

Informática Gráfica: Teoría. Tema 3. Visualización.

Carlos Ureña

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

2018-19

Teoría. Tema 3. Visualización.

Índice.

- 1 Cauce gráfico y definición de la cámara.
- 2 Modelos de Iluminación
- 3 Un Modelo de Iluminación Local (MIL) básico.
- 4 Iluminación en OpenGL.
- 5 Métodos de sombreado para Z-buffer.
- 6 Visualización de texturas.
- 7 Texturas en OpenGL.
- 8 Materiales en el grafo de escena
- 9 Visualización con el cauce gráfico programable.

Sección 1

Cauce gráfico y definición de la cámara.

- 1.1. El cauce gráfico del algoritmo Z-buffer.
- 1.2. Transformación de vista
- 1.3. Transformación de proyección
- 1.4. Recortado y división por W
- 1.5. Transformación de viewport
- 1.6. Representación de parámetros de las transformaciones

Subsección 1.1

El cauce gráfico del algoritmo Z-buffer.

└ Cauce gráfico y definición de la cámara.

└ El cauce gráfico del algoritmo Z-buffer.

Introducción.

El término **cauce gráfico** (*graphics pipeline*) se suele usar para referirnos al conjunto de pasos de cálculo que se realizan para visualizar polígonos en el contexto del **algoritmo de Z-buffer**

- ▶ El algoritmo de Z-buffer se usa para presentar polígonos incluyendo **eliminación de partes ocultas** (EPO) en 3D (es decir: lograr presentar únicamente las partes visibles de los polígonos que se dibujan).
- ▶ OpenGL, DirectX y otras librerías 3D usan Z-buffer.
- ▶ Estos pasos **se implementan en hardware** en las tarjetas gráficas modernas (GPUs: *Graphics Processing Units*).
- ▶ Estos pasos **no se aplican en otros algoritmos** de visualización y EPO en 3D, como por ejemplo en Ray-tracing.

Pasos del cauce gráfico.

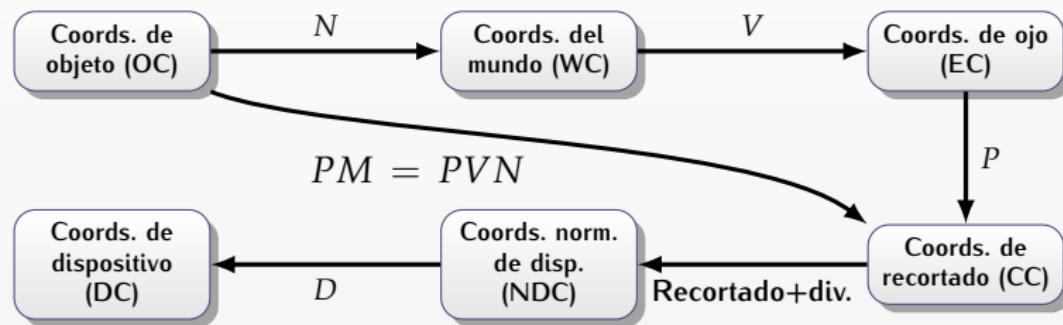
Los pasos del cauce gráfico suelen implementarse en secuencia, cada paso obtiene datos del anterior, los transforma de alguna manera y los entrega al siguiente paso. Los pasos (muy resumidos) son:

- 1 **Transformación** de coordenadas de vértices: cálculo de donde se proyecta en pantalla cada vértice.
- 2 **Recortado**: eliminación de partes de polígonos fuera de la zona visible.
- 3 **Rasterización y EPO**: cálculo de los píxeles donde se proyecta un polígono.
- 4 **Iluminación y texturización**: cálculo del color de cada pixel donde se proyecta un polígono.

la transformación y el recortado se pueden mezclar de diversas formas, ambos pasos son necesariamente previos a los otros dos, que también se pueden combinar de varias formas entre ellos

Esquema de la transformación y recortado

En estas etapas del cauce gráfico, esencialmente los datos que se transforman son coordenadas de vértices y conectividad entre ellos. El esquema es el siguiente:



Este esquema corresponde al recortado en CC (hay otras posibilidades, esta es la mejor).

Sistemas de coordenadas

El cauce gráfico de OpenGL contempla los siguientes:

- ▶ **Coordenadas de objeto o maestras:** son distancias relativas a un sistema de referencia específico o distinto de cada objeto, que se crea en este espacio.
- ▶ **Coordenadas del mundo:** son distancias relativas a un sistema de referencia común para todos los objetos de una escena
- ▶ **Coordenadas de cámara:** son distancias relativas a un sistema de referencia posicionado y alineado con la cámara virtual en uso.
- ▶ **Coordenadas de recortado:** son distancias normalizadas, con $w \neq 1$, relativas a un sistema asociado al rectángulo que forma la imagen en pantalla.
- ▶ **Coordenadas normalizadas de dispositivo (NDC):** similares a CC, pero con $w = 1$, y dentro de la zona visible.
- ▶ **Coordenadas de dispositivo:** similares a NDC, pero en unidades de pixels.

Matrices de transformación

Las matrices de transformación (4×4) involucradas permiten convertir coordenadas en un sistema de coordenadas a coordenadas en otro:

- ▶ **La matriz de modelado y vista (modelview) M** , compuesta de:
 - ▶ **Matriz de modelado N** : convierte de OC a WC
 - ▶ **Matriz de vista V** : convierte de WC a EC
- ▶ **La matriz de proyección P** : convierte de EC a CC. (recibe coordenadas con $w = 1$, pero produce coordenadas en general con $w \neq 1$)
- ▶ **La matriz del viewport D** : convierte de NDC a DC (depende de la resolución de la imagen en pantalla y de la zona de esta donde se visualiza).

Las coordenadas de dispositivo (con $w = 1$ y en unidades de pixels) se usan como entrada para las siguientes etapas del cauce gráfico (rasterización, EPO, iluminación y texturas)

Subsección 1.2

Transformación de vista

Parámetros de la transformación de vista

La matriz de vista se define a partir de los siguientes parámetros (ver figura.)

\dot{o}_c = es el punto del espacio foco de la proyección, donde estaría situado el observador ficticio que contempla la escena (*projection reference point*, PRP)

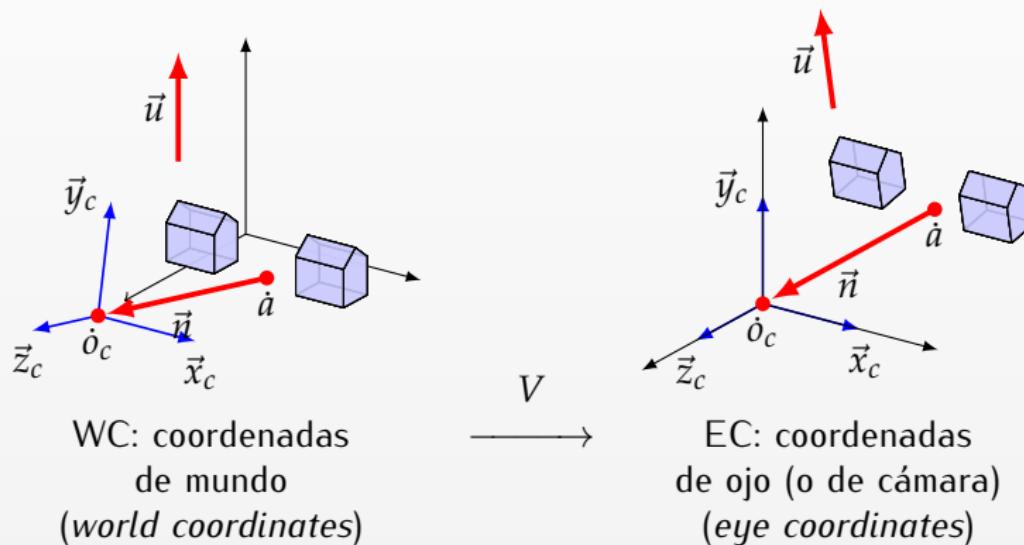
\vec{n} = vector libre perpendicular al plano ficticio donde se proyecta la imagen (paralelo al *eje óptico* de la cámara virtual). (*view plane normal*, VPN).

\vec{a} = punto en el eje óptico, también llamado *punto de atención* o *look-at*. Depende exclusivamente de \dot{o} y \vec{v} , por lo que sirve como alternativa para especificar la vista en lugar de \vec{n} (*view reference point*, VRP).

\vec{u} = es un vector libre que indica una dirección que el observador ve proyectada en vertical en la imagen (apuntando hacia arriba) (*view-up vector*, VUP)

Matriz de vista y efecto.

La **matriz de vista** V (*view matrix*) transforma las coordenadas de los vértices de manera que \vec{n} (VPN) queda alineado con el eje Z, el vector \vec{u} (VUP) en el plano YZ, y \vec{o}_c (PRP) en el origen:



Marco de referencia de la cámara.

A partir de \vec{n} (VPN) y \vec{u} (VUP) se pueden construir tres vectores perpendiculares de longitud unidad \vec{x}_c, \vec{y}_c y \vec{z}_c que, junto con \dot{o}_c , forman el **marco de referencia de la cámara o marco del observador**

$\mathcal{C} = [\vec{x}_c, \vec{y}_c, \vec{z}_c, \dot{o}_p]$:

$$\vec{z}_c = \frac{\vec{n}}{\|\vec{n}\|} \quad (\text{eje Z paralelo a VPN, normalizado})$$

$$\vec{x}_c = \frac{\vec{n} \times \vec{u}}{\|\vec{n} \times \vec{u}\|} \quad (\text{eje X perpendicular a VPN y VUP, normalizado})$$

$$\vec{y}_c = \vec{z}_c \times \vec{x}_c \quad (\text{eje Y perpendicular a los otros dos})$$

(si se especifica \dot{a} en lugar de \vec{n} , el vector \vec{n} se obtendría como $\vec{n} = \dot{o} - \dot{a}$).

Coordenadas del marco de la cámara \mathcal{C}

El marco de referencia de la cámara (o del observador) se suele representar en memoria usando las coordenadas del mundo de los vectores y el punto (coordenadas relativas a \mathcal{W}), es decir:

$$\vec{x}_c = \mathcal{W}(a_x, a_y, a_z, 0)^t = \mathcal{W}\mathbf{x}_c$$

$$\vec{y}_c = \mathcal{W}(b_x, b_y, b_z, 0)^t = \mathcal{W}\mathbf{y}_c$$

$$\vec{z}_c = \mathcal{W}(c_x, c_y, c_z, 0)^t = \mathcal{W}\mathbf{z}_c$$

$$\vec{o}_c = \mathcal{W}(o_x, o_y, o_z, 1)^t = \mathcal{W}\mathbf{o}_c$$

Estas coordenadas se obtienen a partir de las coordenadas en \mathcal{W} de los vectores \vec{u}, \vec{n} y el punto \vec{o} como hemos visto. Esas coordenadas son **u**, **n** y **o**, respectivamente. La matriz V se puede construir directamente a partir de ellas.

Cálculo de la matriz de vista

Supongamos un punto \dot{p} del cual conocemos sus coordenadas del mundo, es decir $\dot{p} = \mathcal{W}\mathbf{p}_w$, donde $\mathbf{p}_w = (x_w, y_w, z_w, 1)^t$. Queremos calcular sus coordenadas $\mathbf{p}_e = (x_e, y_e, z_e, 1)^t$ r, de forma que $\dot{p} = \mathcal{C}\mathbf{p}_e = \mathcal{W}\mathbf{p}_w$.

Por tanto la *matriz de vista* V es la matriz de cambio de base, desde el marco \mathcal{W} hacia el marco \mathcal{C} . Puesto que \mathcal{C} es cartesiano, V se obtiene fácilmente como $A \cdot D[-\mathbf{o}_c]$, donde A es la matriz (ortonormal) que tiene los vectores $\mathbf{x}_c, \mathbf{y}_c$ y \mathbf{z}_c en sus filas.

$$V = \begin{pmatrix} a_x & a_y & a_z & d_x \\ b_x & b_y & b_z & d_y \\ c_x & c_y & c_z & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = \overbrace{\begin{pmatrix} a_x & a_y & a_z & 0 \\ b_x & b_y & b_z & 0 \\ c_x & c_y & c_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}^A \overbrace{\begin{pmatrix} 1 & 0 & 0 & -o_x \\ 0 & 1 & 0 & -o_y \\ 0 & 0 & 1 & -o_z \\ 0 & 0 & 0 & 1 \end{pmatrix}}^{D[-\mathbf{o}_c]}$$

(donde $d_x = -\mathbf{x}_c \cdot \mathbf{o}_c$, $d_y = -\mathbf{y}_c \cdot \mathbf{o}_c$ y $d_z = -\mathbf{z}_c \cdot \mathbf{o}_c$).

Matriz de vista usando rotaciones

La matriz V también se puede escribir como una traslación seguida de una rotación de eje arbitrario, que a su vez se puede poner como tres rotaciones en torno a los ejes de \mathcal{W} , que alinean \mathcal{C} con \mathcal{W} :

$$V = R[\gamma, \mathbf{z}] \cdot R[\beta, \mathbf{y}] \cdot R[\alpha, \mathbf{x}] \cdot D[-\mathbf{o}_c]$$

a los ángulos α, β y γ se les llama **ángulos de Euler**.

Su cálculo se puede hacer a partir de las coordenadas de los elementos de \mathcal{C} , y la matriz que se construye es, lógicamente, la misma V que hemos visto antes.

Fijar la matriz de vista en OpenGL

Por tanto, para activar una cámara, el código OpenGL puede ser del estilo de este:

```
const GLfloat V[4][4] = // matriz  $V$  asociada al marco  $\mathcal{C}$ 
{{ ax, ay, az, dx }, // coords. de mundo de  $\mathbf{x}_c$ , y  $d_x = -\mathbf{o}_c \cdot \mathbf{x}_c$ 
 { bx, by, bz, dy }, // coords. de mundo de  $\mathbf{y}_c$ , y  $d_y = -\mathbf{o}_c \cdot \mathbf{y}_c$ 
 { cx, cy, cz, dz }, // coords. de mundo de  $\mathbf{z}_c$ , y  $d_z = -\mathbf{o}_c \cdot \mathbf{z}_c$ 
 { 0, 0, 0, 1 } } // origen de  $\mathcal{C}$ 
};

glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
glMultMatrixf( V );
```

Lógicamente, los valores del array V se deben calcular como hemos visto antes.

Fijando la matriz de vista en GLU/OpenGL

La función **gluLookAt** (de la librería GLU) permite componer una matriz de vista de forma más cómoda, ya que acepta directamente las componentes de **o**, **a** y **u** como parámetros. Está declarada como sigue:

```
void gluLookAt
( GLdouble ox, GLdouble oy, GLdouble oz,
  GLdouble ax, GLdouble ay, GLdouble az,
  GLdouble ux, GLdouble uy, GLdouble uz
);
```

donde:

$$\mathbf{o} = (o_x, o_y, o_z)$$

$$\mathbf{a} = (a_x, a_y, a_z)$$

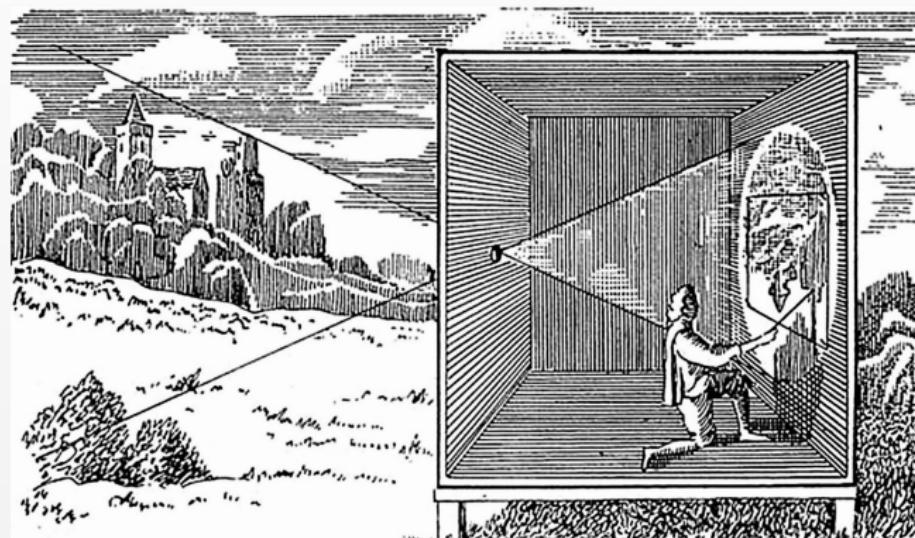
$$\mathbf{u} = (u_x, u_y, u_z)$$

Subsección 1.3

Transformación de proyección

Introducción

La **transformación de proyección** emula la proyección que ocurre idealmente en una *cámara oscura*, sobre la pared opuesta a la apertura. Es similar a lo que ocurre en una cámara de fotografía, al proyectarse la escena sobre el sensor.



El plano de visión. Tipos de proyección

Los vértices se *proyectan* sobre un plano alineado con el sistema de referencia de la cámara:

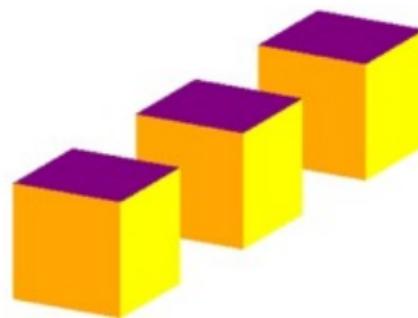
- Dicho plano se denomina **plano de visión (viewplane)**, es siempre perpendicular al eje Z del marco de vista.

La proyección puede ser de dos tipos

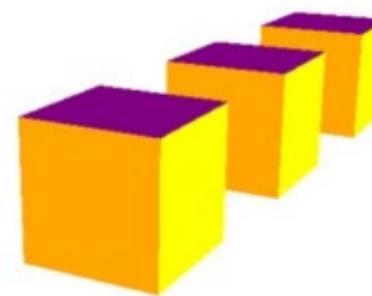
- **Proyección perspectiva:** los vértices se proyectan sobre el plano de visión usando líneas que van desde cada punto al origen del marco de coordenadas de la cámara (a esas líneas se les llama **projectores**, el origen actua como **foco** de la proyección). La coordenada Z del plano de visión debe ser estrictamente positiva.
- **Proyección ortográfica (o paralela):** los projectores son todos paralelos al eje Z. El plano de visión puede estar situado en cualquier valor de Z. Es un caso límite de la perspectiva, con el foco infinitamente alejado de la escena.

Comparación de proyecciones

Aunque ninguna de las dos formas de proyección es igual al comportamiento del sistema visual humano, la proyección perspectiva nos parece más natural (la ortográfica es poco realista):



Orthographic Projection

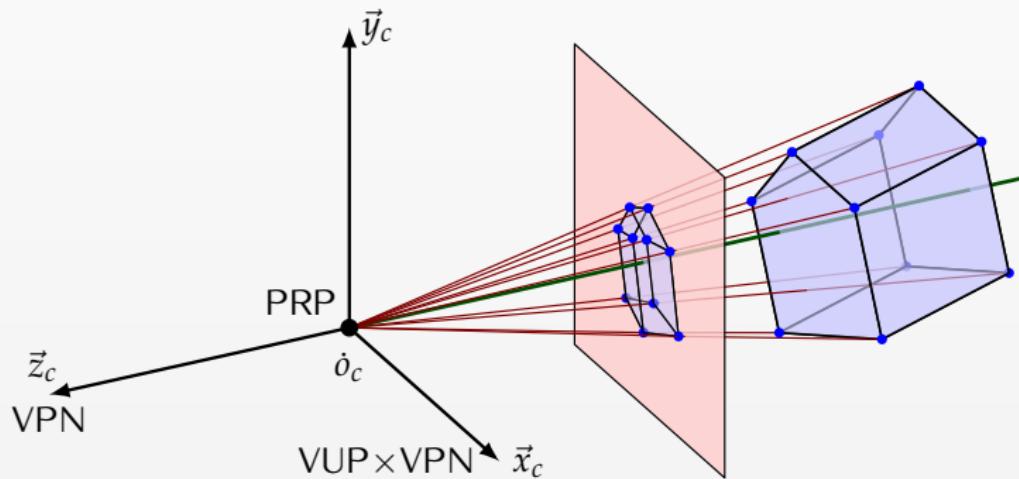


Perspective Projection

(a la izquierda, se interpreta que el cubo más lejano es más grande que los otros, aunque en la imagen son los tres del mismo tamaño)

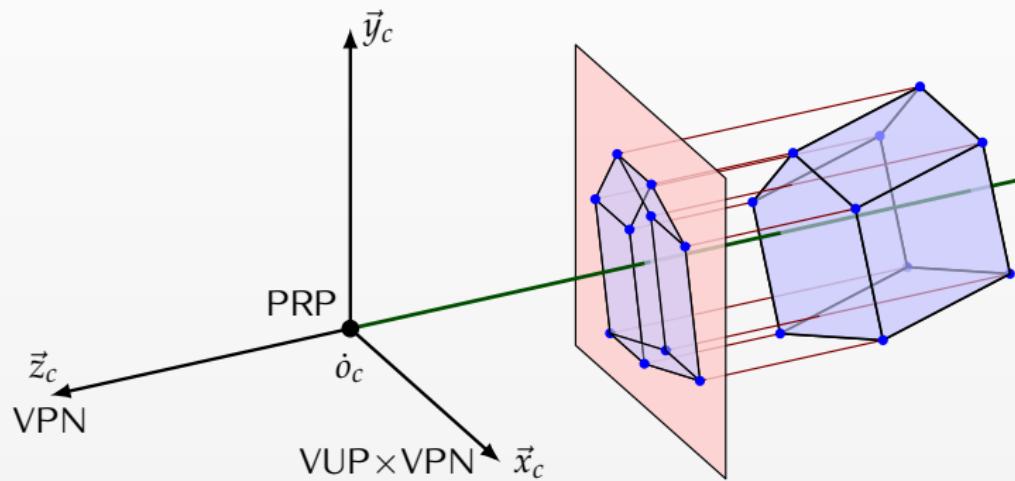
Proyección perspectiva

Cambia el tamaño de los objetos, usando un factor de escala s que crece de forma inv. proporcional a la distancia (d_z) en Z desde el objeto al foco (s es de la forma $1/(ad_z + b)$)



Proyección paralela

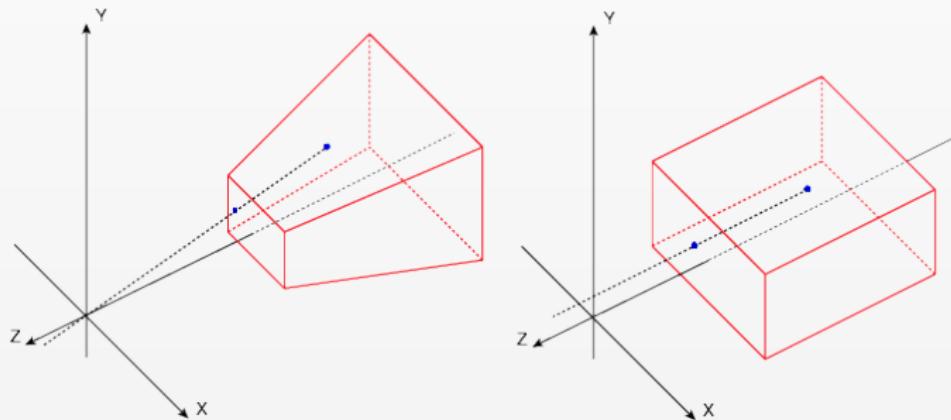
En este caso, no hay transformación de escala, y la proyección se puede ver como una transformación afín:



El *view-frustum*

El ***view-frustum*** designa la región del espacio de la escena que es visible en el viewport. Su forma depende del tipo de proyección:

- ▶ Perspectiva: es un tronco de pirámide rectangular (izq.).
- ▶ Ortográfica: es un paralelepípedo ortogonal u ortoedro (der.).



Transformación del view-frustum en un cubo

El view-frustum está determinado por los 6 planos que contienen a las 6 caras que lo delimitan.

- ▶ Estos planos se determinan por sus coordenadas en el marco de coordenadas de vista.
- ▶ La transformación de proyección transforma el view-frustum (en coordenadas de vista) en un cubo de lado 2 centrado en el origen, entre -1 y 1 en los tres ejes (en coordenadas de recortado, normalizadas).
- ▶ La proyección ortográfica es una transformación afín: una traslación seguida de escalado no necesariamente uniforme.
- ▶ La proyección perspectiva no es una transformación afín, aunque se puede expresar usando una transformación afín en coordenadas homogéneas 4D, seguida de una proyección de 4D a 3D.

Parámetros del view-frustum. Extension en Z

Los 6 valores l, r, t, b, n y f (los **parámetros** de frustum) se interpretan en el marco coordenadas de vista y determinan la transformacion desde las EC (coordenadas de vista) (x_e, y_e, z_e) de un punto (en el view-frustum) en sus coordenadas de recortado (CC) normalizadas (x_n, y_n, z_n) (entre -1 y 1).

- ▶ Los valores n (**near**) y f (**far**) son los límites en Z del view-frustum, pero cambiados de signo (se cumple $n \neq f$).
 - ▶ El plano $z_e = -n$ en EC se transforma en el plano $z_n = -1$ en CC.
 - ▶ El plano $z_e = -f$ en EC se transforma en el plano $z_n = +1$ en CC.
- ▶ En la proyección perspectiva, se exige además $0 < n < f$.
- ▶ Aunque no se exige así, lo usual es que seleccione $n < f$, es decir, el view-frustum se extiende en Z en el intervalo $[-f, -n]$. En adelante supondremos $n < f$, de forma que:
 - ▶ El plano $z_e = -n$ se llama **plano de recorte delantero**
 - ▶ El plano $z_e = -f$ se llama **plano de recorte trasero**

Parámetros del view-frustum. Extensión en X e Y.

Respecto de los otros cuatro valores (l, r, b y t), determinan la extensión en X y en Y:

- ▶ l (**left**) y r (**right**) son los límites en X del view-frustum ($l \neq r$).
- ▶ b (**bottom**) y t (**top**) son los límites en Y ($b \neq t$).
- ▶ En proy. ortográfica:
 - ▶ El plano $x_e = l$ en EC se transforma en el plano $x_n = -1$ en CC.
 - ▶ El plano $x_e = r$ en EC se transforma en el plano $x_n = +1$ en CC.
 - ▶ El plano $y_e = b$ en EC se transforma en el plano $y_n = -1$ en CC.
 - ▶ El plano $y_e = t$ en EC se transforma en el plano $y_n = +1$ en CC.
- ▶ En proy. perspectiva:
 - ▶ El plano $-nx_e = lz_e$ (EC) se transf. en el plano $x_n = -1$ (CC).
 - ▶ El plano $-nx_e = rz_e$ (EC) se transf. en el plano $x_n = +1$ (CC).
 - ▶ El plano $-ny_e = bz_e$ (EC) se transf. en el plano $y_n = -1$ (CC).
 - ▶ El plano $-ny_e = tz_e$ (EC) se transf. en el plano $y_n = +1$ (CC).

Propiedades de la extensión en X e Y

Hay que tener en cuenta que:

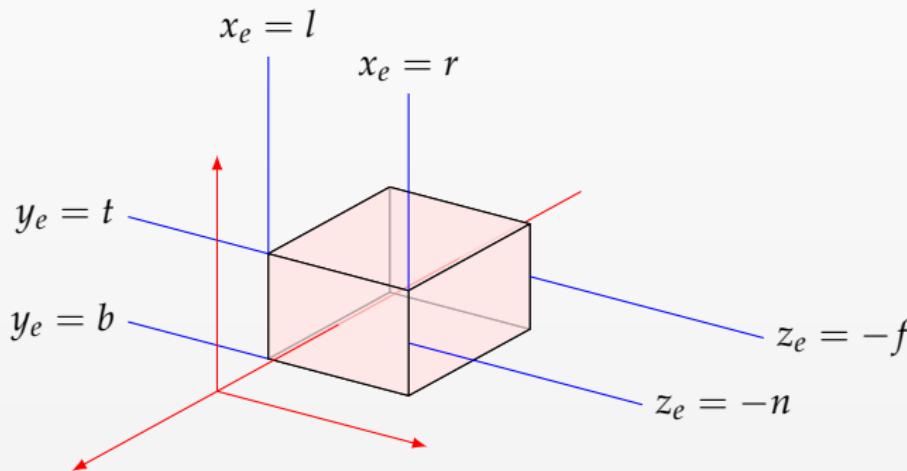
- ▶ Aunque esto no es requerido estrictamente, usualmente se seleccionan los parámetros de forma que $l < r$ y $b < t$.
- ▶ Cuando se cumple $l = -r$ y $b = -t$, decimos que el **view-frustum está centrado** (el eje Z pasa por el centro de las caras delantera y trasera). Esto es lo más usual, y se corresponde con lo que ocurre en una cámara.
- ▶ El valor $(r - l)/(t - b)$ suele coincidir con la relación de aspecto del viewport (ancho/alto, o bien núm.columnas/núm.filas). Si esto no ocurre los objetos aparecerán deformados en la imagen.

En adelante supondremos que siempre seleccionamos $l < r$ y $b < t$.

Parámetros en la proyección ortográfica.

En pr. ortográfica el view-frustum es un **ortoedro**. Contiene los puntos cuyas coordenadas de cámara (x_e, y_e, z_e) cumplen:

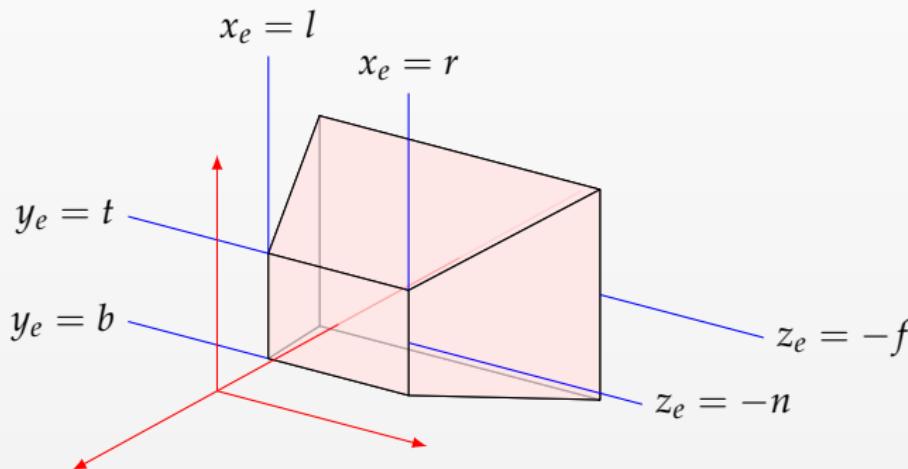
$$l \leq x_e \leq r \quad b \leq y_e \leq t \quad -f \leq z_e \leq -n$$



Parámetros en la proyección perspectiva.

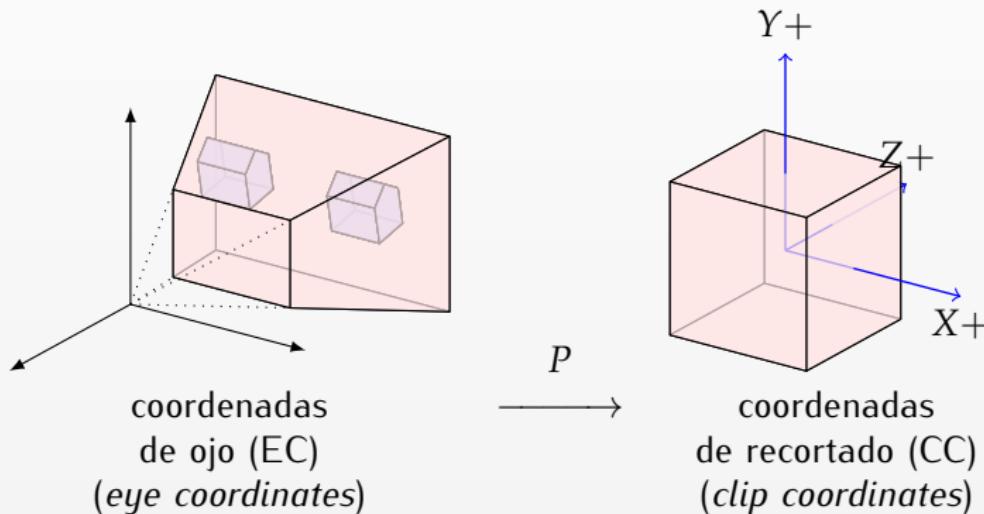
En perspectiva, el view-frustum es una **pirámide rectangular truncada**. Contiene los puntos cuyas coordenadas de cámara (x_e, y_e, z_e) cumplen:

$$l \leq x_e \left(\frac{n}{-z_e} \right) \leq r \quad b \leq y_e \left(\frac{n}{-z_e} \right) \leq t \quad -f \leq z_e \leq -n$$



Efecto de la transformación de proyección

El efecto de la **transformación de proyección** (matriz P) es convertir la región visible del espacio en un cubo de lado 2 unidades y con centro en el origen:



Proyección perspectiva sobre el plano delantero

Podemos suponer que los puntos se proyectan sobre el plano frontal del view-frustum (coord. Z igual a $-n$) con foco en \mathbf{o}_c .

- ▶ Dado un punto $\mathbf{p} = \mathcal{C}(x_c, y_c, z_c, w_c)$ queremos calcular las coordenadas de su proyección (x', y', z', w') (en principio, con $w' = w_c = 1$).
- ▶ Si se asume $z_c < 0$ (el punto está en la recta negativa del eje Z), podemos hacer:

$$x' = \frac{x_c n}{-z_c} \quad y' = \frac{y_c n}{-z_c} \quad z' = \frac{z_c n}{-z_c} = -n$$

esta transformación tiene dos problemas:

- ▶ Las coordenadas resultado no están normalizadas (entre -1 y 1).
- ▶ Colapsa o *aplana* todas las coordenadas Z (son todas $-n$).

Normalización de coordenadas X e Y

Si el punto original está en el view-frustum, entonces:

- ▶ x' está en el intervalo $[l, r]$
- ▶ y' está en el intervalo $[b, t]$.
- ▶ queremos dejar ambas coordenadas en el intervalo $[-1, 1]$
- ▶ podemos usar un escalado y traslación adicionales en X e Y:

$$x'' = 2 \left(\frac{x' - l}{r - l} \right) - 1 = \frac{x_c a_0}{-z_c} - a_2 = \frac{x_c a_0 + z_c a_2}{-z_c}$$

$$y'' = 2 \left(\frac{y' - b}{t - b} \right) - 1 = \frac{y_c b_1}{-z_c} - b_2 = \frac{y_c b_1 + z_c b_2}{-z_c}$$

donde hemos usado estas cuatro constantes:

$$a_0 \equiv \frac{2n}{r - l} \quad a_2 \equiv \frac{r + l}{r - l} \quad b_1 \equiv \frac{2n}{t - b} \quad b_2 \equiv \frac{t + b}{t - b}$$

Información de profundidad y normalización en Z

El problema está de hacer $z' = -n$ está en que **se pierde información de profundidad en Z**, que es necesaria para EPO). Para evitarlo, se usa una función lineal de z con dos constantes c_2 y c_3 :

$$z'' = \frac{z_c c_2 + c_3}{-z_c} \quad \text{donde:} \quad c_2 \equiv \frac{n+f}{n-f} \quad c_3 \equiv \frac{2fn}{n-f}$$

- ▶ los dos valores c_2 y c_3 se eligen de forma que, para $z_c = -n$, se hace $z'' = -1$, y para $z_c = -f$, se hace $z'' = 1$.
- ▶ es decir: el rango $[-f, -n]$ se lleva al rango $[-1, 1]$ (invirtiendo el orden).
- ▶ esta transformación conserva el orden (invertido) de las coordenadas Z (*no aplana*)
- ▶ ahora, valores menores de Z implican más cercanos al observador, y valores mayores, más lejanos.

Coordenadas cartesianas del punto proyectado

En resumen, tenemos estas tres igualdades:

$$x'' = (x_c a_0 + z_c a_2) / (-z_c)$$

$$y'' = (x_c b_1 + z_c b_2) / (-z_c)$$

$$z'' = (z_c c_2 + c_3) / (-z_c) = (z_c c_2 + w_c c_3) / (-z_c)$$

esta transformación incluye una división, y por tanto

- ▶ **no se puede implementar con una matriz** como hacíamos con las anteriores (no es lineal)
- ▶ aunque sí transforma líneas rectas en líneas rectas

Coordenadas de recortado

Para solventar el problema anterior (para poder usar una matriz), se definen las **coordenadas de recortado**, a partir de las coordenadas de cámara del original:

$$x_r = x_c a_0 + z_c a_2$$

$$y_r = y_c b_1 + z_c b_2$$

$$z_r = z_c c_z + w_c c_3$$

$$w_r = -z_c$$

Esta transformación ya **sí se puede hacer con una matriz 4x4**:

- ▶ se ha eliminado la división por $-z_c$, el resto es igual
- ▶ esta división se hace más adelante en el cauce gráfico
- ▶ para ello, el denominador de la división ($-z_c$) queda guardado en w_r (**que ya no es 1**).

La matriz de proyección perspectiva Q

Con todo lo dicho, la proyección perspectiva se puede realizar usando una matriz Q , que se aplica a coordenadas de cámara (con $w_c = 1$) y produce coordenadas de recortado (con $w_r \neq 1$):

$$Q = \begin{pmatrix} a_0 & 0 & a_2 & 0 \\ 0 & b_1 & b_2 & 0 \\ 0 & 0 & c_2 & c_3 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

de forma que, a partir de unas coordenadas de cámara, podemos obtener las coordenadas de recortado:

$$(x_r, y_r, z_r, w_r)^t = Q(x_c, y_c, z_c, w_c)^t$$

La proyección ortográfica.

En el caso de la **proyección ortográfica** (*orthographic projection*), se hace proyección en una dirección paralela al eje Z:

- ▶ esta transformación **solo requiere la normalización de los rangos de valores en los tres ejes** (se usa traslación más escalado)
- ▶ (1) traslación T (lleva el centro del paralelepípedo al origen)

$$T \equiv D \left[-\frac{l+r}{2}, -\frac{t+b}{2}, -\frac{f+n}{2} \right]$$

- ▶ (2) escalado S (deja los valores en $[-1, 1]$ en los tres ejes):

$$S \equiv E \left[\frac{2}{r-l}, \frac{2}{t-b}, \frac{-2}{f-n} \right]$$

(en Z se cambia de signo para *invertir* el eje Z)

La matriz de proyección ortográfica

La matriz de proyección ortográfica O se obtiene por tanto como composición de T seguido de S , es decir $O = S \cdot T$, o lo que es lo mismo:

$$O = \begin{pmatrix} a'_0 & 0 & 0 & a'_3 \\ 0 & b'_1 & 0 & b'_3 \\ 0 & 0 & c'_2 & c'_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \text{ donde: } \begin{cases} a'_0 \equiv \frac{2}{r-l} & a'_3 \equiv \frac{r+l}{r-l} \\ b'_1 \equiv \frac{2}{t-b} & b'_3 \equiv \frac{t+b}{t-b} \\ c'_2 \equiv \frac{-2}{f-n} & c'_3 \equiv \frac{f+n}{f-n} \end{cases}$$

de forma que ahora hacemos:

$$(x_r, y_r, z_r, w_r)^t = O(x_c, y_c, z_c, w_c)^t$$

donde w_r sí vale 1 con seguridad.

Fijando la matriz de proyección en OpenGL

En el estado de OpenGL hay una matriz de proyección (*projection matrix*) que llamaremos P y que puede ser manipulada mediante varias llamadas:

La función **glMatrixMode** se puede usar para poner OpenGL en modo matriz de proyección de forma que posteriores operaciones se realicen sobre la matriz P . Se usaría una llamada como esta:

```
glMatrixMode(GL_PROJECTION) ;
```

Para inicializar P , podemos usar esta función seguida de **glLoadIdentity**:

```
glMatrixMode(GL_PROJECTION) ; // modo 'matriz proyeccion'  
glLoadIdentity() ; // hace  $P := \text{Id}$ 
```

Transf. de proyección en OpenGL (2)

Para componer la matriz Q con P se puede invocar a **glFrustum**, una función declarada como se indica aquí:

```
glFrustum( GLdouble l, GLdouble r,  
           GLdouble b, GLdouble t,  
           GLdouble n, GLdouble f ) ; // hace  $P := PQ$ 
```

Si queremos una proyección ortográfica, podemos usar **glOrtho** en lugar de **glFrustum**, con los mismos parámetros:

```
glOrtho( GLdouble l, GLdouble r,  
         GLdouble b, GLdouble t,  
         GLdouble n, GLdouble f ) ; // hace  $P := PO$ 
```

Transf. de proyección con GLU

En la librería GLU (funciona sobre OpenGL) se puede usar la llamada a la función **gluPerspective**, para componer una proyección perspectiva de forma más intuitiva:

```
gluPerspective( GLdouble  $\alpha$ , GLdouble  $a$ ,  
                 GLdouble  $n$ , GLdouble  $f$  ) ;
```

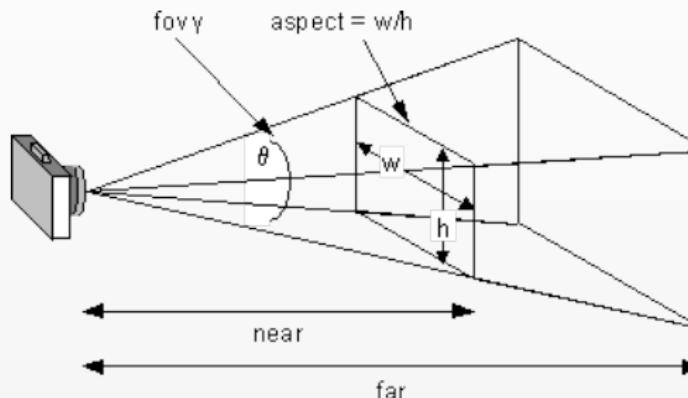
esta función equivale a un **glFrustum** con $r = -l$ y $t = -b$, en este caso:

$\alpha \equiv$ es la *apertura vertical del campo de visión (fovy)*, es el ángulo en grados (entre 0 y 180.0) que hay entre la cara superior y la inferior del frustum, equivale a $2\arctan(t/n)$

$a \equiv$ relación de aspecto de la imagen a producir (ancho dividido por alto), equivale a r/b .

Parámetros de gluPerspective

El significado de los parámetros se aprecia en esta figura:



esta perspectiva se dice que está *centrada*, ya que al ser $r = -l$ y $t = -b$, entonces el eje óptico de la cámara virtual (eje Z negativo) pasa por el centro del plano de visión (es lo más usual).

Subsección 1.4

Recortado y división por W

Recortado

Una vez se tienen las coordenadas de recortado de los vértices, se comprueban que primitivas estan dentro o fuera del viewfrustum (que en CC es un cubo de lado dos unidades centrado en el origen):

- ▶ Las primitivas completamente dentro de la zona visible se mantienen
- ▶ Las primitivas completamente fuera de la zona visible se descartan
- ▶ Las primitivas parcialmente dentro se dividen en partes: unas completamente dentro y otras completamente fuera (que se descartan). Esto causa la inserción de nuevas primitivas con algunos vértices nuevos justo en los planos que delimitan el view-frustum.

División por W . Coordenadas normalizadas de dispositivo.

Los vértices (en el view-frustum) resultado del recorte tienen coordenadas de recorte con $w_r \neq 0$ (si $P = Q$, entonces además $w_r \neq 1$). El siguiente paso es hacer la división por w_r de las tres componentes. Se obtienen las **coordenadas normalizadas de dispositivo**, con componente W de nuevo a 1:

$$(x_n, y_n, z_n, 1) = \frac{1}{w_r} (x_r, y_r, z_r, w_r)$$

los valores x_n, y_n y z_n están los tres en el intervalo $[-1, 1]$ (los vértices ya han pasado el recortado).

Subsección 1.5

Transformación de viewport

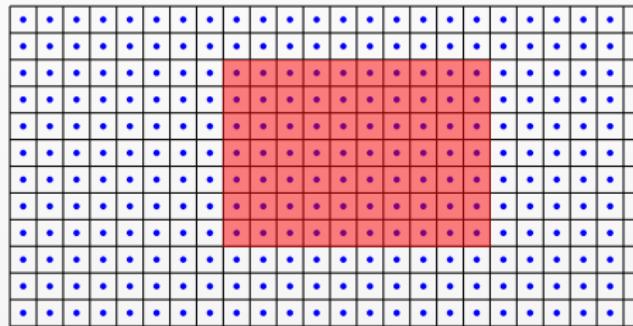
Transformación de Viewport

El siguiente paso consiste en calcular en qué posiciones de la imagen se proyecta cada vértice:

- ▶ Este paso se puede modelar como una transformación lineal que llamaremos **transformación de viewport**. El término **viewport** hace referencia a la zona rectangular de la ventana donde se proyectarán los polígonos que están en el cubo visible (un bloque rectangular de pixels)
- ▶ Esta transformación produce **coordenadas de dispositivo o de ventana** (DC: *device coordinates*, o también llamadas *screen coordinates*, o *window coordinates*). Las coordenadas X e Y en DC se expresan en unidades de pixels.
- ▶ La transformación de viewport es lineal y consta simplemente de escalados y traslaciones.
- ▶ La coordenada Z se transforma y se conserva para poder hacer después *eliminación de partes ocultas*.

Coordenadas de dispositivo

En coordenadas de dispositivo, podemos asociar una región cuadrada (de lado unidad) a cada pixel en el plano de la ventana. El viewport (en rojo) es un bloque rectangular de pixels, contenido en el bloque rectangular correspondiente a la ventana o imagen completa:



los centros de los pixels (puntos azules) tienen coordenadas de dispositivo con parte fraccionaria igual a $1/2$. Los bordes entre pixels tienen coordenadas sin parte fraccionaria (enteras).

Matriz del viewport y parámetros en OpenGL.

OpenGL tiene en su estado una matriz 4x4 que llamaremos V , y que depende de estos parámetros (ver fig.)

x_l, y_b número de columna y fila (enteros no negativos) del pixel que ocupa, en la ventana, la esquina inferior izquierda del viewport.

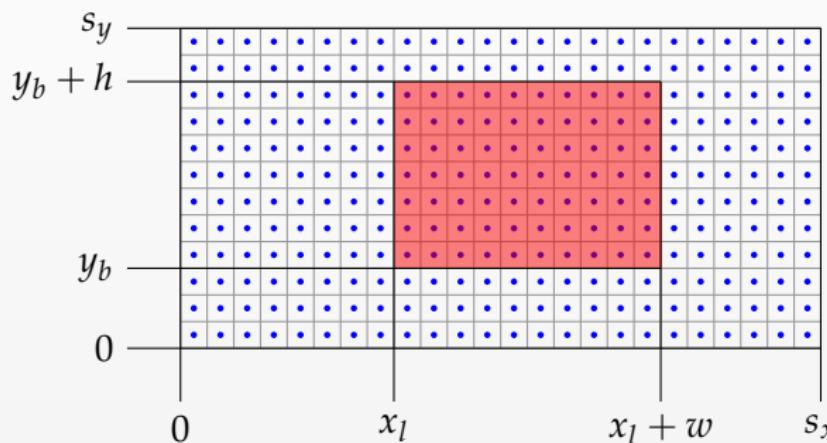
w, h (*width* y *height*) número total (entero no negativo) de columnas y de filas de pixels (respectivamente) que ocupa el viewport.

n_d, f_d rango de valores de salida en Z en DC. El valor n_d es la profundidad más cercana posible al observador, y f_d la más lejana. Por defecto $n_d = 0$ y $f_d = 1$.

aunque los cuatro parámetros relevantes (x_l, y_b, w y h) son enteros, las coordenadas de dispositivos son valores reales, ya que las posiciones de los vértices en DC son en general no enteras (no coinciden necesariamente con los centros o bordes de los pixels).

Parámetros del viewport

Suponemos que la ventana tiene s_x columnas y s_y filas, y que el gestor de ventanas acepta coordenadas de pixels enteras no negativas:



se deben cumplir estas desigualdades: $\begin{cases} 0 \leq x_l < x_l + w \leq s_x \\ 0 \leq y_b < y_b + h \leq s_y \end{cases}$

La transformación de viewport

En NDC las coordenadas están en $[-1, 1]$, luego hay que hacer:

1. traslación de la esquina $(-1, -1, -1)$ al origen.
2. escalado uniforme (por $1/2$) y por $(w, h, f_d - n_d)$
3. traslación del origen a (x_l, y_b, n_d) .

con lo cual la transformación D queda como:

$$D = D[x_l, y_b, n_d] \cdot E[w, h, f_d - n_d] \cdot E[1/2] \cdot D[1, 1, 1]$$

por tanto, las **coordenadas de dispositivo** $(x_d, y_d, z_d, 1)$ se definen a partir de las normalizadas $(x_n, y_n, z_n, 1)$ de esta forma:

$$x_d = (x_n + 1)w/2 + x_l$$

$$y_d = (y_n + 1)h/2 + y_b$$

$$z_d = (z_n + 1)(f_d - n_d)/2 + n_d$$

La matriz de viewport D

Por tanto, la **matriz de viewport D** debe definirse así:

$$D = \begin{pmatrix} \frac{w}{2} & 0 & 0 & x_l + \frac{w}{2} \\ 0 & \frac{h}{2} & 0 & y_b + \frac{h}{2} \\ 0 & 0 & \frac{z_f - z_n}{2} & \frac{z_f + z_n}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

de forma que:

$$(x_d, y_d, z_d, 1)^t = D (x_n, y_n, z_n, 1)^t$$

Fijar la matriz de viewport en OpenGL

En cualquier momento (independientemente del *matrix mode* activo en dicho momento) es posible cambiar la matriz D que OpenGL almacena como parte de su estado.

Para ello llamamos a la función **glViewport**, declarada como sigue:

```
glViewport(GLint xl, GLint yb, GLsizei w, GLsizei h);
```

- ▶ Los rangos de valores permitidos para estos parámetros dependen de la implementación, del hardware subyacente y del gestor o librería de ventanas en uso.
- ▶ Si w y/o h son demasiado grandes, no se produce error, pero se truncan.
- ▶ Por defecto, OpenGL fija el viewport ocupando todos los pixels de la ventana.

Subsección 1.6

Representación de parámetros de las transformaciones

Parámetros necesarios

En muchas aplicaciones es necesario representar los parámetros de las transformaciones:

- ▶ La matriz ***modelview***: obtenida como composición de la matriz de **modelado**, seguida de la matriz de **vista**.
- ▶ La matriz de ***proyección*** (ya sea perspectiva u ortográfica).

Lo anterior hace necesario conocer todos los parámetros de la cámara virtual que nuestro programa necesita:

- ▶ Parámetros de la **vista**: marco de coordenadas de la cámara
- ▶ Parámetros del ***view-frustum***: valores l,r,t,b,n,f (+ un lógico).
- ▶ Parámetros del ***viewport*** (esto es igual que antes: se necesitan para **glViewport**)

Veremos varias estructuras (o clases) para la aplicación C/C++, que soportan esto.

Párametros de transformación de vista

Para representar adecuadamente la transformación de vista, básicamente debemos de almacenar el marco de coordenadas de la cámara, se pueden usar dos clases como estas:

```
class MarcoCoorVista // marco de coordenadas (cartesiano) de la vista
{ public:
    Tupla3f org,           // origen del marco de coordenadas
    eje[3] ; // ejes (0=X, 1=Y, 2=Z) del marco de cc. (ortonormales)
    Matriz4f matrizML, // matriz marco -> mundo
    matrizLM; // matriz mundo -> marco (inversa de la anterior).

    // foco en origen, mirando hacia Z-, vup = Y+
    MarcoCoorVista() ;
    // constructor: mismos parámetros que gluLookAt )
    MarcoCoorVista( const Tupla3f & pfoco, const Tupla3f & paten,
                    const Tupla3f & pvup );
};
```

Transformación de proyección

Los parámetros de la transformación de proyección (determinan el *view-frustum*) se engloban en esta clase:

```
class ViewFrustum
{
    public:
        bool      persp ;           // true para perspectiva, false para ortográfica
        float     left, right,      // extensión en X (left < right)
                  bottom, top,     // extensión en Y (bottom < top)
                  near, far ;       // extensión en Z (rama negativa) (0 < near < far)
        Matriz4f  matrizProy ;     // matriz de proyección P: cc. cámara ==> cc. recortado

        ViewFrustum() ;           // view-frustum ortográfico, de lado 2, centro en (0,0,0)

        // crea view-frustum perspectiva, mismos parámetros que gluPerspective
        ViewFrustum( float hfovy, float aspect, float zNear, float zFar );
};
```

Clase para las cámaras

Una **cámara** es un objeto que contiene tanto los parámetros de la transformación de proyección como de la transformación de vista, y por tanto determina como se visualiza un escena en pantalla. Se puede declarar así:

```
class Camara
{
    public:
        MarcoCoorVista  mcv ; // marco de coordenadas de la vista
        ViewFrustum      vf ; // parámetros de la proyección

        Camara() ;          // usa constructores por defecto para mc y vf
        void fijarMVPogl() ; // fijar matrices MODELVIEW y PROJECTION de OpenGL
};
```

estos objetos constituyen las camaras virtuales usadas en la visualización.

Transformación de viewport

La matriz de viewport, y su inversa, son necesarias en determinadas aplicaciones (p.ej: para selección, lo veremos en el tema 4). Para ello, podemos definir esta clase:

```
class Viewport
{
    public:
        int      org_x, org_y, // origen en pixels (esquina inferior izquierda)
                 ancho, alto ; // dimensiones en pixels (núm. columnas, núm. filas)
        float   ratio_yx ;   // == alto/ancho (relación de aspecto)
        Matriz4f matrizVp , // matriz de viewport ( pasa: NDC ==> DC )
                 matrizVpInv ; // matriz inversa ( pasa: DC ==> NDC )
    // constructor
    Viewport() ; // crea viewport de 512 x 512 con origen en (0,0)
    Viewport( int p_org_x, int p_org_y, int p_ancho, int p_alto );
};
```

- └ Cauce gráfico y definición de la cámara.

- └ Representación de parámetros de las transformaciones

Parámetros para transformación de vértices.

El conjunto de parámetros de la transformación de visualización se pueden agrupas en una única clase, que también contiene las dos matrices que se calculan a partir de ellos:

```
class ParTransf
{
public:
    Camara    cam ; // Cámara activa actualmente para la visualización
    Viewport   vp ; // viewport actualmente en uso
    ParTransf() {} ; // usa constructores por defecto (no hace nada)
};
```

Es conveniente tener estos datos en el contexto de visualización, que se debe ampliar:

```
class ContextoVis
{
public:
    .....
    ParTransf pt ; // parámetros de transformación actuales.
};
```

Sección 2

Modelos de Iluminación

- 2.1. Radiación visible: características, percepción y reproducción computacional.
- 2.2. Emisión y reflexión de la radiación.
- 2.3. Simplificaciones en los modelos computacionales básicos.

Introducción

En este capítulo se hará una introducción a las últimas etapas del cauce gráfico de OpenGL, las encargadas de calcular un color en cada pixel:

- ▶ 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 incorpora un modelo sencillo y computacionalmente eficiente para esto.

Subsección 2.1

Radiación visible: características, percepción y reproducción computacional.

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*).
- ▶ La emisión e interacción de las ondas en los átomos nos permite percibir el entorno.
- ▶ Físicamente, la radiación se describen como algo que tiene características de onda y de corpúsculo a la vez (modelos complementarios).
- ▶ En Informática Gráfica se usa más frecuentemente el *modelo de partículas* (*óptica geométrica*) en lugar del *modelo de ondas* (*óptica física*).

El modelo de partículas. La radiancia.

Bajo este modelo, la radiación se puede describir de forma idealizada como un flujo en el espacio de partículas puntuales llamadas **fotones**, con trayectorias rectilíneas.

- ▶ Cada uno tiene una *energía radiante* que depende únicamente de su longitud de onda (es inv. prop.)
- ▶ En un entorno de punto **p** del espacio (típicamente en la superficie de un objeto) podemos medir la densidad de energía radiante por unid. de tiempo de los fotones de una longitud de onda λ que pasan por **p** en una determinada dirección **v** (un vector libre)

Esa energía se denomina **radiancia** y se nota como $L(\lambda, \mathbf{p}, \mathbf{v})$.

La radiancia determina el tono de color y el brillo con el que observamos el punto **p** cuando lo vemos desde la dirección **v**.

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.



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

Percepción de radiación visible

El ojo es la parte del *sistema visual humano* (SVH) capaz de enviar señales eléctricas al cerebro que dependen de las características de la luz que incide sobre las neuronas de su cara interna (la retina)

- ▶ En cada neurona de la retina, y para cada longitud de onda λ , se recibe una radiancia $L(\lambda)$ distinta.
- ▶ El ojo funciona de forma tal que *simplifica* esa gran cantidad de información y la reduce (en cada neurona) a tres valores reales positivos que forman una tupla (s, m, l) que depende de L , es decir, el ojo tiene asociada una función f tal que:

$$f(L) = (s, m, l)$$

- ▶ Esta simplificación es aprox. lineal, es decir si $f(L) = (s, m, l)$ y $f(L') = (s', m', l')$, entonces:

$$f(aL + bL') = a(s, m, l) + b(s', m', l')$$

donde a, b son valores reales arbitrarios no negativos

Los primarios RGB.

Si x es un valor real ($x > 0$), entonces:

- ▶ la señal $(x, 0, 0)$ enviada desde el ojo se interpreta en el cerebro (SVH) como de color rojo.
- ▶ la señal $(0, x, 0)$ se interpreta como de color verde.
- ▶ la señal $(0, 0, x)$ se interpreta como de color azul.

Como consecuencia, supongamos que tenemos tres distribuciones de radiancia L_r, L_g y L_b tales que:

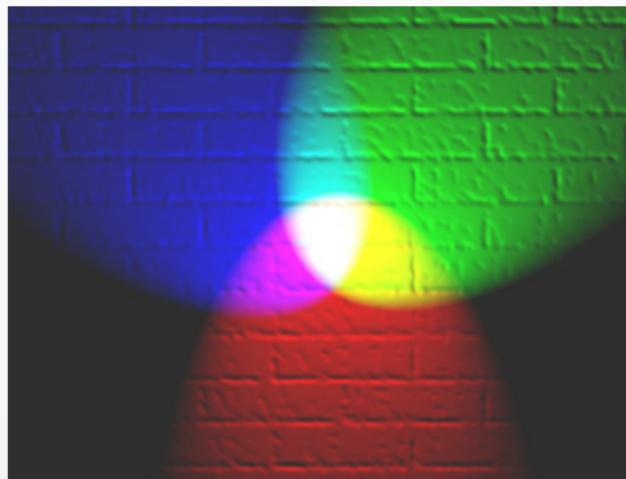
$$f(L_r) \approx (1, 0, 0) \quad f(L_g) \approx (0, 1, 0) \quad f(L_b) \approx (0, 0, 1) \quad (1)$$

A una terna de distribuciones L_r, L_g y L_b que cumplen lo anterior se le denomina una terna de **primarios RGB**, ya que son percibidos como rojo, verde y azul, respectivamente.

Mezcla aditiva de primarios

Podemos usar una *mezcla aditiva* (suma ponderada) de tres primarios RGB para producir una señal arbitraria (r, g, b) en el ojo, ya que se cumple:

$$f(rL_r + gL_g + bL_b) \approx (r, g, b)$$



(imagen obtenida de: http://en.wikipedia.org/wiki/RGB_color_model)

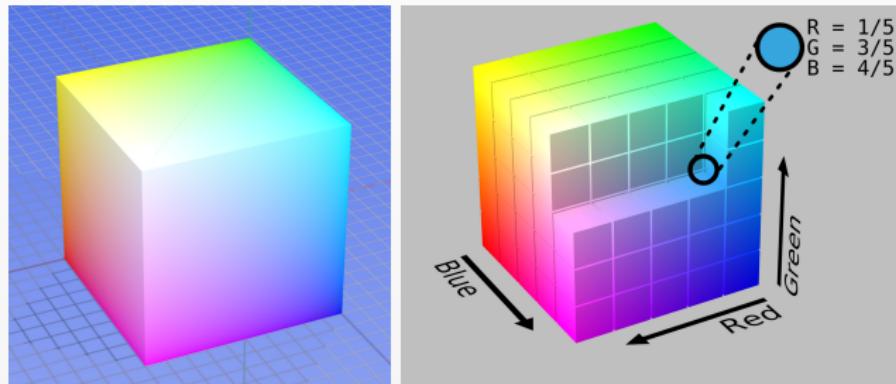
Reproducción de ternas RGB

Un dispositivo de salida de color (monitor, impresora, proyector) tiene asociados tres primarios RGB (las distribuciones obtenidas cuando se muestra el rojo, verde y azul a máxima potencia en el dispositivo)

- ▶ 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.
- ▶ Una misma terna (r, g, b) niveles de brillo que pueden variar mucho entre dispositivos.

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.



(obtenidas de: 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.

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:



(imagen obtenida de: [sitio web de CBC news](#))

Subsección 2.2

Emisión y reflexión de la radiación.

Fuentes de luz y reflectores:

La radiación electromagnética visible se genera en las **fuentes de luz**, por procesos físicos diversos que convierten otras formas de energía en energía radiante. Hay de dos tipos:

- ▶ Fuentes naturales: Sol o estrellas, fuego, objetos incandescentes, órganos de algunos animales, etc...
- ▶ Fuentes artificiales (luminarias): filamentos incandescentes, tubos fluorescentes, LEDs, etc...

Los fotones creados en las luminarias interactúan con los átomos de la materia, que absorben su energía y después pueden radiar de nuevo una parte de ella, proceso conocido como **reflexión**:

- ▶ parte de la energía recibida se convierte en calor
- ▶ parte de la energía recibida se convierte en radiación reflejada
- ▶ la radiación reflejada puede reflejarse de nuevo varias veces

Modelo de la reflexión local en un punto

La radiancia $L(\lambda, \mathbf{p}, \mathbf{v})$ se puede escribir como suma de:

- ▶ la **radiancia emitida** desde \mathbf{p} en la dirección \mathbf{v} (0 si \mathbf{p} no está en una fuente de luz), que llamamos $L_{em}(\lambda, \mathbf{p}, \mathbf{v})$
- ▶ la **radiancia reflejada**, suma, para cada dirección \mathbf{u}_i del producto de:

$L_{in}(\lambda, \mathbf{p}, \mathbf{u}_i) \equiv$ radiancia incidente sobre \mathbf{p} desde \mathbf{u}_i

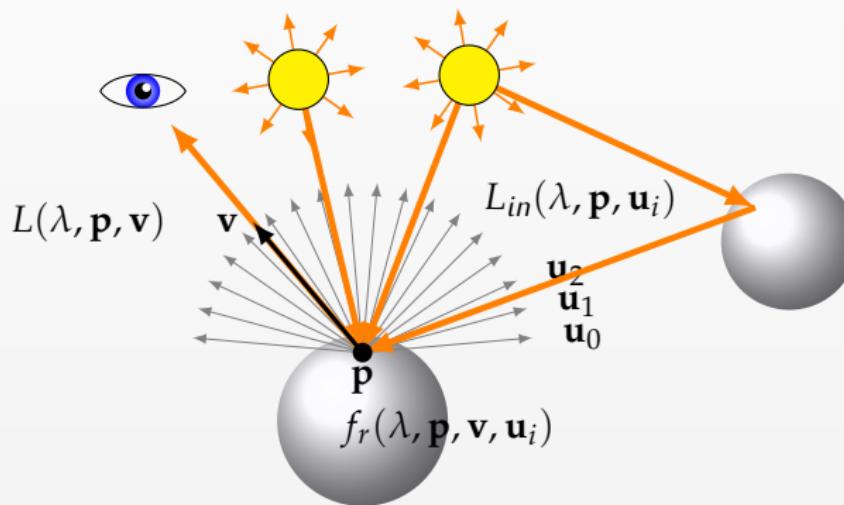
$f_r(\lambda, \mathbf{p}, \mathbf{v}, \mathbf{u}_i) \equiv$ fracción de radiancia que se refleja desde \mathbf{p} en la dirección \mathbf{v} , respecto del total incidente sobre \mathbf{p} proveniente de la dirección \mathbf{u}_i (con l.o. λ)

es decir:

$$L(\lambda, \mathbf{p}, \mathbf{v}) = L_{em}(\lambda, \mathbf{p}, \mathbf{v}) + \sum_i L_{in}(\lambda, \mathbf{p}, \mathbf{u}_i) f_r(\lambda, \mathbf{p}, \mathbf{v}, \mathbf{u}_i)$$

Reflexión local en un punto

Hay muchas trayectorias de fotones que no acaban siendo detectadas por el observador (la mayoría), además las que sí llegan pueden hacerlo por muchos caminos distintos:



Subsección 2.3

Simplificaciones en los modelos computacionales básicos.

Simplificaciones para modelos básicos

La ecuación anterior es complicada (larga) de calcular. Por tanto en OpenGL básico se hacen varias simplificaciones:

- 1 Las fuentes de luz son puntuales o unidireccionales, no extensas, y hay un número finito de ellas.
- 2 No se considera la luz incidente que no provenga directamente de las fuentes de luz (se usa una radiancia *ambiente* constante para suplir la iluminación indirecta).
- 3 Los objetos o polígonos son totalmente opacos (no hay transparencias ni mat. translúcidos).
- 4 No se consideran sombras arrojadas (las fuentes son visibles desde cualquier cara delantera respecto de ellas).
- 5 El espacio entre los objetos no dispersa la luz (la radiancia se conserva en el espacio entre los objetos).
- 6 En lugar de considerar todas las longitudes de onda λ posibles, usamos el modelo RGB.

Efecto de las simplificaciones.

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

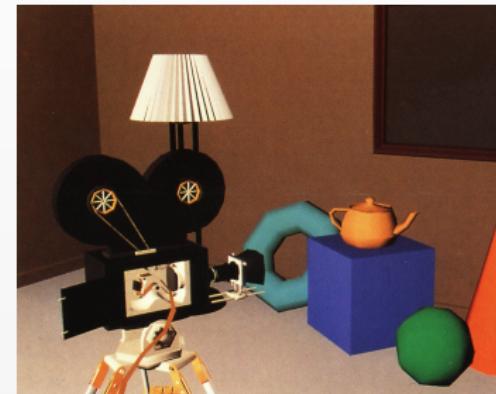
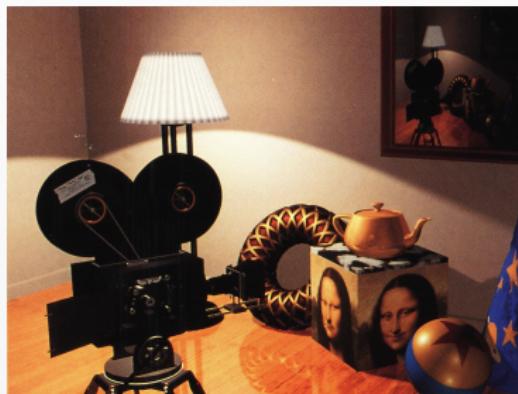


Imagen obtenida de: Computer Graphics: Principles and Practice in C (2nd Edition) Foley, van Dam, Feiner, Hughes.

Modelo simplificado

El modelo que hemos visto antes se simplifica:

- ▶ La iluminación indirecta se reduce a un término ambiente L_{am} que no depende de \mathbf{v} .
- ▶ De todas las direcciones \mathbf{u}_i , solo es necesario considerar las que apuntan hacia una fuente de luz.
- ▶ Todas las fuentes de luz son visibles desde un punto.
- ▶ Los valores de radiancia ($L, L_{em}, L_{in}, L_{am}$) son tuplas (r, g, b) (no acotadas)
- ▶ Los valores de reflectividad (f_r) son tuplas (r, g, b) (entre 0 y 1)

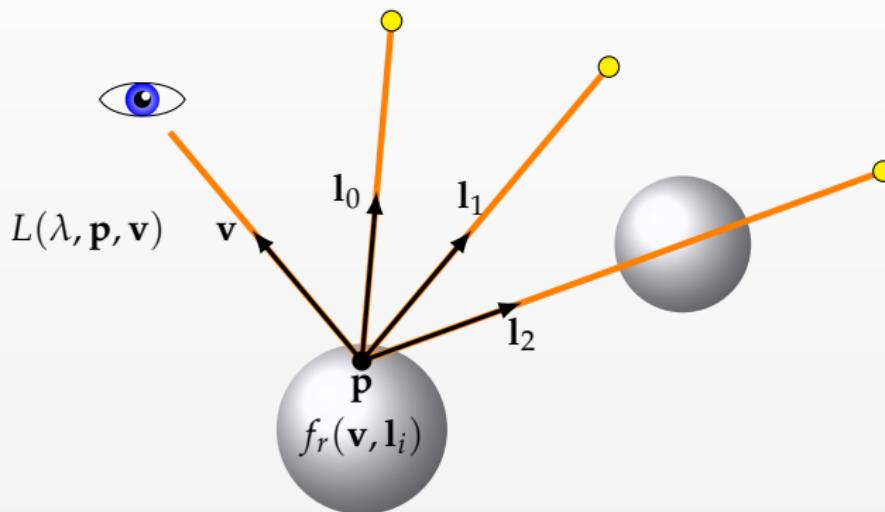
Por tanto:

$$L(\mathbf{p}, \mathbf{v}) = L_{em}(\mathbf{p}) + L_{am}(\mathbf{p}) + \sum_{i=0}^{n-1} L_{in}(\mathbf{p}, \mathbf{l}_i) f_r(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) \quad (2)$$

donde: $n \equiv$ número de fuentes de luz, $\mathbf{l}_i \equiv$ vector que apunta desde \mathbf{p} en la dirección de la i -ésima fuente de luz.

Modelo simplificado (figura)

Ahora solo consideramos trayectorias desde las luminarias hacia p , las luminarias se cuentan aunque la trayectoria esté bloqueada (no hay sombras arrojadas)



Sección 3

Un Modelo de Iluminación Local (MIL) básico.

- 3.1. Elementos del modelo útiles para iluminación: normales y colores.
- 3.2. Fuentes de luz, materiales y reflexión
- 3.3. Componentes del modelo.
- 3.4. El modelo completo.

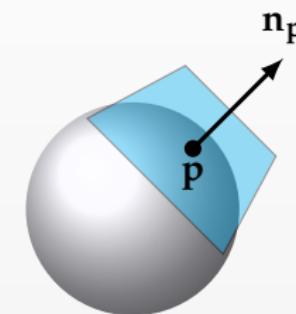
Subsección 3.1

Elementos del modelo útiles para iluminación: normales y colores.

El vector normal

La iluminación (la función f_r en la eq.2) depende la orientación de la superficie en el punto \mathbf{p} . Esta orientación esta caracterizada por el **vector normal \mathbf{n}_p** asociado a dicho punto:

- ▶ \mathbf{n}_p es un vector, de longitud unidad, que depende de \mathbf{p} .
- ▶ idealmente es perpendicular al plano tangente a la superficie en el punto \mathbf{p} (en azul en la fig.)
- ▶ en modelos de fronteras, puede calcularse de varias formas (depende del *método de sombreado*, que veremos más adelante).
- ▶ constituye un parámetro de f_r



Subsección 3.2

Fuentes de luz, materiales y reflexión

Tipos y atributos de las fuentes de luz

En el modelo de escena se puede incluir un conjunto de n fuentes de luz (numeradas de 0 a $n - 1$), cada una de ellas puede ser de dos tipos:

- ▶ Fuentes de luz **posicionales**: ocupan un punto del espacio \mathbf{q}_i . Dado un punto \mathbf{p} , el vector unitario que apunta hacia la fuente de luz desde \mathbf{p} se calcula como:

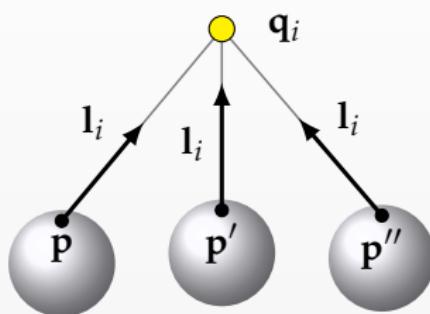
$$\mathbf{l}_i = \frac{\mathbf{q}_i - \mathbf{p}}{\|\mathbf{q}_i - \mathbf{p}\|}$$

- ▶ Fuentes de luz **direcionales**: están en un punto a distancia infinita, por tanto hay un vector \mathbf{l}_i que apunta a la fuente y que es el mismo para cualquier punto \mathbf{p} donde se quiera evaluar el MIL

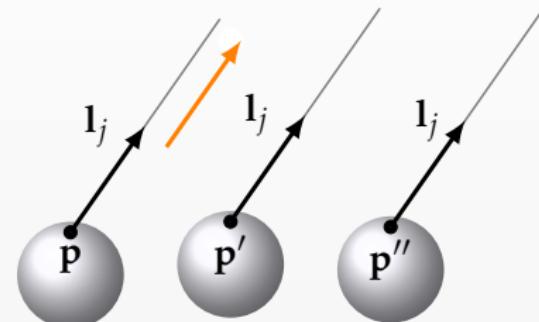
Además de esto, cada fuente de luz emite una radiancia $S_i = (r, g, b)$ (en general no acotada).

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.

Radiancia incidente y tipos de reflexión.

En la ecuación 2 los términos que aparecen pueden reescribirse en términos de los atributos de las fuentes de luz y el material

- ▶ El término $L_{in}(\mathbf{p}, \mathbf{l}_i)$ se hace igual a S_i (no tenemos en cuenta la distancia a la que está la fuente de luz)
- ▶ El término $f_r(\mathbf{p}, \mathbf{v}, \mathbf{l}_i)$ se descompone en tres sumandos o componentes
 - ▶ Luz indirecta reflejada, o término **ambiental**: $f_{ra}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i)$.
 - ▶ Luz reflejada de forma **difusa**: $f_{rd}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i)$.
 - ▶ Luz reflejada de forma **pseudo-especular**: $f_{rs}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i)$.

La ecuación 2 queda como sigue:

$$L(\mathbf{p}, \mathbf{v}) = L_{em}(\mathbf{p}) + L_{am}(\mathbf{p}) + \sum_{i=0}^{n-1} S_i (f_{ra}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) + f_{rd}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) + f_{rs}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i)) \quad (3)$$

Subsección 3.3

Componentes del modelo.

Emisión y componente ambiental global

Los dos primeros sumandos de la ecuación 3 son:

- ▶ Radiancia emitida $L_{em}(\mathbf{p})$, que se puede hacer igual a una tupla RGB $M_E(\mathbf{p})$ que depende del punto \mathbf{p} (es decir, del material, y que puede variar en función del polígono u objeto al que pertenece \mathbf{p}) y que llamamos **emisividad del material**.
- ▶ El término ambiente $L_{am}(\mathbf{p})$ se hace igual a una térrna RGB (que notamos como $A_G(\mathbf{p})$) y que llamamos **luz ambiente global**.

la ecuación 3 se reescribe por tanto como:

$$\begin{aligned} L(\mathbf{p}, \mathbf{v}) &= M_E(\mathbf{p}) + A_G(\mathbf{p}) + & (4) \\ &\quad \sum_{i=0}^{n-1} S_i [f_{ra}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) + f_{rd}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) + f_{rs}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i)] \end{aligned}$$

Componente ambiental específica de cada punto

Cada objeto puede reflejar más o menos cantidad de iluminación indirecta proveniente de la i -ésima fuente de luz.

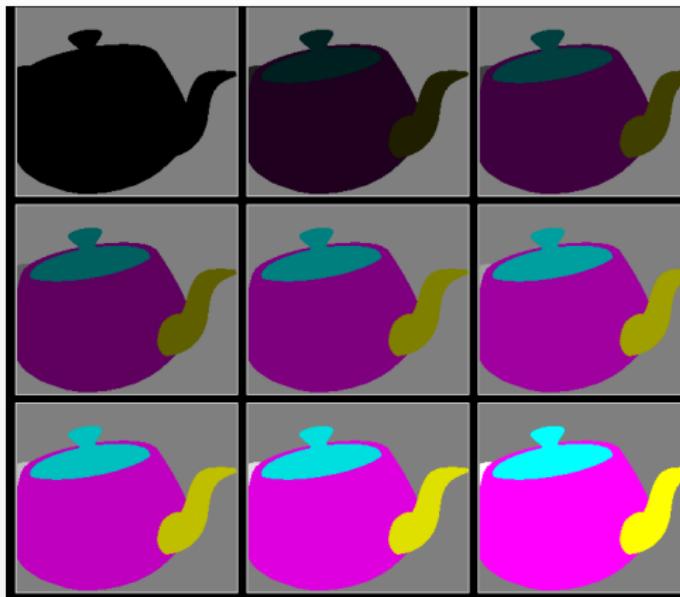
Para poder hacer esto, se asocia a cada polígono o punto una reflectividad difusa del material, una terna RGB que notamos como $M_A(\mathbf{p})$ (con valores entre 0 y 1 pues se trata de una reflectividad), y hacemos:

$$f_{ra}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) = M_A(\mathbf{p}) \quad (5)$$

nótese que este valor no depende de la posición, distancia u orientación de la fuente de luz, ni del vector \mathbf{v} , y por tanto da lugar a colores planos en los objetos.

Reflectividad ambiental del objeto:

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



Componente difusa: expresión.

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

- ▶ La componente **sí depende** de la posición u 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 $\mathbf{n}_\mathbf{p}$ y \mathbf{l}_i),
- ▶ **no depende** de la dirección \mathbf{v} en la que miramos \mathbf{p} (el punto \mathbf{p} se ve de un color igual desde cualquier dirección que lo veamos).
- ▶ La fracción de luz reflejada es igual a una terna de reflectividades $M_D(\mathbf{p})$ que puede hacerse depender de \mathbf{p}

La expresión concreta de f_{rd} es esta:

$$f_{rd}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) = M_D(\mathbf{p}) \max(0, \mathbf{n}_\mathbf{p} \cdot \mathbf{l}_i) \quad (6)$$

Orientación de la superficie:

La orientación de la superficie respecto de la fuente de luz viene determinada por el valor α , que es el ángulo que hay entre los vectores \mathbf{n}_p y \mathbf{l}_i (el valor $\mathbf{n}_p \cdot \mathbf{l}_i$ es igual al coseno de α). Se pueden distinguir dos casos:

- ▶ Si $\alpha > 90^\circ$, entonces:
 - ▶ $\cos(\alpha)$ es negativo.
 - ▶ la superficie, en p , está orientada de espaldas a la fuente de luz.
 - ▶ la contribución de esa fuente debe ser 0.
- ▶ Si $0^\circ \leq \alpha \leq 90^\circ$, entonces:
 - ▶ la superficie, en p , está orientada de cara a la fuente de luz.
 - ▶ $\cos(\alpha)$ estará entre 0 y 1 (entre $\cos(90^\circ)$ y $\cos(0^\circ)$).
 - ▶ se puede demostrar que el valor $\cos(\alpha)$ es proporcional a la densidad de fotones por unidad de área que inciden en el entorno de p , provenientes de la i -ésima fuente de luz.

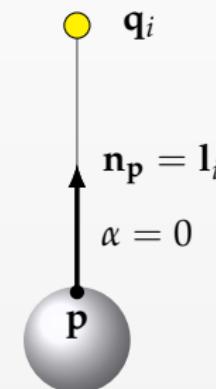
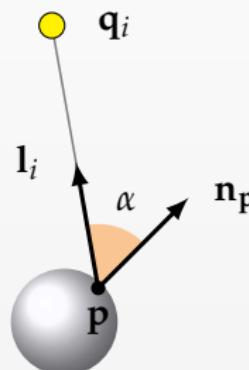
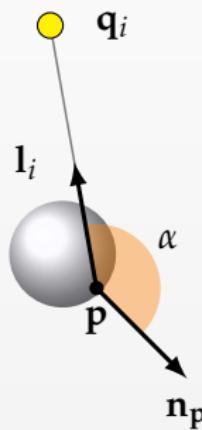
Orientación de la superficie (2)

Aquí se ilustran tres posibles casos:

$$90^\circ < \alpha \\ 0 > \cos(\alpha)$$

$$0^\circ < \alpha < 90^\circ \\ 1 > \cos(\alpha) > 0$$

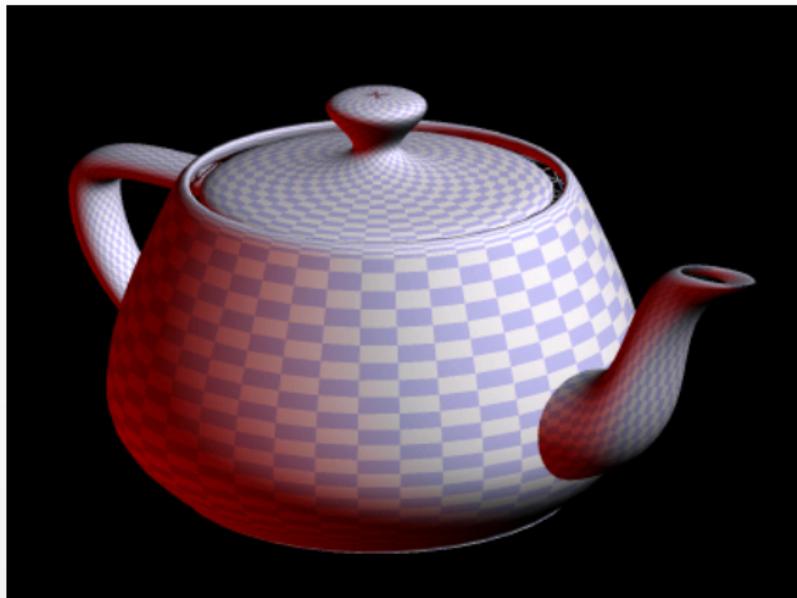
$$\alpha = 0^\circ \\ \cos(\alpha) = 1$$



Material difuso

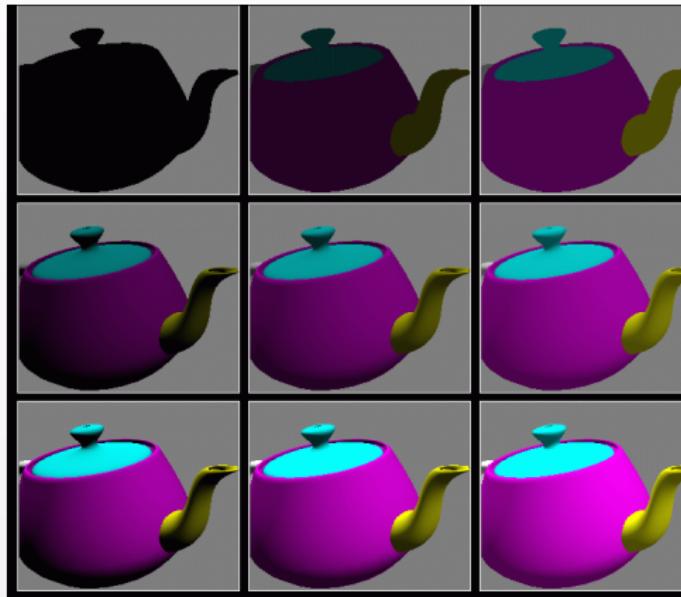
Ejemplo con dos fuentes de luz direccionales,

$M_A(\mathbf{p}) = M_S(\mathbf{p}) = (0, 0, 0)$ (solo hay componente difusa)



Material difuso+ambiental

Aquí M_A crece de izquierda a derecha, y M_D de arriba abajo, $M_S = 0$:



Comp. pseudo-especular: modelo de Phong

La componente **pseudo-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 **sí depende** de la posición u orientación de la fuente de luz (es distinta según como esté orientada la fuente respecto de la superficie en \mathbf{p}),
- ▶ **también depende** de la dirección en la que miramos \mathbf{p} (el punto \mathbf{p} se ve de un color diferente según la dirección en la que lo veamos).
- ▶ La fracción de luz reflejada es proporcional a una terna de reflectividades $M_S(\mathbf{p})$ que puede hacerse depender de \mathbf{p}

La expresión ideada por *Bui Tuong Phong*, y conocida como **modelo de Phong** es esta:

$$f_{rs}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) = M_S(\mathbf{p}) d_i [\max(0, \mathbf{r}_i \cdot \mathbf{v})]^e \quad (7)$$

Parámetros del modelo de Phong

En la expresión anterior:

$\mathbf{r}_i \equiv$ **vector reflejado**, depende tanto de \mathbf{l}_i como de \mathbf{n}_p , y está en el plano formado por ambos, con \mathbf{n}_p como bisectriz de ellos, se obtiene como:

$$\mathbf{r}_i = 2(\mathbf{l}_i \cdot \mathbf{n}_p)\mathbf{n}_p - \mathbf{l}_i$$

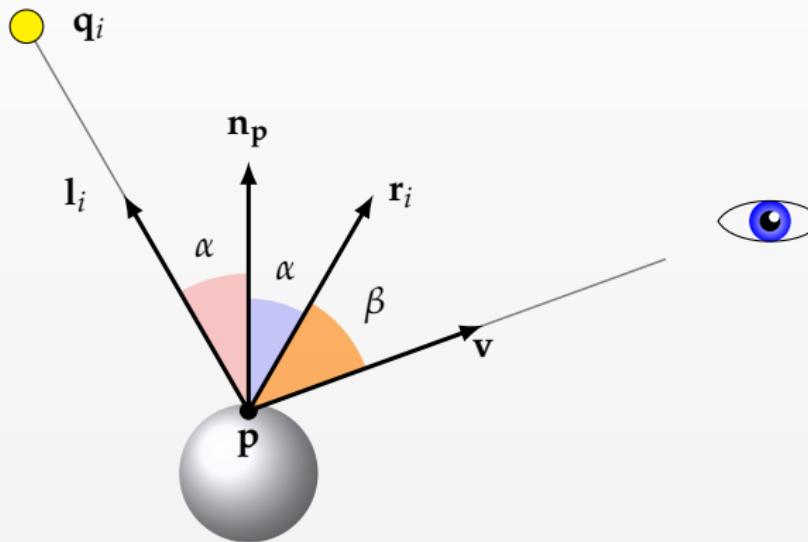
el vector \mathbf{r}_i indica la dirección desde \mathbf{p} en la cual la i -ésima fuente de luz produce el máximo brillo.

$e \equiv$ **exponente de brillo**, un valor real positivo que permite variar el tamaño de las zonas brillantes (a mayor valor, menor tamaño y más pulida o especular).

$d_i \equiv$ vale 1 si $\mathbf{n}_p \cdot \mathbf{l}_i > 0$ (fuente de cara a la superficie), y 0 en otro caso (de espaldas)

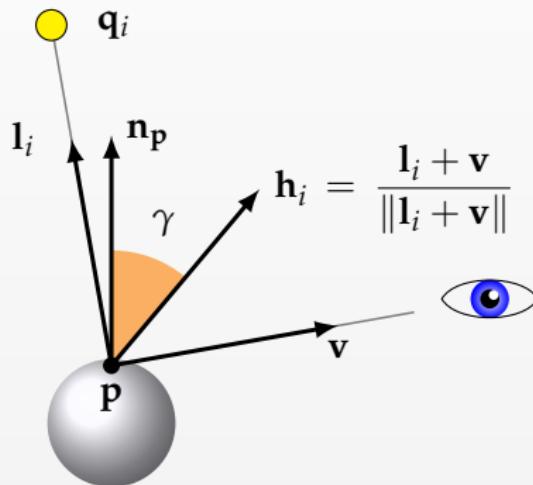
Vectores del modelo de Phong

El valor $\mathbf{r}_i \cdot \mathbf{v}$ es el coseno del ángulo β que hay entre la dirección de máximo brillo \mathbf{r}_i y la dirección \mathbf{v} hacia el observador. Cuando $\mathbf{r}_i = \mathbf{v}$ entonces $\beta = 0^\circ$, $\cos(\beta) = 1$, y el brillo es máximo:



Comp. pseudo-especular: modelo de Blinn-Phong

Una alternativa al modelo anterior consiste en usar el vector *halfway h_i* (bisectriz de \mathbf{l}_i y \mathbf{v} , normalizado). Ahora el brillo es proporcional al coseno del ángulo γ entre \mathbf{h}_i y $\mathbf{n_p}$ (máximo cuando coinciden)



La expresión del **Modelo de Blinn-Phong** es la siguiente:

$$f_{rs}(\mathbf{p}, \mathbf{v}, \mathbf{l}_i) = M_S(\mathbf{p}) d_i [\mathbf{n_p} \cdot \mathbf{h}_i]^e$$

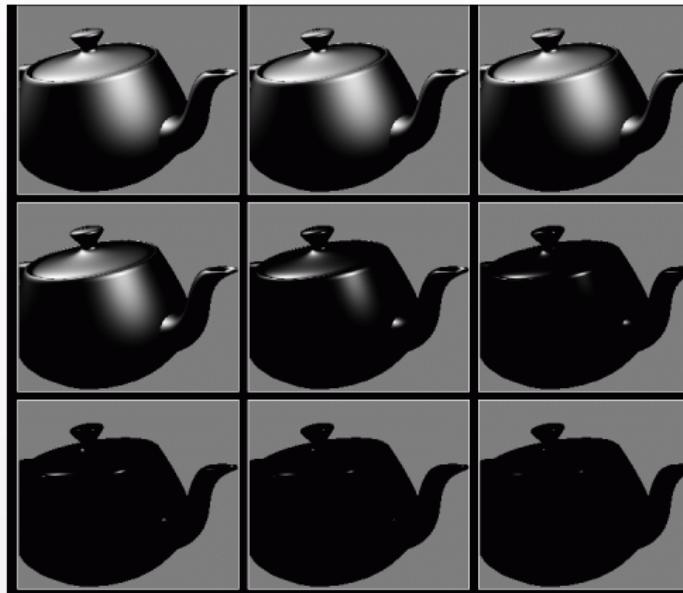
Ejemplo de material pseudo-especular

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



Efecto del exponente de brillo

Aquí crece de izquierda a derecha y de arriba abajo:



Subsección 3.4

El modelo completo.

La expresión del modelo completo:

Sustituyendo la expresiones de f_{ra} , f_{rd} y f_{rs} en la ecuación 4 obtenemos el modelos simplificado completo:

$$L(\mathbf{p}, \mathbf{v}) = M_E(\mathbf{p}) + A_G(\mathbf{p}) + \sum_{i=0}^{n-1} S_i C_i$$

donde:

$$\begin{aligned} C_i &= M_A(\mathbf{p}) \\ &+ M_D(\mathbf{p}) \max(0, \mathbf{n} \cdot \mathbf{l}_i) \\ &+ M_S(\mathbf{p}) d_i [\max(0, \mathbf{r}_i \cdot \mathbf{v})]^e \end{aligned}$$

└ Un Modelo de Iluminación Local (MIL) básico.

└ El modelo completo.

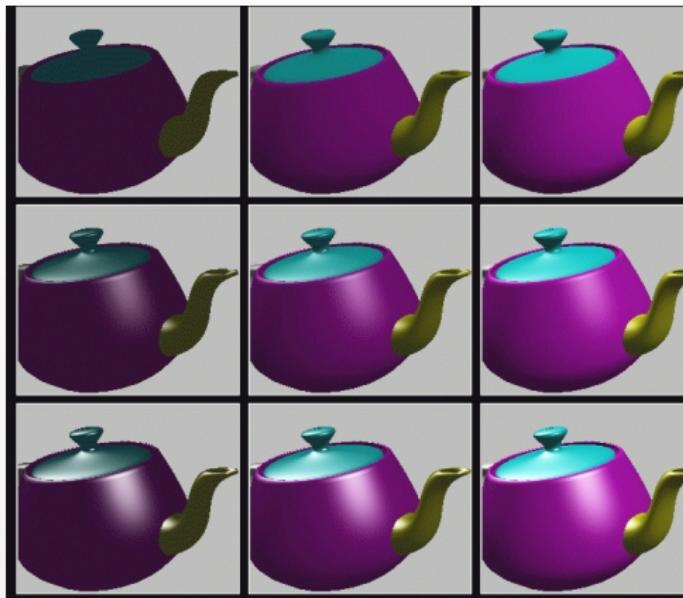
Ejemplo de material combinado

Combinación ambiental, más difusa, más pseudo.especular:



Combinaciones material difuso + pseudo especular

M_D crece de izquierda a derecha y M_S de arriba abajo:



Sección 4

Iluminación en OpenGL.

- 4.1. Introducción: activación, iluminación versus asignación de colores.
- 4.2. Definición de fuentes de luz: tipos y atributos.
- 4.3. Representación de fuentes de luz
- 4.4. El vector hacia el observador
- 4.5. Normales de vértices
- 4.6. Definición de atributos de materiales.
- 4.7. Representación de materiales

Introducción

La librería OpenGL como parte de la funcionalidad fija (pre-programada), incluye una implementación de un modelo de iluminación similar al ya introducido

- 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 fue declarada **obsoleta** (*deprecated*) en la versión 3.0 de la API de OpenGL (Septiembre, 2008), y fue **eliminada** de la versión 3.1 en adelante (Mayo, 2009) (se usa programación del cauce gráfico).

Subsección 4.1

Introducción: activación, iluminación versus asignación de colores.

Cálculo de iluminación

El comportamiento de OpenGL depende de si la evaluación del MIL 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 el MIL se evalua usando unos parámetros incluidos en el estado de OpenGL, y el color resultante obtenido se usa en lugar del especificado con **glColor**.

(activar o desactivar el MIL es independiente del *modo de sombreado* activo en cada momento).

Parámetros del 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.

M_A, M_D, M_S reflectividad difusa, ambiente y pseudo-especular del material.

e exponente de la componente pseudo-especular.

S_{iA}, S_{iD}, S_{iS} luminosidad de la i -ésima fuente de luz (para las componentes ambiental, difusa o pseudo-especular).

$\mathbf{q}_i, \mathbf{l}_i$ posición o dirección de la i -ésima fuente de luz (en EC).

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

El modelo de la funcionalidad fija

Es muy similar al ya visto, excepto que en lugar de usar una sola terna S_i para el color de una fuente de luz, se usan tres de ellas, (S_{iA}, S_{iD}, S_{iS}) , una por cada componente del modelo:

$$I = M_E + A_G + \sum_{i=0}^{n-1} C_i$$

donde:

$$C_i = S_{iA}M_A + S_{iD}M_D \max(0, \mathbf{n} \cdot \mathbf{l}_i) + S_{iS}M_S d_i [\mathbf{n_p} \cdot \mathbf{h}_i]^e$$

el resultado $I = (r, g, b)$ es un color que se asigna al pixel. Si r, g o b tienen un valor superior a 1, dicho valor se trunca a 1.

Evaluación del MIL

La evaluación se hace en coordenadas de ojo (relativa al sist. de ref. de la cámara), usando:

\mathbf{p} ≡ posición del vértice donde se evalúa el MIL, en EC.

\mathbf{v} ≡ vector normalizado desde \mathbf{p} hacia el observador en EC.

Si la proyección es perspectiva, se usa $\mathbf{v} = -\mathbf{p}/\|\mathbf{p}\|$, si es ortogonal se usa $\mathbf{v} = (0, 0, 1)$.

Subsección 4.2

Definición de fuentes de luz: tipos y atributos.

Activación y desactivación.

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) ; // activa la i-esima fuente de luz  
glDisable(GL_LIGHTi) ; // desactiva la i-esima fuente de luz
```

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

Configuración de colores de una fuente.

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

```
const float
    caf[4] = { ra, ga, ba, 1.0 }, // color ambiental de la fuente
    cdf[4] = { rd, gd, bd, 1.0 }, // color difuso de la fuente
    csf[4] = { rs, gs, bs, 1.0 }; // color especular de la fuente

    glLightfv( GL_LIGHTi, GL_AMBIENT, caf ) ; // hace  $S_{iA} := (r_a, g_a, b_a)$ 
    glLightfv( GL_LIGHTi, GL_DIFFUSE, cdf ) ; // hace  $S_{iD} := (r_d, g_d, b_d)$ 
    glLightfv( GL_LIGHTi, GL_SPECULAR, csf ) ; // hace  $S_{iS} := (r_s, g_s, b_s)$ 
```

Configuración de posición/dirección de una fuente.

Los posición (en luces posicionales) o dirección (en direccionales) se especifica con una llamada a **glLightfv**, de esta forma:

```
// fuentes posicionales:  $\mathbf{p}_i = (p_x, p_y, p_z)$ 
const GLfloat posf[4] = {  $p_x, p_y, p_z, 1.0$  } ;
glLightfv( GL_LIGHTi, GL_POSITION, posf );

// fuentes direccionales  $\mathbf{l}_i = (v_x, v_y, v_z)$ 
const GLfloat dirf[4] = {  $v_x, v_y, v_z, 0.0$  } ;
glLightfv( GL_LIGHTi, GL_POSITION, dirf );
```

- ▶ El valor de w determina el tipo de fuente de luz.
- ▶ A la tupla (x, y, z, w) se le aplica la matriz *modelview M* activa en el momento de la llamada, y el resultado se almacena y se interpreta en coordenadas de cámara.

Posición u orientación de la fuente.

La tupla (x, y, z, w) puede especificarse en varios marcos de coordenadas:

- ▶ Coordenadas de cámara: si se especifica cuando $M = \text{Id}$.
- ▶ Coordenadas del mundo: si se especifica cuando M contiene la matriz de vista V .
- ▶ Coordenadas maestras (de algún objeto O): si se especifica cuando $M = VN$ (donde N es la matriz de modelado del objeto O)

En todos los casos se pueden usar (adicionalmente) transformaciones específicas para esto, situando dichas transformaciones, seguidas del **glLightfv**, entre **glPushMatrix** y **glPopMatrix**.

Dirección en coordenadas polares

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

```
const float[4] ejeZ = { 0.0, 0.0, 1.0, 0.0 } ;
glMatrixMode( GL_MODELVIEW ) ;
glPushMatrix() ;

glLoadIdentity() ;      // hacer M = Ide
glMultMatrix( A ) ;   // A podría ser Ide,V o VN

// (3) rotación  $\alpha$  grados en torno a eje Y
glRotatef(  $\alpha$ , 0.0, 1.0, 0.0 ) ;
// (2) rotación  $\beta$  grados en torno al eje X-
glRotatef(  $\beta$ , -1.0, 0.0, 0.0 ) ;
// (1) hacer  $\mathbf{l}_i := (0,0,1)$  (paralela eje Z+)
glLightf( GL_LIGHTi, GL_POSITION, ejeZ ) ;

glPopMatrix() ;
```

Subsección 4.3

Representación de fuentes de luz

La clase para fuentes de luz

La clase **FuenteLuz** tiene los atributos de una fuente:

```
class FuenteLuz
{
public:
    Tupla4f posvec; // posición (si w=1) o vector de dirección (si w=0)
    Tupla4f colores[3]; // colores: 0=ambiental, 1=difuso, 2=especular.
    float longi, lati; // ángulos de rotación (si direccional)

    void activar( int i ); // activarla (con ident. OpenGL GL_LIGHT0+i)
};
```

- ▶ el método **activar** habilita (con **glEnable**) una fuente de luz OpenGL, configurandola con los parámetros que hay en la instancia.
- ▶ **longi** y **lati** son las coordenadas esféricas de **posvec**, siempre que la fuente de luz sea direccional (es decir, si **posvec** tiene $w = 0$).

Colecciones de fuentes de luz

El conjunto de fuentes de luz de una escena se puede guardar en una instancia de **ColecciónFL**:

```
class ColecciónFL
{
public:
    std::vector<FuenteLuz *> fuentes ; // vector de fuentes de luz
    void activar( ) ; // activa todas las fuentes de luz
};
```

- ▶ la activación de la colección supone: activar iluminación, y activar todas las fuentes de luz presentes (llamando a **activar** de las fuentes).
- ▶ el número máximo de fuentes será 8 (numeradas de 0 a 7)
- ▶ en **activar**, si hay $n < 8$ fuentes, deben desactivarse las fuentes OpenGL desde la n hasta la 7 (con **glDisable**).

Subsección 4.4

El vector hacia el observador

Observador local o en el infinito

OpenGL debe calcular el vector \mathbf{v} que va desde el punto \mathbf{p} hacia el observador (en EC), esto debe hacerse en función del tipo de proyección activo. La función `glLightModel` permite configurar este comportamiento:

- ▶ Si la proy. es ortogonal, \mathbf{v} debe ser $(0, 0, 1)$ (el observador está en el infinito), es necesario hacer esta llamada:

```
glLightModeli( GL_LIGHT_MODEL_LOCAL_VIEWER, GL_FALSE );
```

- ▶ Si la proy. es perspectiva, \mathbf{v} debe ser $-\mathbf{p}/\|\mathbf{p}\|$ (se dice que el observador es **local**, no está en el infinito) en este caso debemos de hacer:

```
glLightModeli( GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE );
```

Subsección 4.5

Normales de vértices

Asignación de normales a vértices

Cada vez que se llama a **glVertex**, al vértice que se crea se le asocia un vector normal (el valor en ese momento de **n**, que es un vector libre que forma parte del estado de OpenGL). Para cambiar el vector normal, debemos usar **glNormal**. Por ejemplo, para visualizar triángulos con iluminación, debemos de hacer:

```
glBegin(GL_TRIANGLES);
    // hace n := (nx0, ny0, nz0), inserta vértice en (x0, y0, z0)
    glNormal3f( nx0, ny0, nz0 ); glVertex3f( x0, y0, z0 );

    // hace n := (nx1, ny1, nz1), inserta vértice en (x1, y1, z1)
    glNormal3f( nx1, ny1, nz1 ); glVertex3f( x1, y1, z1 );

    // hace n := (nx2, ny2, nz2), inserta vértice en (x2, y2, z2)
    glNormal3f( nx2, ny2, nz2 ); glVertex3f( x2, y2, z2 );

    // ..... otros triángulos ....
glEnd();
```

Normalización del vector normal.

Los vectores normales especificados con `glNormal` se transforman por la matriz *modelview* (M) activa en el momento de la llamada, y se almacenan en coordenadas de cámara. Es necesario que OpenGL use normales de longitud unidad para evaluar el MIL. Para lograrlo hay tres opciones:

- ▶ Enviar normales de longitud unidad (solo válido si M no incluye cambios de escala ni cizallas).
- ▶ Enviar normales de longitud unidad, y habilitar **GL_RESCALE_NORMAL** (solo válido si M no incluye cizallas, aunque puede tener cambios de escala)

```
glEnable(GL_RESCALE_NORMAL); // deshabilitado por defecto
```

- ▶ Enviar normales de longitud arbitraria, y habilitar **GL_NORMALIZE** (válido para cualquier M) (preferible).

```
glEnable(GL_NORMALIZE); // deshabilitado por defecto
```

Subsección 4.6

Definición de atributos de materiales.

Termino ambiente global y emisividad

El término ambiente global (A_G) es una terna RGB que forma parte del estado de OpenGL y que se cambia con la función **glLightModel**, como sigue:

```
GLfloat color[4] = { r, g, b, 1.0 } ;  
// hace  $A_G := (r,g,b)$ , inicialmente es (0.2,0.2,0.2)  
glLightModelf( GL_LIGHT_MODEL_AMBIENT, color ) ;
```

Las propiedades del material también forman parte del estado y se modifican con llamadas a la función **glMaterial**, para modificar la emisividad del material (M_E), hacemos:

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

Colores del material

Además de la emisión, el resto de colores que definen el material (M_A, M_D, M_S) y el exponente de brillo e también se cambian con **glMaterial**, como se indica aquí:

```
GLfloat color[4] = { r, g, b, 1.0 } ;  
  
// hace  $M_A := (r,g,b)$ , inicialmente (0.2,0.2,0.2)  
glMaterialfv( GL_FRONT_AND_BACK, GL_AMBIENT, color ) ;  
  
// hace  $M_D := (r,g,b)$ , inicialmente (0.8,0.8,0.8)  
glMaterialfv( GL_FRONT_AND_BACK, GL_DIFFUSE, color ) ;  
  
// hace  $M_S := (r,g,b)$ , inicialmente (0,0,0)  
glMaterialfv( GL_FRONT_AND_BACK, GL_SPECULAR, color ) ;  
  
// hace  $e := v$ , inicialmente 0.0 (debe estar entre 0.0 y 128.0)  
glMaterialf( GL_FRONT_AND_BACK, GL_SHININESS, v ) ;
```

Asociación de colores del material al color actual

La función **glColor** actualiza una terna RGB que llamaremos C y que forma parte del estado de OpenGL. Con la función **glColorMaterial** podemos hacer que el valor de alguna de las reflectividades del material se haga igual a C cada vez que C cambie (por una llamada a **glColor**)

```
// asociar  $M_E$  (emisión) con  $C$  :
glColorMaterial( GL_FRONT_AND_BACK, GL_EMISSION ) ;
// asociar  $M_A$  (refl. ambiente) con  $C$  :
glColorMaterial( GL_FRONT_AND_BACK, GL_AMBIENT ) ;
// asociar  $M_D$  (refl. difusa) con  $C$  :
glColorMaterial( GL_FRONT_AND_BACK, GL_DIFFUSE ) ;
// asociar  $M_S$  (refl. pseudo-especular) con  $C$  :
glColorMaterial( GL_FRONT_AND_BACK, GL_SPECULAR ) ;
// asociar  $M_A$  y  $M_D$  con  $C$  :
glColorMaterial( GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE ) ;
```

por defecto, esta funcionalidad está activada, y OpenGL asocia siempre M_A y M_D con C . Se puede activar o desactivar con:

```
 glEnable( GL_COLOR_MATERIAL ) ;
 glDisable( GL_COLOR_MATERIAL ) ;
```

Atributos de material de caras delanteras y traseras

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 aquellaos 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:

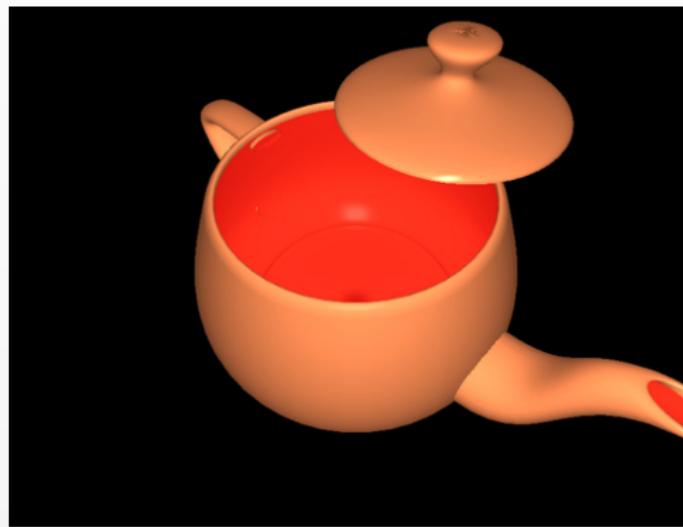


Imagen obtenida de: editorial Packt

Subsección 4.7

Representación de materiales

Materiales abstractos

En adelante, denominaremos como **material** al conjunto de parámetros necesarios para definir un MIL de cualquier tipo en OpenGL.

- ▶ En una aplicación OpenGL, podemos considerar un material como cualquier instancia de una clase que tiene un método para *activar* el material,
- ▶ Una vez activado: el material será usado durante la rasterización de las primitivas que se envíen a continuación de dicha activación, hasta que se haga otra activación de otro material.

Se puede implementar con esta clase base abstracta:

```
class Material
{
public:
    virtual void activar( ContextoVis & cv ) = 0 ;
};
```

El material estándar

Denominamos **material estándar** al material que implementa el MIL que hemos visto en este capítulo. Se puede representar con una clase derivada de **Material**, que contiene los atributos vistos y un método de activación concreto:

```
class MaterialEstandar : public Material
{
public:
    Textura * textura ; // punt. a textura (NULL si no hay)
    Tupla4f color[4] ; // 0=ME, 1=MA, 2=MD, 3=MS
    float exponente ; // exponente (e)

    virtual void activar( ContextoVis & cv ) ; // activa material
};
```

en clases derivadas de **MaterialEstandar** se pueden incluir constructores para distintos tipos de material.

Sección 5

Métodos de sombreado para Z-buffer.

- 5.1. Evaluación del MIL en el contexto del algoritmo de Z-buffer.
- 5.2. Sombreado plano.
- 5.3. Sombreado en los vértices.
- 5.4. Sombreado en los píxeles.
- 5.5. OpenGL: definición del método de sombreado.

Subsección 5.1

Evaluación del MIL en el contexto del algoritmo de Z-buffer.

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 *Gouroud 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 5.2

Sombreado plano.

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).

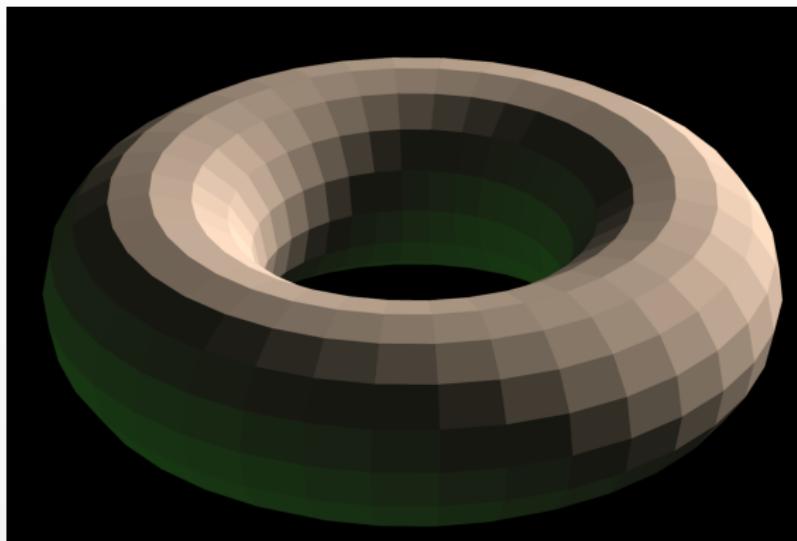
- ▶ Se debe seleccionar un punto cualquiera p de cada polígono, típicamente se usa un vértice, pero podría ser cualquier otro.
- ▶ Se usa la normal al polígono n_p .
- ▶ Se calcula el vector al observador v en p .

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).



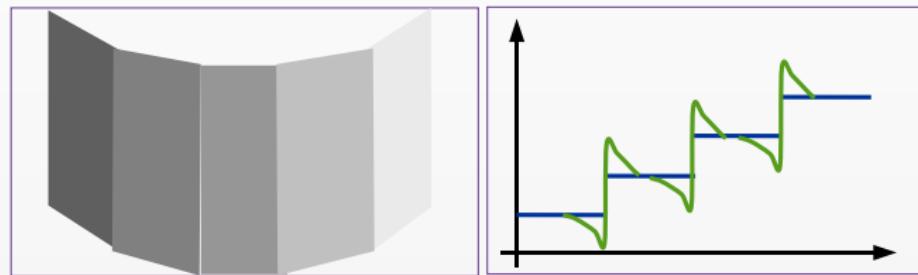
Resultados del sombreado plano

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

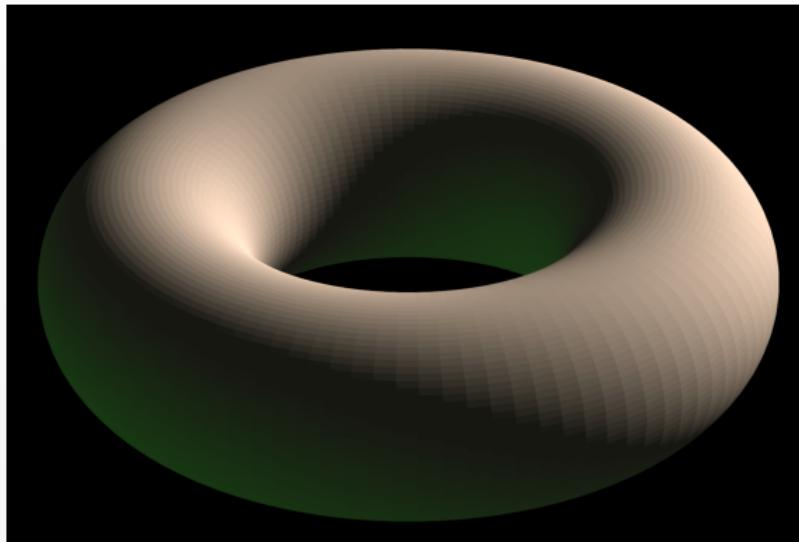
La **Bandas Mach** son una ilusión visual producida por la *inhibición lateral de las neuronas de la retina*, que es un mecanismo desarrollado evolutivamente para resaltar el contraste en aristas entre colores planos:



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 5.3

Sombreado en los vértices.

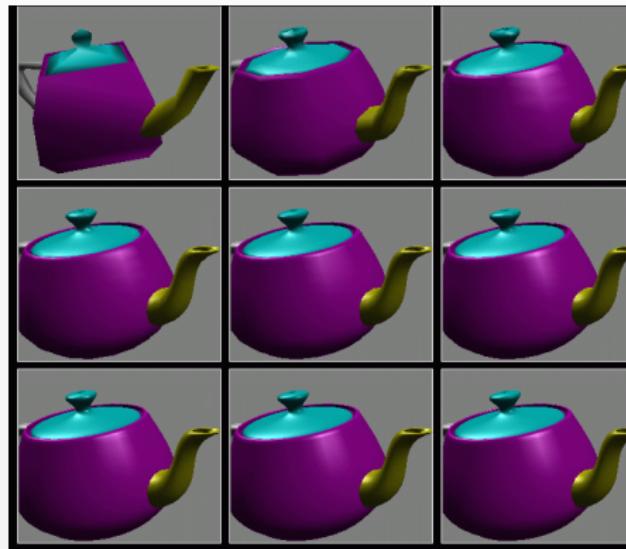
Sombreado en los vértices

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

- ▶ La posición \mathbf{p} coincide con la posición del vértice.
- ▶ Si la malla de polígonos aproxima un objeto curvo, la normal \mathbf{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.

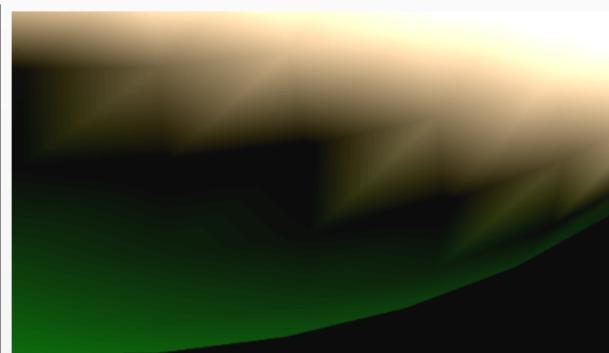
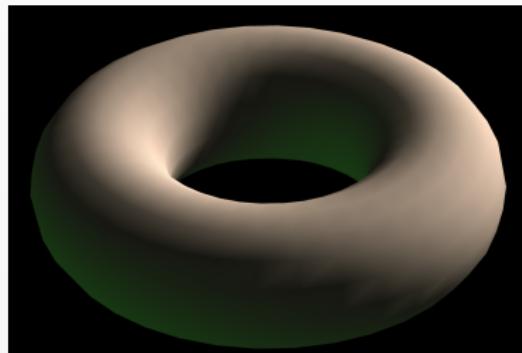
Pérdida de zonas brillantes

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)



(al izquierda aparece una ampliación, con el brillo y contraste aumentado)

Subsección 5.4

Sombreado en los píxeles.

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

- ▶ 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.

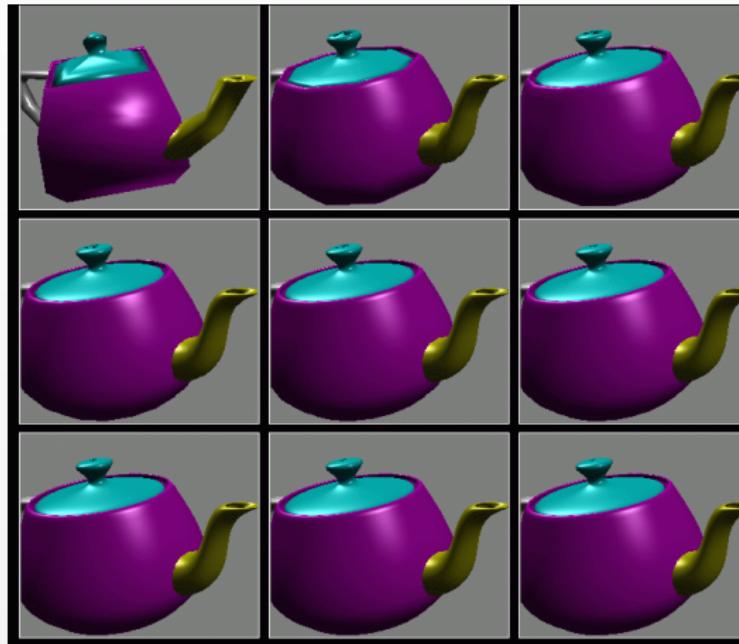
Ejemplo de sombreado

Esta imagen se ha creado con sombreado en los pixels:



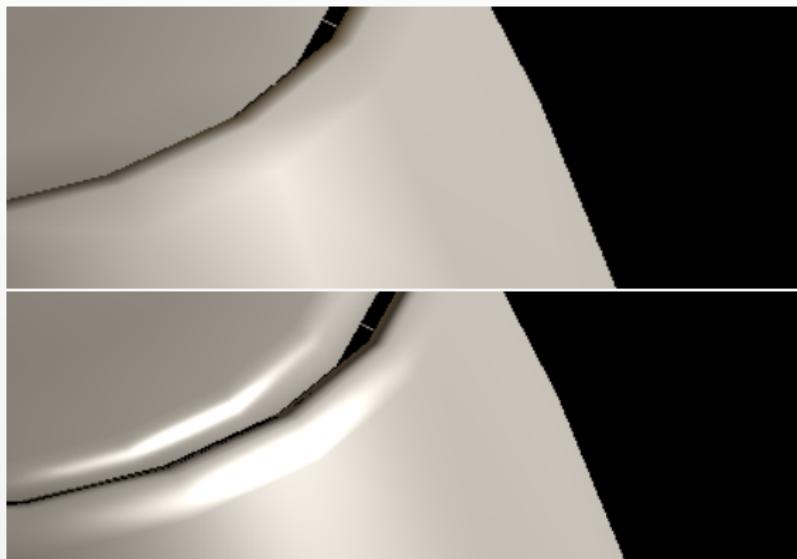
Reproducción de zonas de brillo

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:



Subsección 5.5

OpenGL: definición del método de sombreado.

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.

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

Sombreado y 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.

Sección 6

Visualización de texturas.

- 6.1. Reproducción de detalles a pequeña escala: problemas computacionales.
- 6.2. Coordenadas de texturas.
- 6.3. Asignación explícita de coordenadas de textura.
- 6.4. Asignación procedural de coordenadas de textura.
- 6.5. Consulta de texels en texturas de imagen.

Subsección 6.1

Reproducción de detalles a pequeña escala: problemas computacionales.

Detalles a pequeña escala

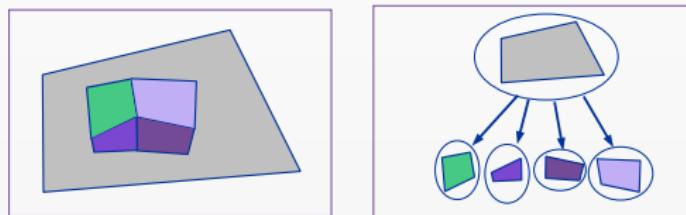
Los objetos reales presentan a veces detalles a pequeña escala, como son manchas, defectos, motivos ornamentales, rugosidades, o, en general, cambios en el espacio de los atributos que determinan su apariencia:



- ▶ 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).

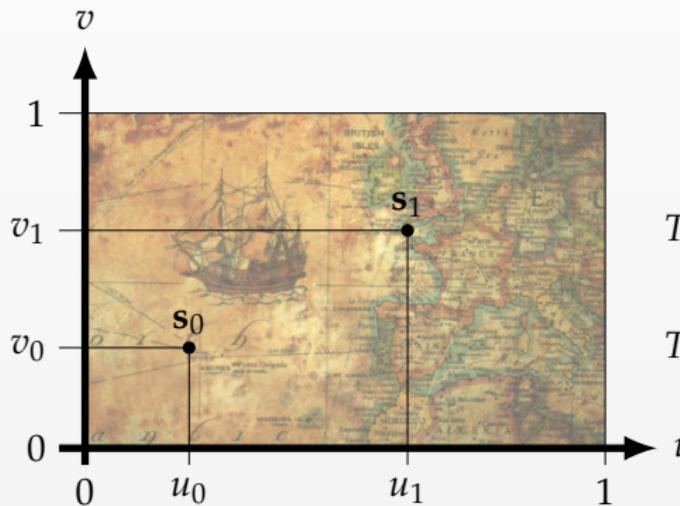
Texturas

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 \mathbf{s} de un dominio D (usualmente $[0, 1] \times [0, 1]$) un valor para un parámetro del MIL (típicamente M_D y M_A). 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 tambien representarse como un subprograma que calcula los valores a partir de \mathbf{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 $\mathbf{s} = (u, v)$. Los valores $T(\mathbf{s}) = T(u, v)$ son ternas RGB.



$$T(s_1) = T(u_1, v_1) = (r_1, g_1, b_1)$$

$$T(s_0) = T(u_0, v_0) = (r_0, g_0, b_0)$$

Aplicación de texturas

Vemos varias formas de asignar colores a puntos del objeto:

- ▶ evaluación del MIL con reflectividades blancas (izq. abajo)
- ▶ uso directo de colores de la textura (izq. arriba)
- ▶ evaluación del MIL con reflectividades obtenidas de la textura (der.)



Resultado (modulación)

Subsección 6.2

Coordenadas de texturas.

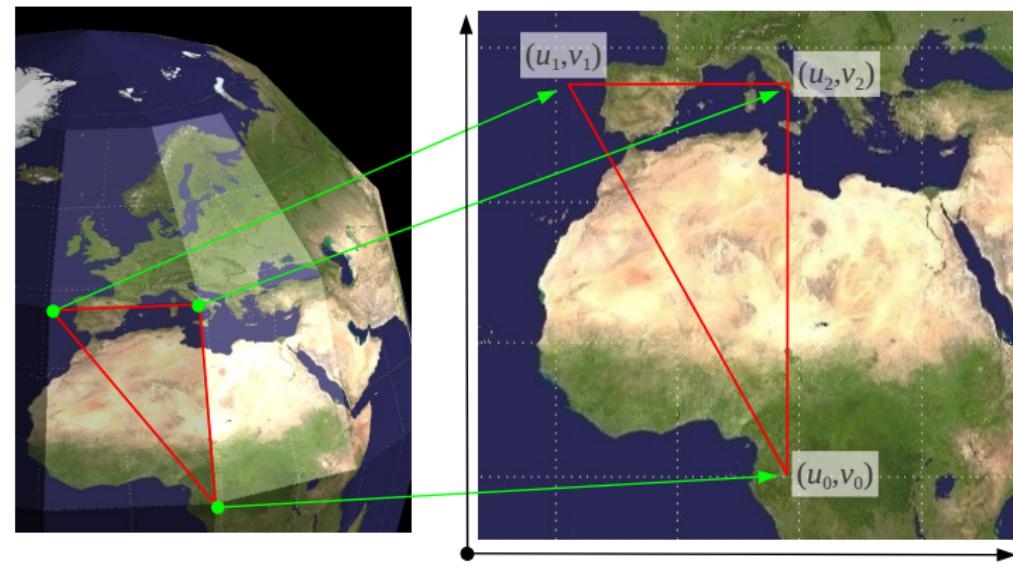
Coordenadas de textura

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.

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 coordenadas de textura se puede hacer de dos formas:

- ▶ **Asignación explícita a vértices:** La 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 $(v_0, u_0), (v_1, u_1), \dots (v_{n-1}, u_{n-1})$.
 - ▶ se puede hacer manualmente en objetos sencillos, o bien
 - ▶ de forma asistida usando software para CAD (p.ej. 3D Studio).

esta modalidad hace necesario realizar una interpolación de coordenadas de textura en el interior de los polígonos de la malla.
- ▶ **Asignación procedural:** La función f se implementa como un subprograma `CoordText(p)` que se invoca para calcular las coordenadas de textura (acepta un punto p como parámetro y devuelve el par $(u, v) = f(p)$ con las coords. de textura de p).

Subsección 6.3

Asignación explícita de coordenadas de textura.

Ejemplo de asignación explícita.

Esto es posible en objetos sencillos como este cubo construido con triángulos:

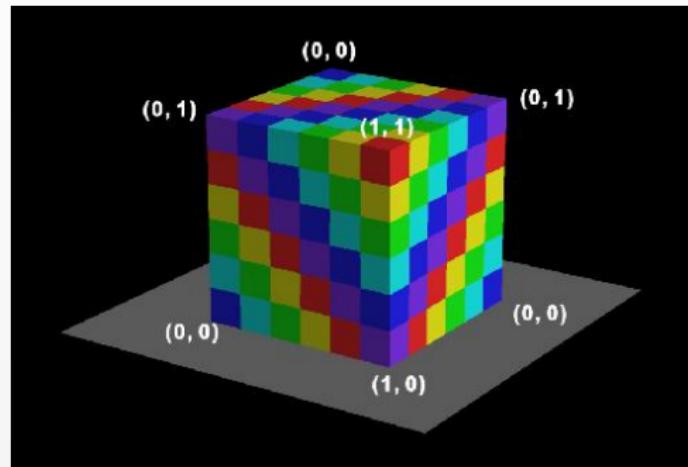
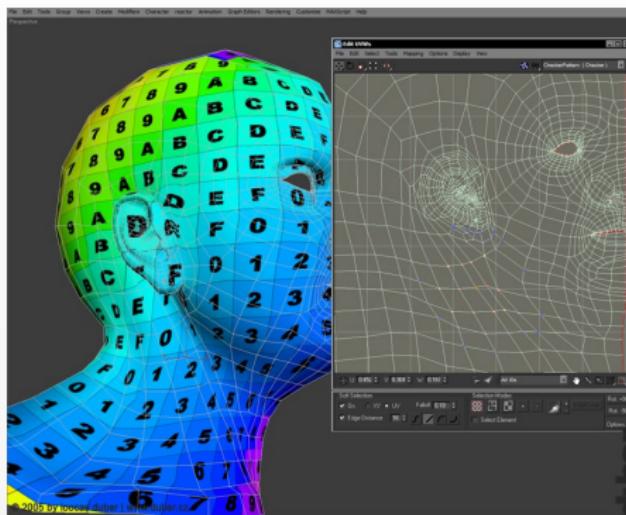


Imagen de Rosalee Wolfe, en [Teaching Texture Mapping Visually](#)
de SIGGRAPH 97 Education Slide Set.

Ejemplo de uso de herramientas CAD.

En objetos complejos es necesario el uso de este tipo de herramientas:



Imágenes de Lukáš Duběda (izquierda) y Mayan Escalante (derecha).

Subsección 6.4

Asignación procedural de coordenadas de textura.

Tipos de asignación procedural

Hay dos opciones:

- ▶ **Asignación procedural a vértices:** se invoca `CoordText (\mathbf{v}_i)` para calcular las coordenadas de textura en cada vértice \mathbf{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 (\mathbf{p})` cada vez que sea necesario evaluar el MIL en un punto de la superficie \mathbf{p} .
 - ▶ Permite exactitud incluso aunque f sea no lineal.
 - ▶ En OpenGL, esto requiere programación del cauce gráfico, invocando a `CoordText` en cada pixel desde el *fragment shader*.

Funciones para asignación procedural:

Los tipos de funciones f más frecuentes son:

- ▶ **Funciones lineales** de la posición (proyección en un plano): el punto $\mathbf{p} = (x, y, z)$ 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.

Otras opciones para asignación procedural

Otras opciones (no lineales) son estas dos:

- ▶ **Coordenadas polares** (proyección en una esfera): el punto \mathbf{p} se expresa en coordenadas polares como una terna (α, β, r) , los valores u y v se obtienen de α y β .
- ▶ **Coordenadas cilíndricas** (proyección en un cilindro): el punto \mathbf{p} se expresa en coordenadas cilíndricas como una terna (α, y, r) , los valores u y v se obtienen de α e y .

Es muy complicado usarlas con asignación a vértices (α 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 lineales (proyección).

En este caso el punto $\mathbf{p} = (x, y, z)$ 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 (\mathbf{q}) y por dos vectores libres (\mathbf{e}_u y \mathbf{e}_v , de longitud unidad y perpendiculares entre sí). En estas condiciones:

$$u = f_u(\mathbf{p}) = (\mathbf{p} - \mathbf{q}) \cdot \mathbf{e}_u \quad v = f_v(\mathbf{p}) = (\mathbf{p} - \mathbf{q}) \cdot \mathbf{e}_v$$

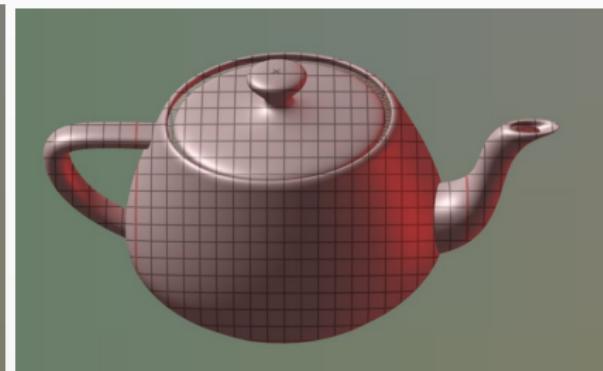
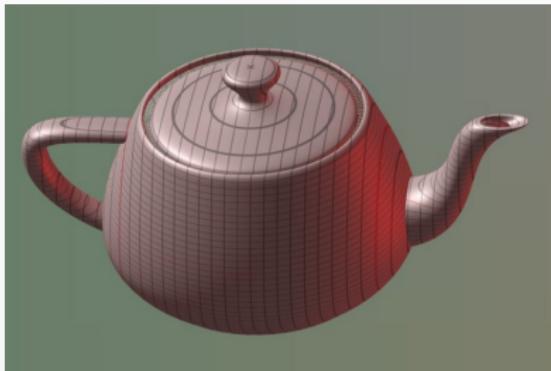
como casos particulares, y a modo de ejemplo, podemos hacer \mathbf{q} igual al origen $(0, 0, 0)$, $\mathbf{e}_u = \mathbf{x} = (1, 0, 0)$ y $\mathbf{e}_v = \mathbf{y} = (0, 1, 0)$, y en este caso es una proyección paralela el eje Z, sobre el plano XY (descarta la Z)

$$u = x = \mathbf{p} \cdot \mathbf{x} = (x, y, z) \cdot (1, 0, 0)$$

$$v = y = \mathbf{p} \cdot \mathbf{y} = (x, y, z) \cdot (0, 1, 0)$$

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).

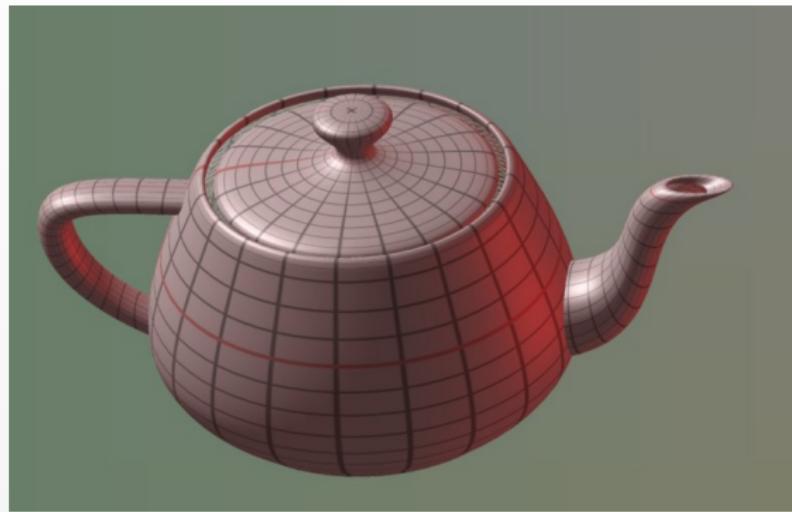
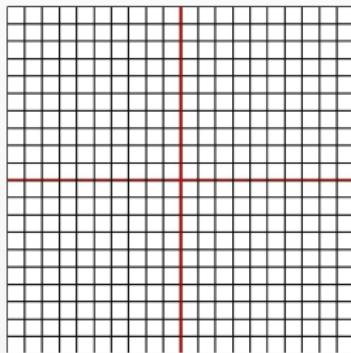
Coordenadas paramétricas.

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

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

Ejemplo de coordenadas paramétricas

Vemos un ejemplo de textura (izq.) y su aplicación a la tetera (der.)



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

Coordenadas esféricas

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

- ▶ Equivale a una proyección radial en una esfera.
- ▶ Las coordenadas (α, β, 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 α en el rango $[-\pi, \pi]$ y β en el rango $[-\pi/2, \pi/2]$. 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.

Ejemplo de coordenadas esféricas

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

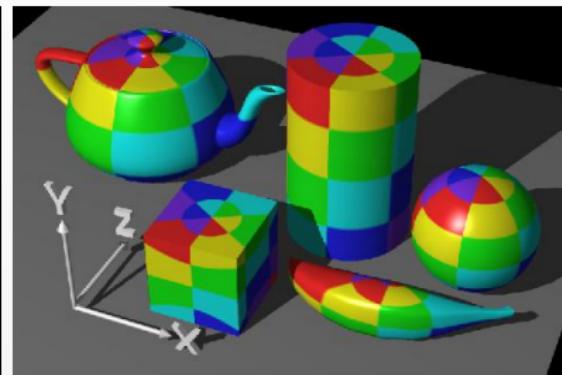
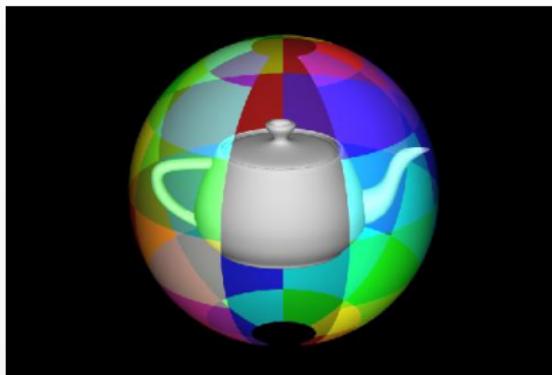


Imagen de Rosalee Wolfe, en [Teaching Texture Mapping Visually](#) de SIGGRAPH 97 Education Slide Set.

Coordenadas cilíndricas

Se basa en usar las coordenadas polares (ángulo y altura) del punto \mathbf{p}

- ▶ Equivale a una proyección radial en un cilindro (cuyo eje es usualmente un eje vertical central al objeto).
- ▶ Las coordenadas (α, 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 α 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 cilíndricas

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

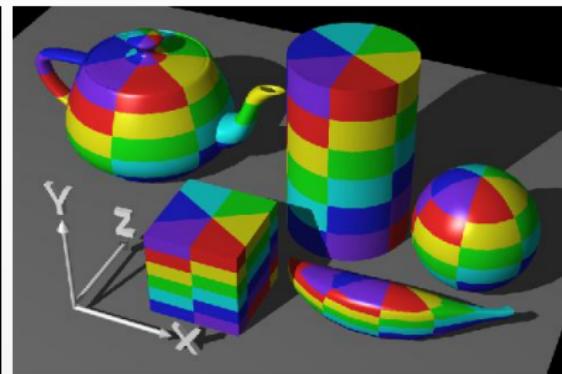


Imagen de Rosalee Wolfe, en [Teaching Texture Mapping Visually](#)
de SIGGRAPH 97 Education Slide Set.

Subsección 6.5

Consulta de texels en texturas de imagen.

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)

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) .

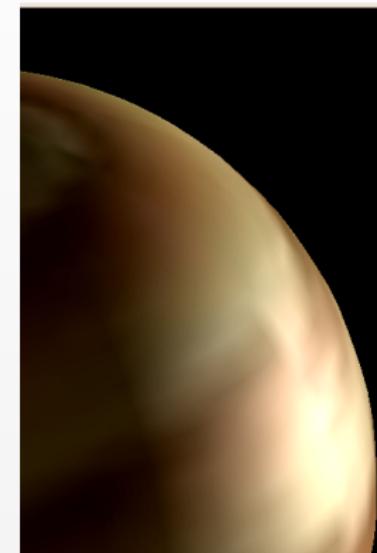
Las diferencias entre ambos métodos son visibles cuando la proyección en la ventana de un texel ocupa muchos pixels.

Interpolación bilineal

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



más cercano



interpolación bilineal

Sección 7

Texturas en OpenGL.

- 7.1. Activación y desactivación. Texturas en la evaluación del MIL.
- 7.2. Carga de texturas en el sistema gráfico.
- 7.3. Configuración de texturas.
- 7.4. Asignación explícita de coordenadas de textura.
- 7.5. Representación de texturas.

Subsección 7.1

Activación y desactivación. Texturas en la evaluación del MIL.

Activación y desactivación

Las ordenes **glEnable** y **glDisable** se pueden usar para activar o desactivar toda la funcionalidad de OpenGL relacionada con las texturas, que se encuentra inicialmente desactivada:

```
glEnable( GL_TEXTURE_2D ) ; // habilita texturas  
glDisable( GL_TEXTURE_2D ) ; // deshabilita texturas
```

Cuando se habilitan las texturas hay una textura activa en cada momento, que se consulta cada vez que un polígono se proyecta en un pixel, antes de calcular el color de dicho pixel:

- ▶ con iluminación desactivada, el color de la textura sustituye al especificado con **glColor**
- ▶ con iluminación activada, el color de la textura sustituye a las reflectividades del material (usualmente a la difusa y la ambiental).

todas las operaciones de texturas requieren que esta funcionalidad esté activada.

Subsección 7.2

Carga de texturas en el sistema gráfico.

Identificadores de textura. Creación.

OpenGL puede gestionar más de una textura a la vez. Para diferenciarlas usa un valor entero único para cada una de ellas, que se denomina **identificador de textura** (*texture name*) (de tipo **GLuint**).

- ▶ Para crear o generar un nuevo identificador de textura único (distinto de cualquiera ya existente) usamos:

```
GLuint idTex ;  
// hace idTex igual a un nuevo identificador  
glGenTextures( 1, &idTex );
```

- ▶ Para crear n nuevos identificadores de textura (en un array de **GLuint**), hacemos:

```
GLuint arrIdTex[n] ; // n es una constante entera (n > 0)  
// crea n nuevos identificadores en arrIdTex  
glGenTextures( n, arrIdTex );
```

Textura activa

En el estado interno de OpenGL hay en cada momento un identificador de textura activa:

- ▶ Cualquier operación de visualización de primitivas usará la textura asociada a dicho identificador.
- ▶ Cualquier operación de configuración de la funcionalidad de texturas se referirá a dicha textura activa.

Para cambiar el identificador de textura activa podemos hacer:

```
// activa textura con identificador 'idTex' :  
glBindTexture( GL_TEXTURE_2D, idTex );
```

Alojamiento en RAM de imágenes de textura

Antes de usar una textura en OpenGL (de tamaño $n_x \times n_y$), es necesario alojar en la memoria RAM una matriz con los colores de sus texels:

- ▶ Cada texel se representa (usualmente) con tres bytes (enteros sin signo entre 0 y 255), que codifican la proporción de rojo, verde y azul, respecto al valor máximo (255).
- ▶ Los tres bytes de cada texel se almacenan contiguos, usualmente en orden RGB.
- ▶ Los $3n_x$ bytes de cada fila de texels se almacenan contiguos, de izquierda a derecha.
- ▶ Las n_y filas se almacenan contiguas, desde abajo hacia arriba.
- ▶ Se conoce la dirección de memoria del primer byte, que llamamos **texels** (es un puntero de tipo **void ***)

con este esquema la imagen ocupará, lógicamente, $3n_xn_y$ bytes consecutivos en memoria.

Especificación de los texels de la imagen de textura

En cualquier momento podemos especificar cual será la imagen de textura asociada al identificador de textura activa, con **glTexImage2D**:

```
glTexImage2D( GL_TEXTURE_2D,  
    0,                  // nivel de mipmap (para imágenes multiresolución)  
    GL_RGB,             // formato interno  
    ancho,              // núm. de columnas (potencia de dos: 2n) (GLsizei)  
    alto,               // núm de filas (potencia de dos: 2m) (GLsizei)  
    0,                  // tamaño del borde, usualmente es 0  
    GL_RGB,             // formato y orden de los texels en RAM  
    GL_UNSIGNED_BYTE,  
                    // tipo de cada componente de cada texel  
    texels              // puntero a los bytes con texels (void *)  
);
```

Al llamar a esta función, OpenGL leerá los bytes de la RAM y los copiará en otra memoria (típicamente la memoria de vídeo o de la GPU, en un formato interno).

Especificación de la imagen con GLU

Si es posible usar GLU, hay una alternativa preferible a **glTexImage2D** que no requiere imágenes de tamaño potencia de dos, y que además genera automáticamente versiones a múltiples resoluciones (*mip-maps*)

```
gluBuild2DMipmaps( GL_TEXTURE_2D,  
    GL_RGB,      // formato interno  
    ancho,       // núm. de columnas (arbitrario) (GLsizei)  
    alto,        // núm de filas (arbitrario) (GLsizei)  
    GL_RGB,      // formato y orden de los texels en RAM  
    GL_UNSIGNED_BYTE,  
                // tipo de cada componente de cada texel  
    texels       // puntero a los bytes con texels (void *)  
);
```

(esta función hace copias escaladas de la imagen para adaptarla a tamaños potencias de dos a distintas resoluciones)

Subsección 7.3

Configuración de texturas.

Configuración de texturas.

En el estado de OpenGL, hay un conjunto de atributos o parámetros que determinan la apariencia de las texturas. Estos parámetros determinan, entre otros aspectos:

- ▶ Como se usa el color de los texels en el MIL con ilum. activada (que reflectividades del material son obtenidas de la textura activa)
- ▶ Como se selecciona el texel o texels a partir de una coords. de textura (más cercano o interpolación).
- ▶ Como se selecciona el texel cuando las coords. de textura no están en el rango $[0,1]$ (replicado o truncamiento).
- ▶ Si se asignan explícitamente coordenadas o bien OpenGL las genera proceduralmente, y en este caso como se hace.

Texturas e iluminación.

La función **glLightModel** puede usarse para determinar como los colores de la textura afectan al MIL, cuando la iluminación y la texturas están activadas. Hay dos opciones:

- ▶ El color de la textura se use en lugar de todas las reflectividades del material, M_A, M_D y M_S ,

```
glLightModeli( GL_LIGHT_MODEL_COLOR_CONTROL,  
               GL_SINGLE_COLOR ) ; // inicialmente activado
```

- ▶ El color de la textura se usa en lugar de M_A y M_D , pero no M_S , esto permite brillos especulares de color blanco cuando hay texturas de color.

```
glLightModeli( GL_LIGHT_MODEL_COLOR_CONTROL,  
               GL_SEPARATE_SPECULAR_COLOR ) ;
```

Selección de texels

OpenGL permite especificar como se seleccionarán los texels en cada consulta posterior de la textura activa, cuando el pixel actual es igual o más pequeño que el texel que se proyecta en él:

- seleccionar el texel con centro más cercano al centro del pixel:

```
glTexParameteri( GL_TEXTURE_2D,  
                  GL_TEXTURE_MAG_FILTER, GL_NEAREST );
```

- hacer interpolación bilineal entre los cuatro texels con centros más cercanos al centro del pixel:

```
glTexParameteri( GL_TEXTURE_2D,  
                  GL_TEXTURE_MAG_FILTER, GL_LINEAR );
```

(la opción inicialmente activada es la segunda de ellas).

Generación procedural de coordenadas de textura.

OpenGL puede generar proceduralmente las coordenadas de textura (s, t) en cada pixel, cada vez que se consulte la textura, a partir de las coordenadas de objeto (o de mundo) del punto de la superficie \mathbf{p} que se proyecta en el centro del pixel.

Para habilitar esta posibilidad, hacemos:

```
glEnable( GL_TEXTURE_GEN_S ); // desactivado inicialmente  
glEnable( GL_TEXTURE_GEN_T ); // desactivado inicialmente
```

Igualmente podemos usar **glDisable** para desactivar.

Es necesario hacer ambas llamadas ya que OpenGL permite generar únicamente la coordenada s o únicamente la t (normalmente se generan ambas o ninguna).

Tipo de función para generación procedural.

Con OpenGL podemos usar funciones lineales (proyección en un plano) para la generación automática de coords. de textura.

- ▶ Para hacerlo en **coordenadas de objeto** (antes de modelview), usaríamos:

```
glTexGeni( GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR ) ;  
glTexGeni( GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR ) ;
```

- ▶ Para **coordenadas de ojo** (después de modelview), haríamos:

```
glTexGeni( GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR ) ;  
glTexGeni( GL_T, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR ) ;
```

(la opción inicial es usar coordenadas de objeto).

Parámetros de la función lineal

La función lineal que se usa para calcular (s, t) a partir de (x, y, z) es de la forma:

$$\begin{aligned}s &= a_s x + b_s y + c_s z + d_s \\t &= a_t x + b_t y + c_t z + d_t\end{aligned}$$

Si queremos proyectar en un plano que pasa por \mathbf{q} y contiene los vectores $\mathbf{e}_s = (s_x, s_y, s_z)$ y $\mathbf{e}_t = (t_x, t_y, t_z)$ (de longitud unidad y perpendiculares entre sí), entonces debemos de hacer:

$$\begin{array}{lll}a_s &= s_x & a_t &= t_x \\b_s &= s_y & b_t &= t_y \\c_s &= s_z & c_t &= t_z \\d_s &= -\mathbf{q} \cdot \mathbf{e}_s & d_t &= -\mathbf{q} \cdot \mathbf{e}_t\end{array}$$

Especificación de coeficientes

Para especificar los coeficientes de las funciones lineales, podemos usar **glTexGenfv**. OpenGL guarda en su estado dos juegos de parámetros, cada uno usado para un modo de generación de coordenadas.

```
GLfloat coefsS[4] = { as, bs, cs, ds } ,  
    coefsT[4] = { at, bt, ct, dt } ;  
  
// para el modo de coords. de objeto:  
glTexGenfv( GL_S, GL_OBJECT_PLANE, coefsS ) ;  
glTexGenfv( GL_T, GL_OBJECT_PLANE, coefsT ) ;  
  
// para el modo de coords. del ojo:  
glTexGenfv( GL_S, GL_EYE_PLANE, coefsS ) ;  
glTexGenfv( GL_T, GL_EYE_PLANE, coefsT ) ;
```

Subsección 7.4

Asignación explícita de coordenadas de textura.

Asignación explícita con *begin/end*

Cuando la generación automática de coords. de textura está desactivada, es necesario especificar a OpenGL las coords. de textura de cada vértice. En el estado interno hay un par (s, t) que se asignan a cada vértice que se envia con **glVertex**.

Para cambiar el par (s, t) actual podemos usar **glTexCoord2f**:

```
glBegin(GL_TRIANGLES);
    glTexCoord2f( s0, t0 ); glVertex3f( x0, y0, z0 );
    glTexCoord2f( s1, t1 ); glVertex3f( x1, y1, z1 );
    glTexCoord2f( s2, t2 ); glVertex3f( x2, y2, z2 );

    // .... otros triángulos ...
glEnd();
```

La tabla de coordenadas de textura

En las mallas indexadas se puede incluir un puntero a una tabla de coordenadas de textura de vértices (una entrada por vértice, dos flotantes por entrada, de tipo **Tupla2f**).

```
class MallaInd
{
    ...
    // variables de instancia relacc. con coords. de textura (cc.tt.)
    std::vector<Tupla2f> cctt;           // tabla de coords. de textura.
    Natural tam_cctt;                   // tamaño en bytes de la tabla
    GLuint id_vbo_cctt;                 // identificador de VBO de la tabla
};
```

Las clases derivadas de **MallaInd** definen constructores concretos. Algunas de ellas pueden crear la tabla **cctt**, en el resto de los casos, tendrá 0 elementos (la malla no tiene cc.tt.).

Asignación explícita con Vertex Arrays

Si los vértices se envían con `glDrawElements` (o `glDrawArrays`), es necesario indicar antes donde está la tabla de coordenadas de textura, esto se hace con `glTexCoordPointer`:

```
void MallaInd::visualizarDE_NT( ) // visu. con normales y cc.tt.  
{  
    glVertexPointer( 3, GL_FLOAT, 0, ver.data() );  
    glTexCoordPointer( 2, GL_FLOAT, 0, cctt.data() );  
    glNormalPointer( GL_FLOAT, 0, nor_ver.data() );  
  
    glEnableClientState( GL_VERTEX_ARRAY );  
    glEnableClientState( GL_NORMAL_ARRAY );  
    glEnableClientState( GL_TEXTURE_COORD_ARRAY );  
  
    glDrawElements(GL_TRIANGLES, num_tri, GL_UNSIGNED_INT, tri.data());  
  
    glDisableClientState( GL_VERTEX_ARRAY );  
    glDisableClientState( GL_NORMAL_ARRAY );  
    glDisableClientState( GL_TEXTURE_COORD_ARRAY );  
}
```

Creación de VBOs con coordenadas de textura

También es posible alojar la tabla de coordenadas de textura en un VBO en la memoria de la GPU, de forma similar a como hacíamos con las normales y los colores de vértices (en la malla se almacena el identificador de vbo)

```
// calcular tamaño de la tabla de cc.tt.  
tam_cctt = sizeof(Real)*2L*num_ver ;  
// enviar datos a las GPU, obtener y guardar ident. de VBO  
id_vbo_cctt = VBO_Crear( GL_ARRAY_BUFFER,  
                           tam_cctt, cctt.data() ) ;
```

este código requiere guardar en la malla:

- ▶ **id_vbo_cctt**: identificador del VBOs con las coordenadas de textura, de tipo **GLuint**.
- ▶ **tam_cctt**: tamaño de la tabla de coordenadas de textura, en bytes.

Visualización de VBOs con coordenadas de textura

Para visualizar las c.t. y las normales en los VBOs, haríamos:

```
void MallaInd::visualizarVBOs_NT( ) // vis. normales y cc.tt. en VBOs
{
    // activar VBO de coordenadas de normales
    glBindBuffer( GL_ARRAY_BUFFER, id_vbo_nor_ver );
    glNormalPointer( GL_FLOAT, 0, 0 ); // (0 == offset en vbo)
    glEnableClientState( GL_NORMAL_ARRAY );

    // activar VBO de coordenadas de textura
    glBindBuffer( GL_ARRAY_BUFFER, id_vbo_cctt );
    glTexCoordPointer( 2, GL_FLOAT, 0, 0 ); // (0 == offset en vbo)
    glEnableClientState( GL_TEXTURE_COORD_ARRAY );

    // visualizar (el mismo método ya visto)
    visualizarVBOs();
    // desactivar punteros a tablas
    glDisableClientState( GL_NORMAL_ARRAY );
    glDisableClientState( GL_TEXTURE_COORD_ARRAY );
}
```

Subsección 7.5

Representación de texturas.

La clase para texturas

La textura forma parte del modelo de material. Para implementar esto podemos usar una clase, de nombre **Textura**,

```
class Textura
{
public:
    GLuint      idText; // identificador OpenGL de la textura
    jpg::Imagen * img; // puntero a objeto imagen con los texels en RAM
    unsigned     mgct; // modo generación cc.tt.: 0 =desactivado, 1 =objeto, 2 =mundo.
    float       cs[4], // coeficientes (S) para generación de cc.tt. (si mgct≠ 0)
                ct[4]; // coeficientes (T) para generación de cc.tt. (si mgct≠ 0)

    // activar la textura: habilita texturas y activa esta en concreto
    void activar( ContextoVis & cv );
};
```

La configuración de la textura debe hacerse una vez al crearla. En principio no necesita constructor, pues estas instancias se pueden inicializar en los constructores de las clases concretas derivadas de la clase **MaterialEstandar**.

Sección 8

Materiales en el grafo de escena

- 8.1. Modelo de aspecto: definición y ejemplos
- 8.2. Implementación de materiales en el grafo

Subsección 8.1

Modelo de aspecto: definición y ejemplos

Modelo de aspecto en los grafos de escena

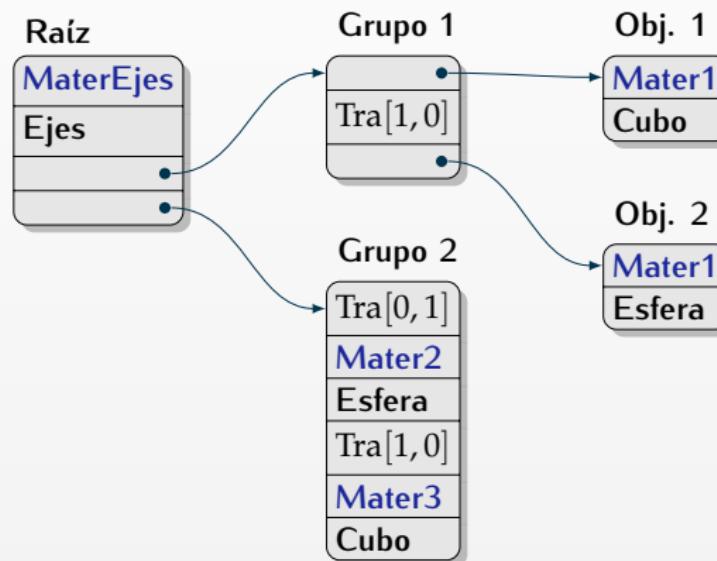
Los objetos de tipo material constituyen un **modelo de aspecto** de los distintos objetos de una escena, al igual que las mallas constituyen **modelos geométricos** de los mismos objetos.

En los grafo de escena vamos a incorporar modelos de aspecto al mismo nivel que los geométricos, es decir,

- ▶ Contemplaremos un nuevo tipo de entrada en los nodos: las **entradas de tipo material**. Contiene una referencia a un material concreto a usar para visualizar una parte de los nodos del grafo.
- ▶ Un material en una entrada de un nodo afecta a todas las entradas posteriores de dicho nodo, hasta el final del nodo o bien hasta otra entrada posterior del nodo también de tipo material.
- ▶ Por tanto, toda entrada está afectada del primer material encontrado en el camino desde esa entrada hasta la primera entrada del nodo raíz (si no hay ninguno, se usaría uno por defecto).

Ejemplos de modelo de aspecto

Disponemos de las primitivas **Cubo**, **Esfera** y **Ejes**, y cuatro materiales posibles (se muestran en azul):



La pila de materiales

Para visualizar un grafo es cómodo disponer en el contexto de visualización una pila de materiales (parecido a lo que ocurre con la matriz *modelview*)

- ▶ Permite activar materiales que solo afectan a un nodo o sub-árbol del grafo de escena (o a una parte del mismo). La pila gestiona punteros a los materiales.
- ▶ La pila contiene un puntero al material activo en cada momento, y una lista o vector de materiales guardados.
- ▶ Al inicio de la visualización de un nodo, se guarda en la pila un puntero al material activo actual (push).
- ▶ Durante la visualización de un nodo, el material activo puede ir cambiando.
- ▶ Al final de la visualización de un nodo, se elimina el tope de la pila y se reactiva de nuevo el material anterior.

Implementación de la pila

La pila de materiales puede declararse con una clase así:

```
class PilaMateriales
{
    private:
        Material * actual ; // material actualmente activado (NULL al inicio)
        std::vector<Material *> pila ; // materiales activados antes

    public:
        PilaMateriales() ; // pone actual a NULL, pila está vacía
        void push() ; // añade una copia de actual en el tope de pila
        void pop() ; // copia tope de pila en actual, elimina el tope, activa actual
        void activarMaterial( Material * material ) ;
                            // activa material y lo copia en actual
} ;
```

- ▶ En los métodos **activarMaterial** y **pop**, si el material a activar es igual al activado, no se hace nada (por eficiencia).
- ▶ En el método **activarMaterial**, si el material es NULL, no se hace nada.

Uso de una pila de materiales

La pila de materiales se puede incluir como una instancia (**pilaMateriales**) dentro del contexto de visualización, de forma que, por ejemplo, para visualizar un par de objetos (**obj1** y **obj2**) con un material **mat**, podemos hacer:

```
void VisualizarEjemploMat( ContextoVis & cv )
{
    cv.pilaMateriales.push();
    cv.pilaMateriales.activar( mat );
    obj1->visualizarGL( cv );
    obj2->visualizarGL( cv );
    cv.pilaMateriales.pop();
}
```

- ▶ al hacer **push/pop**, el material activo a la entrada está de nuevo activo a la salida.
- ▶ los objetos **obj1** y **obj2** pueden consistir únicamente de geometría, (p.ej., pueden ser mallas indexadas), o bien pueden ser objetos complejos, posiblemente con otros materiales para algunas de sus partes.

Subsección 8.2

Implementación de materiales en el grafo

Entradas de tipo *material*

Ahora las entradas de los nodos del tipo grafo de escena pueden contener un puntero a un material cualquiera:

```
struct EntradaNGE
{
    unsigned char tipoE ; // 0 => objeto, 1 => transformacion, 2 => material
    union
    {
        Objeto3D * objeto ; // ptr. a un objeto
        Matriz4f * matriz ; // ptr. a matriz 4x4 transf. (propriet.)
        Material * material ; // ptr. a material
    } ;
    // constructores (uno por tipo)
    EntradaNGE( Objeto3D * pObjeto ) ; // (copia únicamente el puntero)
    EntradaNGE( const Matriz4f & pMatriz ) ; // (crea copia de la matriz)
    EntradaNGE( Material * pMaterial ) ; // (copia únicamente el puntero)
};
```

Habrá una nueva versión del método **agregar** de los nodos:

```
class NodoGrafoEscena : public Objeto3D
{
    .....
    void agregar( Material * pMaterial ) ; // añadir material al final
};
```

Procesamiento de nodos con materiales

Hay que actualizar el método **visualizarGL** de la clase **NodoGrafoEscena**:

- ▶ Al inicio, hay que hacer **push** de la pila de materiales:

```
cv.pilaMateriales.push();
```

- ▶ En el bucle, al encontrar una entrada de tipo material, hay que activarlo:

```
if ( entradas[i].tipoE == 2 )           // si entrada tipo material:  
    cv.pilaMateriales.activarMaterial( entrada[i].material );
```

- ▶ Al finalizar, reactivamos el material activo originalmente, y eliminamos el tope de la pila:

```
cv.pilaMateriales.pop();
```

Sección 9

Visualización con el cauce gráfico programable.

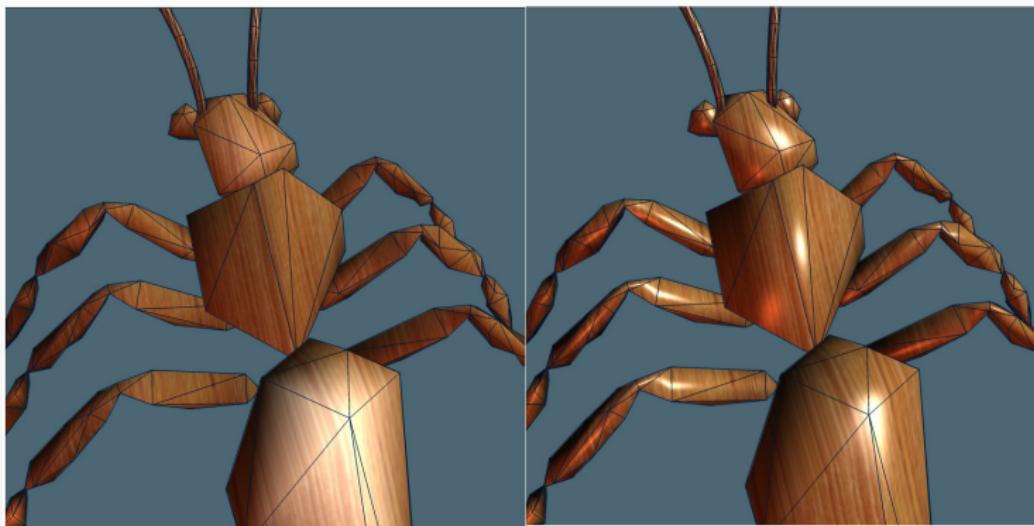
- 9.1. Introducción.
- 9.2. Sombreado de pixels (*fragment shader*)
- 9.3. Atributos de vértice genéricos

Subsección 9.1

Introducción.

Introducción.

En OpenGL, la única forma de evaluar el MIL en cada pixel es usar un programa (*vertex + fragment shader*) para el cauce gráfico, distinto del cauce de la funcionalidad fija (la calidad es mucho mayor)



Uso de GLSL

En esta sección veremos como emular la funcionalidad fija (con una configuración sencilla) usando el cauce programable con GLSL:

- ▶ Usaremos GLSL versión 1.3 (Agosto 2008, simultáneo con OpenGL 3.0)
- ▶ En general, se intenta evitar funcionalidad obsoleta en esa versión.
- ▶ Se usa la forma tradicional de envío de parámetros (posicion, normal, coordenadas de textura) al vertex shader.
- ▶ Esta forma es obsoleta en 1.3 y no está disponible en 1.4 y posteriores.
- ▶ Esta forma es compatible con el envío tradicional de arrays de valores (visto en teoría: 'glVertexPointer', 'glTexCoordCoordPointer', etc...)

Parámetros de entrada a los vertex shader

Un *vertex shader* puede recibir dos tipos de parámetros de entrada:

► **Parámetros uniform:**

- Tienen el mismo valor para todos los vértices en una primitiva.
- Se pueden fijar desde la aplicación con la familia de funciones `glUniform`.

► **Atributos de vértice:**

- Tienen valores potencialmente distintos para cada vértice en una primitiva.
- Hay de dos tipos:
 - atributos *estándard* (posición, color, normal, coordenadas de textura): están **implicitamente declarados** en el shader (hasta GLSL 1.3).
 - definidos por el programador: deben declararse en el shader.

Cada uniform o atributo tiene asociado un identificador (*location*) es un valor entero que lo identifica en el contexto de un shader y que **debe ser conocido en la aplicación** (hasta OpenGL 3.0 no es estrictamente necesario conocerlo si es un atributo estándar).

Envío de atributos de vértices al vertex shader

Desde la aplicación se deben especificar los atributos de vértice para cada vértice, ya sea en modo inmediato (individualmente o con arrays) o en modo diferido.

- ▶ Se envían desde la aplicación (desde OpenGL 2.0) con:
 - ▶ **glVertexAttribPointer**: especifica dirección de un array de valores para un atributo cualquiera.
 - ▶ **glVertexAttrib**: cambia valor actual de un atributo cualquiera para los siguientes vértices a enviar
- ▶ Los atributos *estándar* se pueden enviar (solo hasta OpenGL 3.0) con:
 - ▶ **glNormalPointer**, **glTexCoordPointer**, etc.. : especifica dirección de un array de valores para un atributo *estándar*.
 - ▶ **glNormal**, **glTexCoord**, etc..: cambia valor actual de un atributo *estándar* para los siguientes vértices.

Vertex shader. Parámetros de salida

Cada ejecución del vertex shader debe escribir uno o varios parámetros de salida o resultados:

- ▶ Tras la ejecución del vertex shader, quedan ligados al vértice en cuestión.
- ▶ Se interpolan durante la rasterización, y se entregan interpolados al *fragment shader* en cada pixel.

Hay dos tipos de parámetros de salida:

- ▶ Declarados explícitamente a conveniencia del programador, con **out** (es equivalente a **varying**, que está disponible solo hasta GLSL 4.2)
- ▶ Declarados implicitamente (predefinidos): existe **gl_Position** (posición del vértice), y también **gl_Color**.

Vertex shader: declaraciones.

```
// directiva que indica que el código corresponde a GLSL ver 1.3
#version 130

// parámetros de entrada uniform (constantes en cada primitiva)
uniform mat4 matrizModelado ;      // matriz de modelado actual
uniform mat4 matrizVista ;         // matriz de vista (mundo -> cámara)
uniform mat4 matrizProyeccion ;    // matriz projection
uniform int tipoGCT ;             // tipo gen.cc.tt.(0=desact.,1=obj.,2=cám.)

// parámetros de entradas predefinidos en GLSL (solo disponibles hasta GLSL 1.30)
// vec4 gl_Vertex coordenadas del vértice en el m.c. de objeto
// vec3 gl_Normal normal del vértice en el m.c. de objeto
// vec3 gl_TexCoord[0] coordenadas de textura (unidad 0)

// valores calculados como salida (todas en coordenadas de cámara, EC)
out vec4 normalPunto;              // normal del vértice (w=0)
out vec4 posPunto ;                // posición del vértice
out vec3 vecObservador;           // vector del vértice hacia el observador
out vec2 coordsText ;              // coordenadas de textura vért.

....
```

Vertex Shader: función principal (main)

```
....  
void main()  
{  
    // calcular la matriz modelview  
    mat4 matrizModelView = matrizVista * matrizModelado ;  
  
    // calcular normal, posición y vector al obs. (en el m.c. de cámara)  
    // son variables de salida (out) que se interpolan durante rasterización  
    normalPunto     = matrizModelView * vec4(gl_Normal, 0.0) ;  
    posPunto       = matrizModelView * gl_Vertex ;  
    vecObservador = -posPunto.xyz ;  
  
    // calcular coordenadas de textura del vértice (tamb. se interpolan)  
    if ( tipoGCT == 0 )      coordsText = gl_TexCoord[0].st ;  
    else if ( tipoGCT == 1 ) coordsText = gl_Vertex.xy ;  
    else                      coordsText = posPunto.xy ;  
  
    // calcular posición de vértice, en coords de recortado (requerido)  
    gl_Position   = (matrizProyeccion * modelView) * gl_Vertex;  
}
```

Subsección 9.2

Sombreado de pixels (*fragment shader*)

Simplificaciones

Veremos un shader sencillo que calcula el color de los pixels, y que **concuerda exactamente** con el cauce fijo, asumiendo:

- ▶ Iluminación habilitada (opción *local viewer* activada, opción *color material* desactivada, opción *separate specular color* activada).
- ▶ Exactamente **dos fuentes de luz direccionales**, cada una con un único color para todas las componentes.
- ▶ Se considera un **material** con cuatro reflectividades y el exponente.
- ▶ Texturas desactivadas o bien una **textura** activa (en la unidad 0).
- ▶ Respecto a la **generación automática de coordenadas de textura**, hay dos opciones:
 - ▶ activada: se pueden usar coords. de objeto o de la cámara.
 - ▶ desactivada: se usan las coords. asignadas explícitamente por la aplicación a los vértices.

Parámetros del *fragment shader*

El *fragment shader* puede tener también dos tipos de parámetros de entrada:

- ▶ **Parámetros uniform:** su valor es igual en todos los pixels de una primitiva, se envían desde la aplicación con `glUniform`.
 - ▶ cada uno puede ser o no el mismo (mismo nombre y tipo) que un **uniform** del *vertex shader*.
- ▶ **Parámetros in:** valores interpolados a partir de los parámetros **out** generados desde un *vertex shader*.
 - ▶ sus nombres y tipos deben coincidir uno a uno con los parámetros **out** del *vertex shader*.

A continuación veremos el código GLSL del fragment shader

- ▶ Incluye dos funciones GLSL (**FGE** y **EvaluarmIL**), con sintaxis semejante a la de C/C++.

Fragment shader: declaraciones.

```
#version 130          // pedir glsl ver. 1.3
const int nf = 2 ; // numero de fuentes de luz

// parametros de entrada (uniform)
uniform vec3      vecFue[nf]; // vector hacia la fuente de luz (w=0)
uniform vec3      colFue[nf]; // colores de cada fuentes (único)
uniform vec3      colMat[4]; // colores material (0=emi,1=amb,2=dif,3=esp)
uniform float     expPse;    // exponente de la componente pseudo-especular
uniform sampler2D textura;   // textura
uniform int       usarText;  // 1 = usar 'textura', 0 = no usarla

// parametros de entrada (interpolados), todos en coords. de cámara
in vec4 posPunto ;    // posicion del punto central al pixel
in vec3 normalPunto; // normal del punto anterior
in vec3 vecObservador; // vector de punto hacia el observador
in vec2 coordsText ; // coordenadas de textura del punto
....
```

Factor geométrico de la comp. pseudo-especular.

Esta función calcula el valor $(\mathbf{h} \cdot \mathbf{n})^e$, donde \mathbf{h} es el vector *halfway* (es decir: $\mathbf{v} + \mathbf{l}$, normalizado). Es la fórmula de Blinn-Phong.

```
// Función FGE: tiene estos parámetros:  
//     e = exponente de brillo  
//     n = vector normal (normalizado)  
//     v = vector hacia el observador (normalizado)  
//     l = vector hacia la fuente (normalizado)  
  
float FGE( float e, vec3 n, vec3 v, vec3 l )  
{  
    return pow(max(dot(normalize(v+l),n),0.0),e) ; // Blinn-Phong  
    // return pow(max(dot(2.0*dot(n,l)*n-l,v),0.0),0.25*e) ; // Phong  
}
```

la opción comentada evalua $(\mathbf{r} \cdot \mathbf{v})^{e/4}$, donde \mathbf{r} es el vector reflejado (simétrico de \mathbf{l}), es decir, es $2(\mathbf{n} \cdot \mathbf{l})\mathbf{n} - \mathbf{l}$. Es la fórmula original de Phong.

Evaluación del M.I.L. simplificado

```
// Función EvaluarMIL: parámetros:  
//   m = material: (0=emi,1=amb,2=dif,3=espec)  
//   t = color de la textura ((1,1,1) si no hay textura)  
//   e = exponente de brillo (componente pseudo-especular)  
//   n = normal (normalizada)  
//   v = vector hacia el observador (normalizado)  
//   l = vector hacia la fuente (normalizado)  
//   f = color de la fuente  
vec3 EvaluarMIL( vec3 m[4], vec3 t, float e,  
                      vec3 n, vec3 v, vec3 l, vec3 f )  
{  
    vec3 // calcular colores de: emisión, ambiental, difusa, especular  
    cEmis= m[0], cAmb= f*t*m[1], cDif= f*t*m[2], cEsp= f*m[3];  
    // calcular color resultado ('res'): float  
    vec3 res = cEmis + cAmb ; // sumar emision+ ambiente  
    float nl = dot(n,l) ; // coseno del ángulo entre n y l  
    if ( 0.0 < nl ) // si la normal está de cara a la fuente:  
        res += cDif*nl + cEsp*FGE(e,n,v,l) ; // sumar dif. y espec.  
    return res ;  
}
```

Fragment shader: función principal

```
void main()
{
    vec3
        norm    = normalize(normalPunto.xyz), // normal del punto
        vecobs  = normalize(vecObservador),   // vector hacia el observador
        colRes  = vec3(0.0,0.0,0.0),          // color resultado
        colText = vec3(1.0,1.0,1.0);         // color de la textura
    // si se debe usar la textura, consultarla
    if ( usarTextura != 0 )
        colText = texture( textura, coordsText ).rgb ;
    // para cada fuente de luz, sumar su contrib al MIL
    for( int i = 0 ; i < nf ; i++ )
    {
        vec3 vluz = normalize(vecFue[i]);
        colRes += EvaluarMIL( colMat, colText, expPse, norm,
                               vecobs, vluz, colFue[i] );
    }
    // escribir en 'gl_FragColor' para producir resultado (color del fragmento)
    gl_FragColor = vec4( colRes, 1.0 ) ;
}
```

Subsección 9.3

Atributos de vértice genéricos

Atributos de vértices

Un **atributo de vértice** es un parámetro de entrada a un *vertex shader*, como puede ser: posición, normal, color, coordenadas de textura (estándar), o bien otros como, p.ej., los vectores tangentes (no estándar, o genéricos)

- ▶ a partir de OpenGL 2.0 (2004), es **posible** asociar cualquier atributo (no estándar) a los vértices como parámetro de entrada a los *vertex shaders*.
- ▶ a partir de OpenGL 3.2 / GLSL 1.50 (2009), es **necesario** usar atributos de vértice genéricos para cualquier parámetro de entrada del *vertex shader* (no existe `gl_Position`, `gl_Normal`, `glVertexPointer`, `glEnableClientState`, etc...)

En esta sección veremos una versión moderna del *vertex shader*, que no usa atributos estándar, y se basa exclusivamente en atributos genéricos.

Localización de los atributos

Cada atributo genérico tiene asociados:

- ▶ Una **localización**: un número entero que lo identifica en la aplicación,
- ▶ Un **nombre**: un identificador en el fuente del *vertex shader* (en la aplicación es una cadena de caracteres).

En la aplicación se pueden asociar localizaciones a los atributos. A modo de ejemplo, podemos usar estas localizaciones y nombres:

- ▶ **Posición**: localización 0, nombre `av_posicion`
- ▶ **Normal**: localización 1, nombre `av_normal`
- ▶ **Coordenadas de textura**: localización 2, nombre `av_coordsText`

(posición y normal en coordenadas de objeto).

Asignación de localizaciones

En la aplicación, hacemos estas declaraciones globales para referirnos a las localizaciones con identificadores descriptivos:

```
const GLuint loc_posicion = 0 ,  
          loc_normal   = 1 ,  
          loc_coordsText = 2 ;
```

Para asociar las localizaciones a los nombres de atributos, se debe usar **glBindAttribLocation** antes de enlazar (después de compilar) los shaders (usando el identificador)

```
....  
glBindAttribLocation( idProg, loc_posicion, "av_posicion" );  
glBindAttribLocation( idProg, loc_normal,   "av_normal"  );  
glBindAttribLocation( idProg, loc_coordsText, "av_coordText" );  
  
glLinkProgram( idProg );  
...
```

Envío de tablas de atributos.

Cuando se usan atributos genéricos, se deben usar

- ▶ la función `glVertexAttribPointer`, en lugar de `glVertexPointer`, `glNormalPointer`, `glColorPointer`, `glNormalPointer`, `glTexCoordPointer`.
- ▶ la función `glEnableVertexAttribArray`, en lugar de `glEnableClientState`.
- ▶ la función `glDisableVertexAttribArray`, en lugar de `glDisableClientState`.

estas nuevas funciones tienen los mismos parámetros que las que sustituyen, y adicionalmente todas ellas necesitan la **localización** del atributo al que se refieren.

Vertex shader: declaraciones

```
#version 130 // se compila como GLSL versión 1.3

// parámetros de entrada (uniform: constantes en cada primitiva)
uniform mat4 matrizModelado ; // matriz de modelado actual
uniform mat4 matrizVista ; // matriz de vista (mundo -> cámara)
uniform mat4 matrizProyección; // matriz projection
uniform int tipoGCT ; // tipo gen.cc.t.(0=desact.,1=obj.,2=cám.)

// parámetros de entrada (atributos de vértice: distintos en cada vértice)
in vec3 av_posicion ; // coordenadas del vértice en el m.c. de objeto
in vec3 av_normal ; // normal del vértice en el m.c. de objeto
in vec2 av_coordText ; // coordenadas de textura

// parámetros de salida (todas en coordenadas de cámara, EC)
out vec3 normalPunto; // normal del vértice
out vec4 posPunto ; // posición del vértice
out vec3 vecObservador; // vector desde vért. hacia el observador (norm.)
out vec2 coordsText ; // coordenadas de textura del vértice
...
```

Vertex shader: función principal

```
void main()
{
    // calcular la matriz modelview
    mat4 modelView = matrizVista * matrizModelado ;

    // calcular normal, posición y vector al obs. (en el m.c. de cámara)
    posPunto      = modelView * vec4( av_posicion, 1.0 ) ;
    normalPunto   = (modelView* vec4( av_normal, 0.0 )).xyz ;
    vecObservador = -posPunto.xyz ;

    // calcular la coordenadas de textura
    if ( tipoGCT == 0 )      coordsText = av_coordText ;
    else if ( tipoGCT == 1 )  coordsText = av_posicion.xy ;
    else                      coordsText = posPunto.xy ;

    // calcular posición a usar en rasterización, en coords de rec.
    gl_Position= (matrizProyeccion*modelView)*vec4(av_posicion,1.0);
}
```

Fin de la presentación.