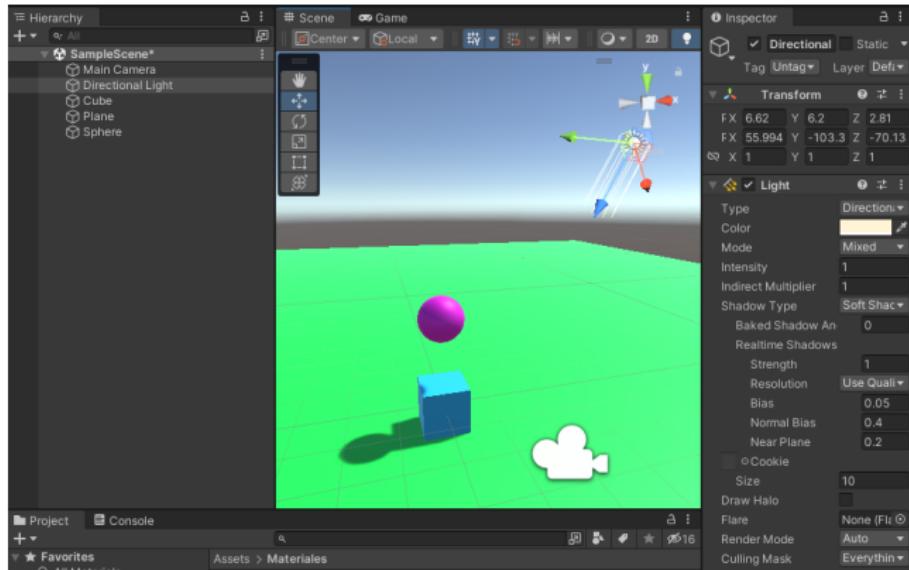
- Sin iluminación activada el color asociado a cada vértice es la terna C de su estado interno en el momento de llamar a **glVertex** (C se cambia con **glColor**)
- Con iluminación activada el color asociado a cada vértice es el resultado de evaluar el modelo de iluminación local en dicho vértice.

nota: la función **glShadeModel** fue declarada *obsoleta (deprecated)* en OpenGL 3.0 y eliminada de OpenGL 3.1 y posteriores, al igual que lo relacionado con iluminación.

Illuminación y materiales en Unity

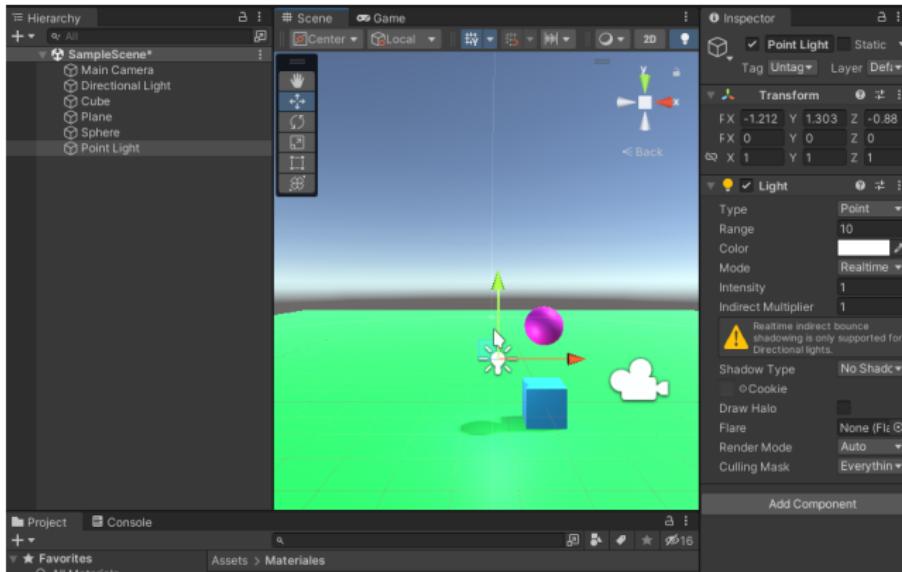
Luces en Unity: Luz direccional

Añadir como *GameObject > Light*. Fijar parámetros en *Inspector* (Type: *Directional*).



Luces en Unity: Luz puntual

Añadir como *GameObject > Light*. Fijar parámetros en *Inspector* (Type: Point).



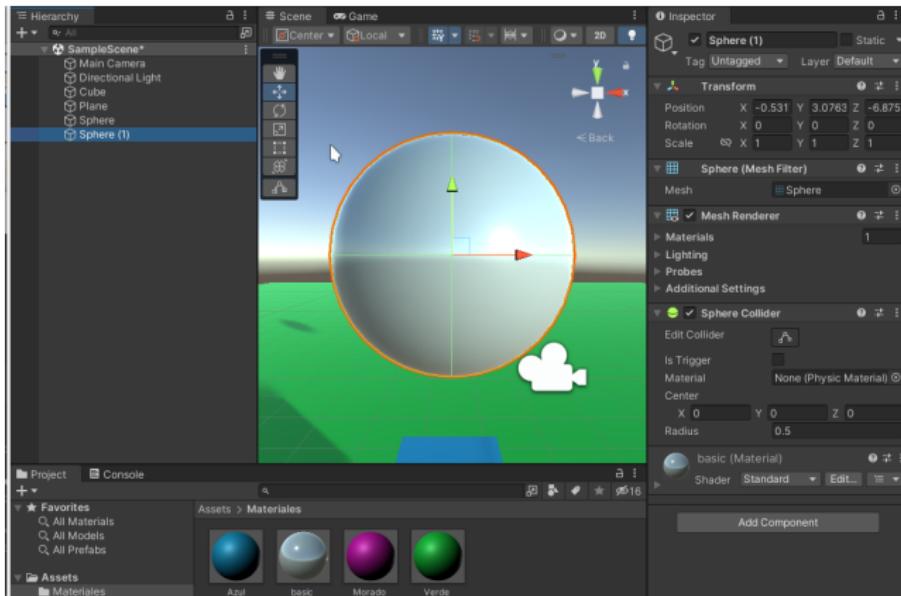
Materiales en Unity

Los materiales se crean como *Assets*. Los parámetros se editan en *Inspector*.



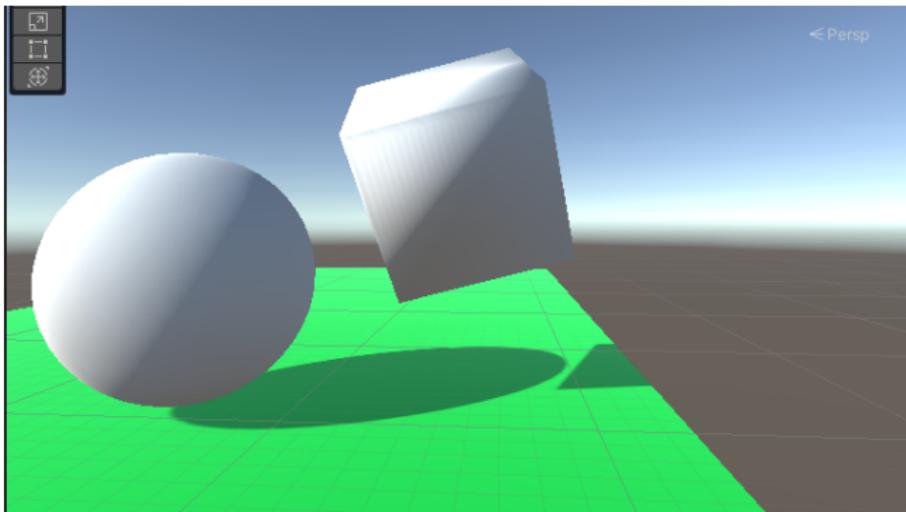
Materiales en Unity

Los materiales se asignan arrastrando su assests sobre el objeto. Aparecerán en el *Inspector* del objeto.



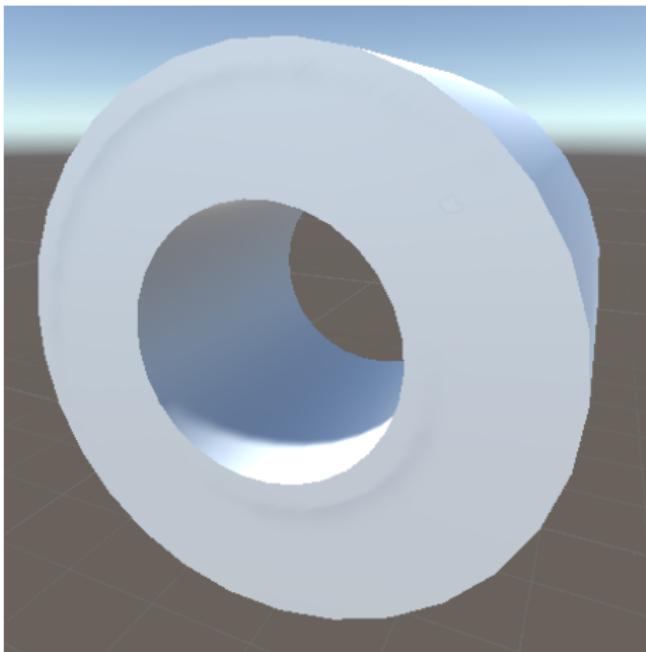
Sombreado en Unity

Unity no tiene un mecanismo para commutar entre cálculo de iluminación en vértices y en caras. Se utilizan normales de vértice. Para conseguir aristas vivas es necesario duplicar los vértices.



Sombreado en Unity

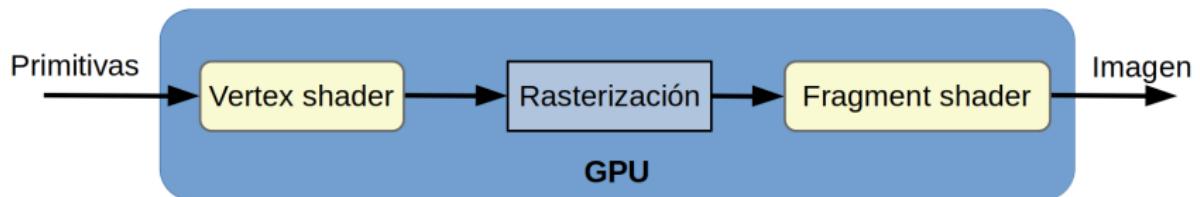
Se utilizan normales de vértice. Para conseguir aristas vivas es necesario duplicar los vértices.



Shaders

Shaders

Los sistemas gráficos modernos permiten personalizar el cauce gráfico, especialmente la transformación que se realiza con los vértices antes de rasterizar y las operaciones con los fragmentos de la rasterización.



La GPU puede cargar "shaders" que son programas un *vertexshader* y un *fragmentshader*.

Shaders: GLSL

GLSL es el acrónimo de *GL Shading Language*. Un shading language es un lenguaje de programación del renderizado. Hay lenguajes para renderizado off-line como RenderMan, y para renderizado en tiempo real como:

- GLSL
- CG (Nvidia)
- HLSL (Windows DirectX)

Vertex Shaders

El vertex shader procesa un único vértice. En el vertex shader se puede programar:

- La transformación de los vértices.
- La transformación de la normal.
- La generación y transformación de coordenadas de textura.
- Cálculo de iluminación por vértice
- Cálculo de color por vértice.

Entrada

- Estado de OpenGL
- Vértice
- Variables

Salida

- `gl_Position`
- Variables interpoladas (`varying`)

Rasterización

Esta fase no es programable. En ella la GPU determina los pixels en los que se debe dibujar cada primitiva, eliminando los innecesarios (*culling*). Para cada pixel sobre el que está la primitiva genera un fragmento.

En esta fase el hardware realiza las siguiente operaciones:

- Clipping, eliminando los que quedan fuera del campo de visión.
- Face culling, dibujando solo la cara visible de cada triángulo (front, back, front and back).
- Calcula la profundidad del fragmento.
- Interpola el valor de las variables de vértices para el fragmento.

Fragment Shaders

Para cada fragmento se ejecuta el fragment shader. Es decir el fragment shader procesa un único fragmento, sin acceso a los vecinos. En el fragment shader se puede programar:

- Calculo de color y coordenadas de textura por pixel.
- Aplicación de textura por pixel.
- Calculo de normales por pixel.
- Iluminación por pixel.

Entrada

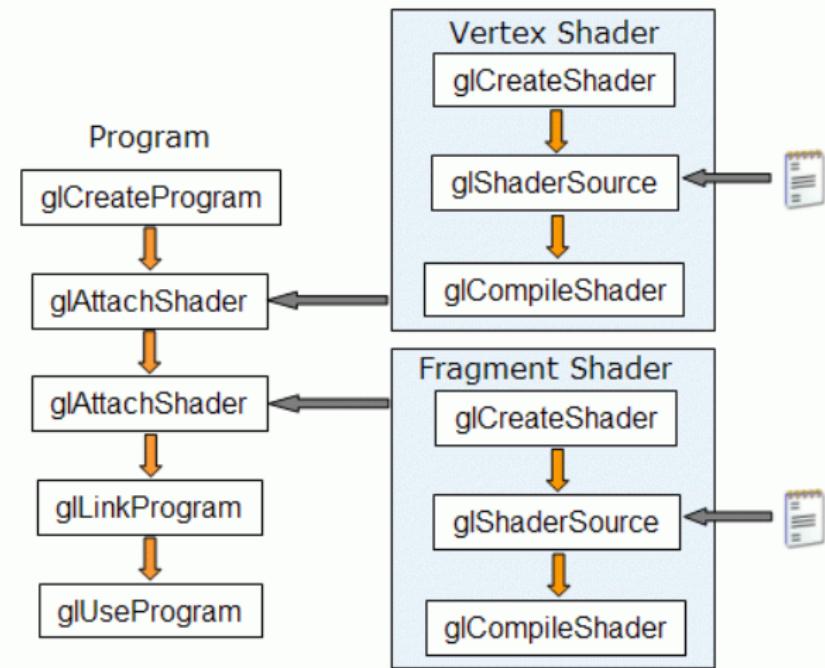
- Estado de OpenGL
- Texturas
- Variables interpoladas (varying)

Salida

- `gl_FragColor`

Shaders en OpenGL

Los shaders se cargan y compilan en tiempo de ejecución.



Shaders en OpenGL (con la librería GLEW)

Código para activar un shader

```
createShaders(&v1,&f1, "vertexShader.vert", "fragmentShader.frag");
p1 = glCreateProgram();
glAttachShader(p1,v1);
glAttachShader(p1,f1);
glLinkProgram(p1);
printProgramInfoLog(p1);
glUseProgram(p1);
```

Shaders en OpenGL (con la librería GLEW)

```
void createShaders(GLuint *vertex, GLuint *fragment, char * vertexFile,
char * fragmentFile) {
    char *vs = NULL,*fs = NULL;

    (*vertex) = glCreateShader(GL_VERTEX_SHADER);
    (*fragment) = glCreateShader(GL_FRAGMENT_SHADER);

    vs = textFileRead(vertexFile);
    fs = textFileRead(fragmentFile);

    const char * vv = vs;
    const char * ff = fs;

    glShaderSource((*vertex), 1, &vv,NULL);
    glShaderSource((*fragment), 1, &ff,NULL);

    free(vs);free(fs);

    glCompileShader((*vertex));
    glCompileShader((*fragment));

    printShaderInfoLog( (*vertex) );
    printShaderInfoLog( (*fragment) );
}
```

Ejemplo de vertex shader

```
varying vec3 normal;
varying vec3 vertex_to_light_vector;

void main()
{
    // Transforming The Vertex
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

    // Transforming The Normal To ModelView-Space
    normal = gl_NormalMatrix * gl_Normal;
    normal = normalize(normal);

    // Transforming The Vertex Position To ModelView-Space
    vec4 vertex_in_modelview_space = gl_ModelViewMatrix * gl_Vertex;

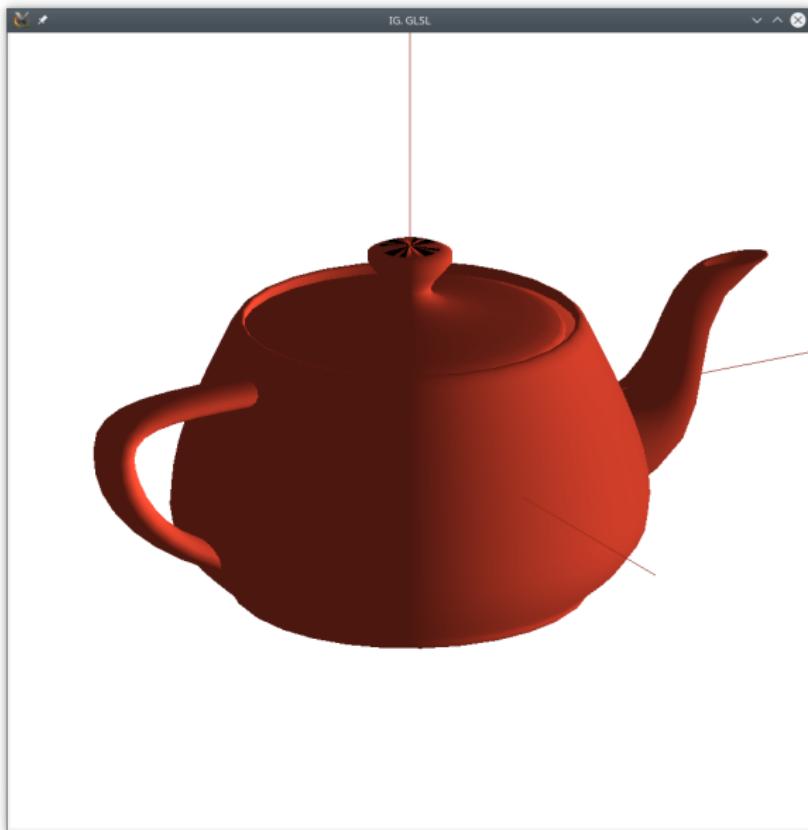
    // Calculating The Vector from The Vertex Position To The Light Position
    vertex_to_light_vector = vec3(gl_LightSource[0].position - vertex_in_mod
    vertex_to_light_vector = normalize(vertex_to_light_vector);
}
```

Ejemplo de fragment shader

```
varying vec3 normal;
varying vec3 vertex_to_light_vector;

void main()
{
    // Calculating The Diffuse Term And Clamping It To [0;1]
    float DiffuseTerm = normal*vertex_to_light_vector;
    if(DiffuseTerm<0.0) DiffuseTerm=0.0;
    // Calculating The Final Color
    gl_FragColor = gl_LightSource[0].ambient*gl_FrontMaterial.ambient +
    gl_LightSource[0].diffuse* DiffuseTerm*gl_FrontMaterial.diffuse;
}
```

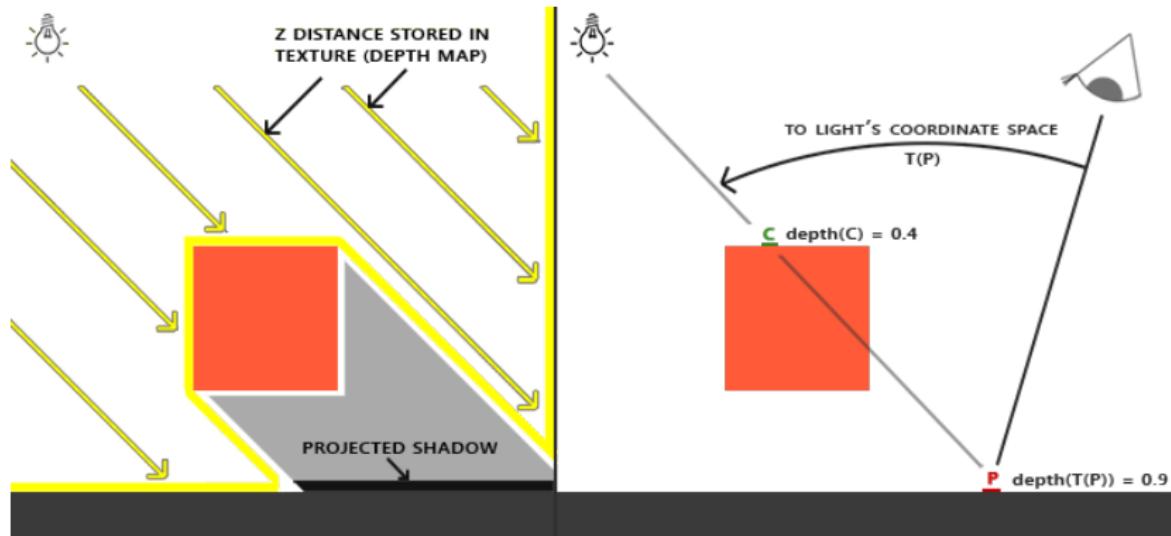
Shaders en OpenGL



Sombras y transparencias

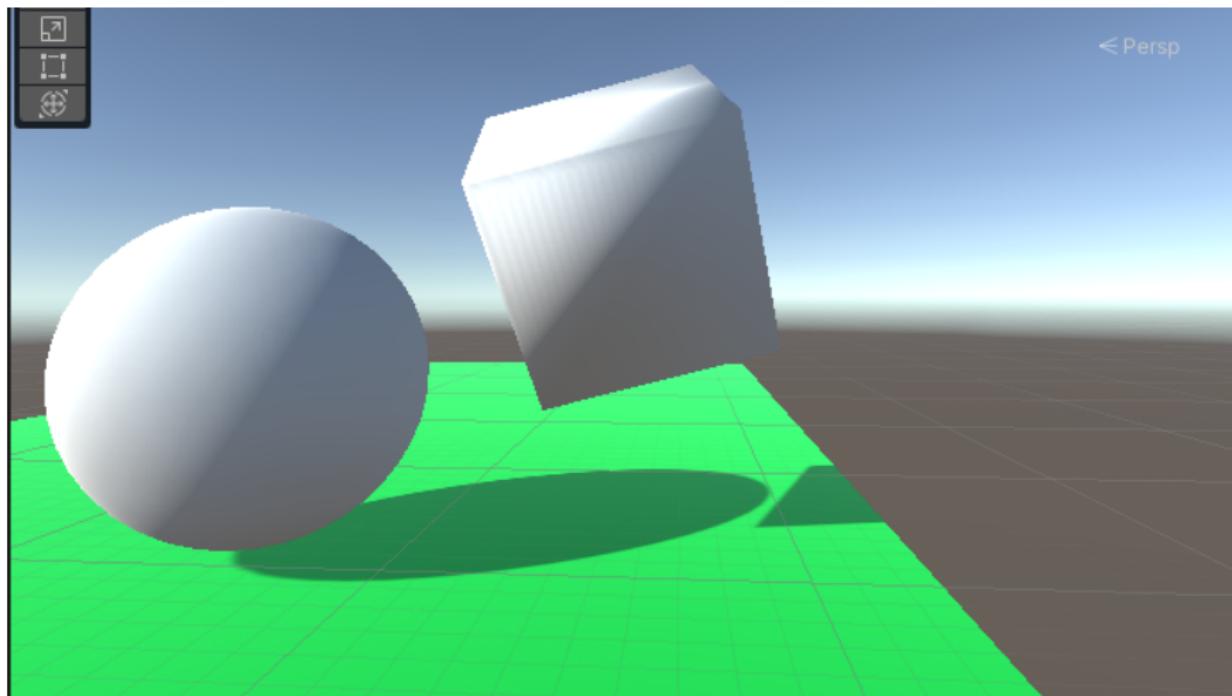
Sombras

Una técnica simple para dibujar sombras es "shadow mapping" [L. Williams 1978]. Se calcula el mapa de profundidad desde la fuente de luz, y se comprueba al dibujar cada fragmento su distancia a la fuente de luz.



Sombras en Unity

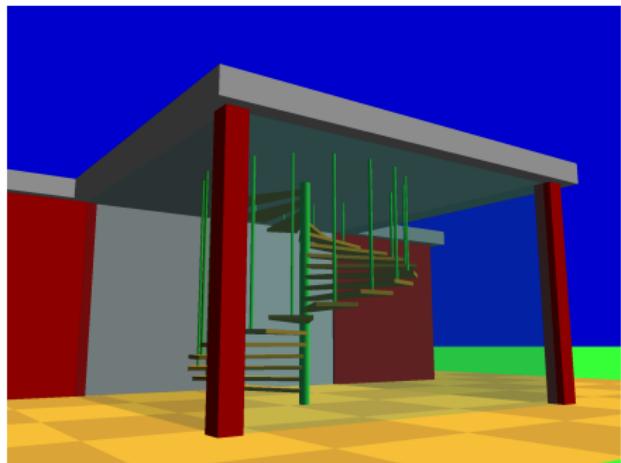
Unity calcula sombras usando shadow mapping.



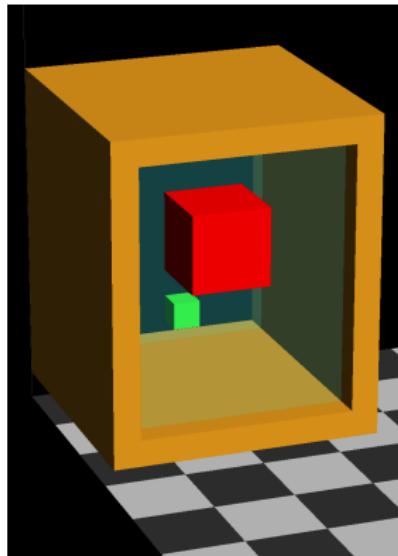
Transparencia

Para dibujar objetos transparentes se les asigna una opacidad (componente α del color) menor que 1, y:

- Dibujar los objetos transparentes después de los opacos.
- Que no se actualice el zbuffer al dibujar los transparentes.
- Que al dibujar objetos transparentes se combine su color con el de los ya dibujados.



Transparencia en OpenGL



```
...
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
...
void draw()
...
//Dibujar objetos opacos
...
glEnable(GL_BLEND);
glDepthMask(GL_FALSE);
...
// Dibujar objetos transparentes
...
glDisable(GL_BLEND);
glDepthMask(GL_TRUE);
...
```

$$color = Source_{\alpha}Source_{Color} + (1 - Source_{\alpha})Destination_{Color}$$

Source = Fragmento a dibujar

Destination = Contenido actual de Frame buffer

Transparencia en Unity

Cambiar rendering mode del material a *transparente* y asignar α en el Albedo.

