



UNIVERSIDAD  
DE GRANADA

# Informática Gráfica: Teoría. Tema 3. Visualización.

---

Carlos Ureña

2021-22

**Grado en Informática y Matemáticas**  
Dpt. Lenguajes y Sistemas Informáticos  
ETSI Informática y de Telecomunicación  
Universidad de Granada

## Teoría. Tema 3. Visualización.

### Índice.

1. Cauce gráfico y definición de la cámara.
2. Modelos de Iluminación, texturas y sombreado.
3. Iluminación y texturas con el cauce fijo
4. Iluminación y texturas en el cauce programable
5. Representación de materiales, texturas y fuentes.

## 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 cámaras

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 1. Cauce gráfico y definición de la cámara.

Subsección 1.1.

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.

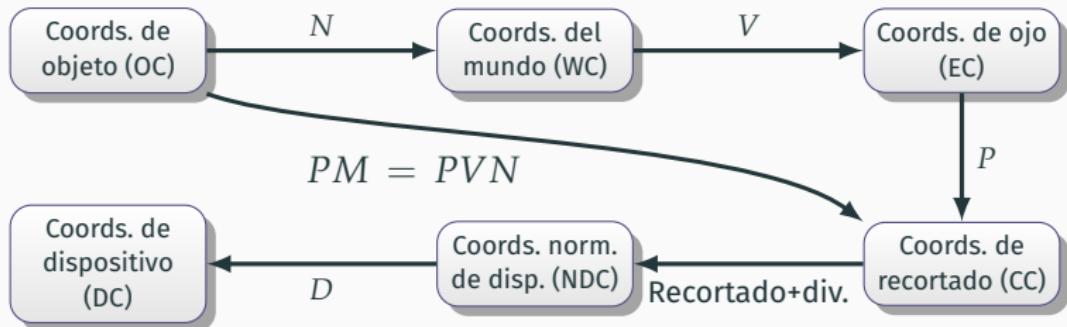
Los pasos del cauce gráfico suelen implementarse en secuencia, cada paso obtiene datos del anterior, los transforma de alguna manera y los entrega al siguiente paso. Los pasos (muy resumidos) son:

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 ( $4 \times 4$ ) 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)

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 1. Cauce gráfico y definición de la cámara.

Subsección 1.2.

Transformación de vista.

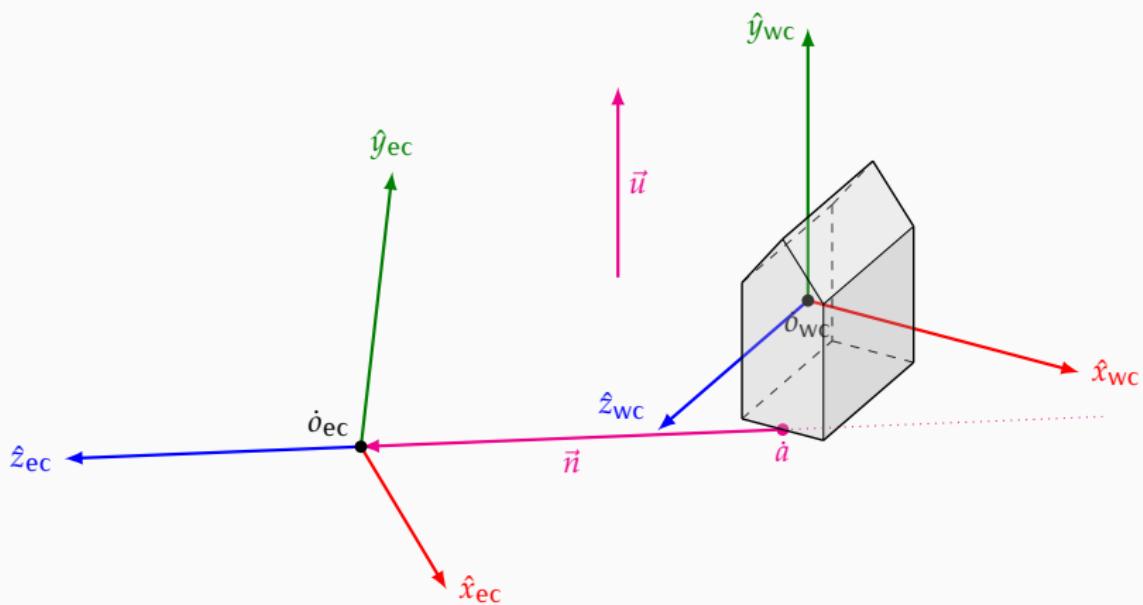
# La transformación de vista.

La **transformación de vista** es el cálculo que permite convertir **coordenadas de mundo** (*world coordinates*, WCC) en **coordenadas de ojo** (o **de cámara**) (*eye or camera coordinates*, ECC).

- ▶ Se usa un marco de referencia cartesiano  $\mathcal{V} = [\hat{x}_{\text{ec}}, \hat{y}_{\text{ec}}, \hat{z}_{\text{ec}}, \dot{o}_{\text{ec}}]$ , llamado **marco de cámara** (o **de vista**), que está posicionado y alineado con la cámara virtual. Las **coordenadas de cámara** (o **de vista**) de un punto son las coordenadas de ese punto en el marco  $\mathcal{V}$ .
- ▶ Para hacer la conversión de coordenadas se debe usar la **matriz de vista**, la llamamos  $V$ .
- ▶ Puesto que el marco de coordenadas de mundo  $\mathcal{W}$  es cartesiano y  $\mathcal{V}$  también, la matriz  $V$  puede construirse fácilmente como la composición de una matriz de traslación por  $\dot{o}_{\text{wc}} - \dot{o}_{\text{ec}}$  seguida de una matriz (ortonormal) de rotación, que tiene las coordenadas de mundo de  $\hat{x}_{\text{ec}}$ ,  $\hat{y}_{\text{ec}}$  y  $\hat{z}_{\text{ec}}$  en sus filas.

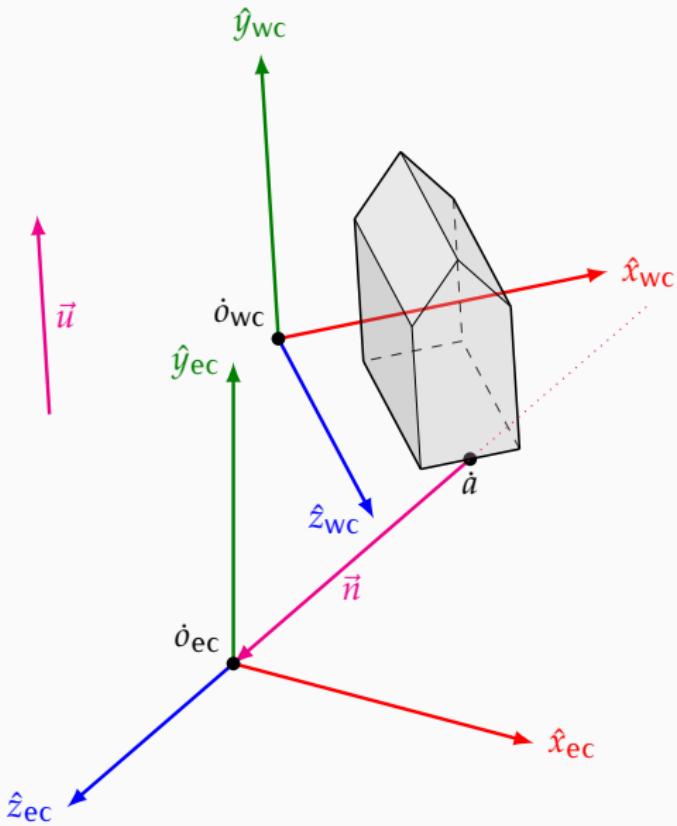
# El marco de coordenadas de vista o de cámara.

El marco (cartesiano) de coordenadas de vista se construye usando el punto  $\dot{a}$ , el punto  $\dot{o}_{ec}$ , el vector  $\vec{u}$  y el vector  $\vec{n}$ . El observador estaría situado en  $\dot{o}_{ec}$  mirando en la dirección de  $-\hat{z}_{ec}$ .



# Escena posterior a transformación de vista.

Aquí vemos la escena anterior una vez transformada por la matriz  $V$



## Cálculo del marco de vista.

El marco de referencia de vista  $\mathcal{V}$ , se define a partir de los siguientes parámetros

$\dot{o}_{ec}$  = 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 de visión* (plano ficticio donde se proyecta la imagen perpendicular al *eje óptico* de la cámara virtual). (*view plane normal, VPN*).

$\dot{a}$  = punto en el eje óptico, también llamado *punto de atención* o *look-at point*.

$\vec{u}$  = 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*)

De los tres parámetros  $\dot{o}_{ec}$ ,  $\vec{n}$  y  $\dot{a}$  solo hay que especificar dos, ya que no son independientes (se cumple  $\dot{o}_{ec} = \dot{a} + \vec{n}$ ).

## Cálculo del marco de vista.

A partir de esos parámetros se obtiene se calculan los versores del marco de vista:

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

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

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

Para que este cálculo pueda hacerse, los vectores  $\vec{u}$  y  $\vec{n}$  no pueden ser nulos ni paralelos, de forma que siempre  $\|\vec{u} \times \vec{n}\| > 0$ .

## Coordenadas del mundo del marco de vista $\mathcal{C}$

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

$$\begin{aligned}\hat{x}_{\text{ec}} &= \mathcal{W}(a_x, a_y, a_z, 0)^t = \mathcal{W}\mathbf{x}_{\text{ec}} \\ \hat{y}_{\text{ec}} &= \mathcal{W}(b_x, b_y, b_z, 0)^t = \mathcal{W}\mathbf{y}_{\text{ec}} \\ \hat{z}_{\text{ec}} &= \mathcal{W}(c_x, c_y, c_z, 0)^t = \mathcal{W}\mathbf{z}_{\text{ec}} \\ \dot{o}_{\text{ec}} &= \mathcal{W}(o_x, o_y, o_z, 1)^t = \mathcal{W}\mathbf{o}_{\text{ec}}\end{aligned}$$

Estas coordenadas se calculan a partir de las coordenadas de mundo (en el marco  $\mathcal{W}$ ) de los vectores  $\vec{u}, \vec{n}$  y el punto  $\dot{o}$  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

La matriz de vista  $V$  es la matriz que convierte desde coordenadas en  $\mathcal{W}$  hacia coordenadas en  $\mathcal{V}$ . Se obtiene como la composición de una matriz de traslación (por  $-\mathbf{o}_{\text{ec}}$ ) seguida de una matriz de rotación (de eje arbitrario)  $R$ :

$$V \equiv R \cdot \text{Tra}[-\mathbf{o}_{\text{ec}}]$$

Donde  $R$  es la matriz (ortonormal) que tiene a  $\mathbf{x}_{\text{ec}}, \mathbf{y}_{\text{ec}}$  y  $\mathbf{z}_{\text{ec}}$  en sus filas:

$$V = \begin{pmatrix} a_x & a_y & a_z & d_x \\ b_x & b_y & b_z & d_y \\ c_x & c_y & c_z & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = \underbrace{\begin{pmatrix} a_x & a_y & a_z & 0 \\ b_x & b_y & b_z & 0 \\ c_x & c_y & c_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_R \underbrace{\begin{pmatrix} 1 & 0 & 0 & -o_x \\ 0 & 1 & 0 & -o_y \\ 0 & 0 & 1 & -o_z \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\text{Tra}[-\mathbf{o}_{\text{ec}}]}$$

(donde  $d_x \equiv -\mathbf{x}_{\text{ec}} \cdot \mathbf{o}_{\text{ec}}$ ,  $d_y \equiv -\mathbf{y}_{\text{ec}} \cdot \mathbf{o}_{\text{ec}}$ ,  $d_z \equiv -\mathbf{z}_{\text{ec}} \cdot \mathbf{o}_{\text{ec}}$ ).

# Fijar la matriz de vista en OpenGL con el cauce fijo

Por tanto, para fijar la matriz de vista, el código OpenGL puede ser del estilo de este:

```
const GLfloat V[4][4] = // matriz V asociada al marco  $\mathcal{V}$  (por filas)
{{ ax, ay, az, dx }, // coords. de mundo de  $\mathbf{x}_{\text{ec}}$ , y dx = - $\mathbf{o}_{\text{ec}} \cdot \mathbf{x}_{\text{ec}}$ 
{ bx, by, bz, dy }, // coords. de mundo de  $\mathbf{y}_{\text{ec}}$ , y dy = - $\mathbf{o}_{\text{ec}} \cdot \mathbf{y}_{\text{ec}}$ 
{ cx, cy, cz, dz }, // coords. de mundo de  $\mathbf{z}_{\text{ec}}$ , y dz = - $\mathbf{o}_{\text{ec}} \cdot \mathbf{z}_{\text{ec}}$ 
{ 0, 0, 0, 1 } // origen de  $\mathcal{V}$ 
};
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
glMultTransposeMatrixf( V );
```

- ▶ Estas instrucciones ponen la matriz de modelado igual a la identidad.
- ▶ Puesto que la matriz está en memoria por filas, usamos **glMultTransposeMatrixf**
- ▶ Si la matriz estuviese en memoria por columnas, usaríamos **glMultMatrixf**.

## Fijando la matriz de vista usando GLU

La función **gluLookAt** (de la librería GLU) permite componer una matriz de vista de forma más cómoda, ya que acepta directamente como parámetros las coordenadas de mundo **o<sub>ec</sub>**, **a** y **u** de  $\vec{o}_{ec}$ ,  $\vec{a}$  y  $\vec{u}$ . Está declarada como sigue:

```
void gluLookAt
( GLdouble o_x, GLdouble o_y, GLdouble o_z,
  GLdouble a_x, GLdouble a_y, GLdouble a_z,
  GLdouble u_x, GLdouble u_y, GLdouble u_z
) ;
```

donde:

$$\begin{aligned}\mathbf{o}_{ec} &= (o_x, o_y, o_z) \\ \mathbf{a} &= (a_x, a_y, a_z) \\ \mathbf{u} &= (u_x, u_y, u_z)\end{aligned}$$

# Construcción explícita de la matriz de vista

Se puede construir explicitamente en la aplicación la matriz de vista, usando la librería de generación de matrices (es especialmente útil para el cauce programable):

```
// construye la misma matriz que glutLookAt:  
Matriz4f MAT_LookAt( const float origen[3],  
                      const float centro[3],  
                      const float vup[3] );
```

O bien, si se conocen las tuplas  $x_{ec}$ ,  $y_{ec}$ ,  $z_{ec}$  y  $\mathbf{o}_{ec}$  que definen el marco de cámara, se puede componer la matriz explicitamente usando **MAT\_Filas** y **MAT\_Traslacion**:

```
// construye V usando la matriz de traslación seguida con la de alineamiento  
Matriz4f V = MAT_Filas( {x_{ec},y_{ec},z_{ec}} )*  
                  MAT_Traslacion( -o_{ec} ) ;
```

# Matriz de vista en el cauce programable

En un cauce programable, la aplicación es necesario:

- ▶ Construir la matriz de vista  $V$  en la aplicación, usando las coordenadas de los vectores y el punto que definen el marco de coordenadas de vista.
- ▶ Construir la matriz *modelview*, como composición de la matriz de modelado  $N$  y matriz de vista  $V$ . Esto puede hacerse bien el shader, bien en la aplicación.
- ▶ Declarar un parámetro uniform en el vertex shader que contiene la matriz *modelview* (o bien la matriz de vista únicamente, de forma separada a la matriz de modelado).
- ▶ Escribir el código del vertex shader de forma que la posición de los vértices sea transformada usando la matriz *modelview*.
- ▶ Fijar el valor del uniform usando **`glUniformMatrix4fv`** y la localización.

# Transformación de vista en el *vertex shader*

En este ejemplo, el uniform **matrizMV** tiene la matriz *modelview*:

```
// parámetros de entrada uniform (iguales en todos los vértices de cada primitiva)
uniform mat4 matrizMV ;          // matriz 4x4 de transf. de coord. de vértices
uniform mat4 matrizMV_nor;        // matriz 4x4 de transf. de normales
uniform mat4 matrizP ;           // matriz 4x4 de proyección (produce coord.pantalla)

// variables de salida varying (atributos de vértice: serán interpolados a pixels)
varying vec4 var_posic_ec;       // posición (en coords de cámara)
varying vec3 var_normal_ec;       // normal (en coords. de camara)
varying vec4 var_color;          // color
varying vec2 var_coord_text;     // coordenadas de textura

// vars. de entrada predefinidas (posición + atributos, recibidos de la aplicación):
//      gl_Vertex, gl_Normal, gl_Color, gl_MultiTexCoord0

void main() // escribe variables 'varying', más 'gl_Position'
{
    var_posic_ec  = matrizMV * gl_Vertex;          // transf. coord. recibida
    var_normal_ec = matrizMV_nor * gl_{Normal}; // transf. normal recibida
    var_color     = gl_Color ;                      // usar color enviado
    var_coord_text= gl_MultiTexCoord0.st ;         // usa cc.t. enviadas
    gl_Position    = matrizP * var_posic_ec;        // proyecta a pantalla
}
```

## Fijar $V$ usando la clase **Cauce**

Para facilitar la definición de  $V$  en el cauce fijo y el programable, se puede usar la función **fijarMatrizVista(v)** de la clase **Cauce**, usando una **Matriz4f** de vista V:

- ▶ Copia la matriz V sobre la matriz de vista o modelview actual, que pierde el valor que tuviese antes.
- ▶ Fija la matriz de modelado  $N$  como igual a la matriz identidad.
- ▶ Fija la matriz de modelado de normales como igual a la matriz identidad.
- ▶ Reinicializa la pila de matrices de modelado.
- ▶ Reinicializa la pila de matrices de modelado de normales

En la siguiente transparencia vemos el código de este método en el cauce programable y en el fijo.

# Implementaciones de fijarMatrizVista

```
void CauceFijo::fijarMatrizVista( const Matriz4f & nue_mat_vista )
{
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    glMultMatrixf( nue_mat_vista );
}

void CauceProgramable::fijarMatrizVista( const Matriz4f & nue_mat_vista )
{
    // registrar matriz de vista y matriz de modelado en la instancia
    mat_vista      = nue_mat_vista ;
    mat_modelado   = MAT_Ident();
    mat_modelado_nor = MAT_Ident();

    // cambiar matriz de vista y matriz de modelado en los shaders
    glUseProgram( id_prog );
    glUniformMatrix4fv( loc_mat_vista, 1, GL_FALSE, mat_vista );
    glUniformMatrix4fv( loc_mat_modelado, 1, GL_FALSE, mat_modelado );
    glUniformMatrix4fv( loc_mat_modelado_nor, 1, GL_FALSE, mat_modelado_nor );

    // vaciar pila de matrices de modelado
    pila_mat_modelado.clear();
    pila_mat_modelado_nor.clear();
}
```

# Problemas: parámetros para una vista concreta (1/3)

## Problema 3.1.

Supongamos una escena que contiene una representación visible del marco de coordenadas del mundo como tres flechas (roja, verde y azul), como ocurre en las prácticas. Queremos visualizar esa escena en pantalla, de forma que:

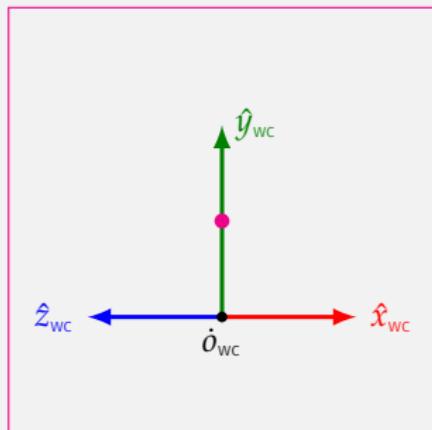
1. El eje Y aparezca vertical, hacia arriba, el eje X horizontal, hacia la derecha, el eje Z horizontal, hacia la izquierda (los ejes X y Z se visualizan con la misma longitud aparente).
2. El punto de coordenadas  $(0,0.5,0)$  (aparece como un disco de color morado en la figura) debe aparecer en el centro del viewport
3. El observador (foco de la proyección) estará a 3 unidades de distancia del punto  $(0,0.5,0)$

(continua en la siguiente transparencia).

# Problemas: parámetros para una vista concreta (2/3)

## Problema 3.1. (continuación)

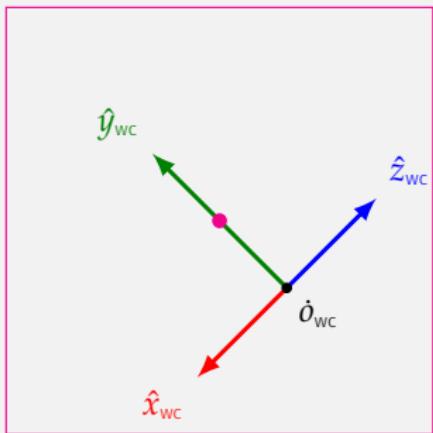
Escribe unos valores que podríamos usar para  $\mathbf{a}$ ,  $\mathbf{u}$  y  $\mathbf{n}$  de forma que se cumplan estos requisitos. En la figura se observa una vista esquemática de como quedaría la figura en un viewport cuadrado, no necesariamente a escala.



# Problema: parámetros para una vista concreta (3/3)

## Problema 3.2.

Repite el problema anterior, pero ahora para esta vista:



# Problema: construcción de la matriz de vista

## Problema 3.3.

Escribe el código para calcular los vectores de coordenadas  $\mathbf{x}_{ec}$ ,  $\mathbf{y}_{ec}$ ,  $\mathbf{z}_{ec}$  y  $\mathbf{o}_{ec}$  que definen el marco de vista a partir de los vectores de coordenadas  $\mathbf{a}$ ,  $\mathbf{u}$  y  $\mathbf{n}$  (todos estos vectores de coordenadas son de tipo **Tupla3f**).

## Problema 3.4.

Partiendo de los vectores de coordenadas  $\mathbf{x}_{ec}$ ,  $\mathbf{y}_{ec}$ ,  $\mathbf{z}_{ec}$  y  $\mathbf{o}_{ec}$  que se calculan en el problema anterior, escribe el código que calcula explícitamente las 16 entradas de la matriz de vista (crea una **Matriz4f** llamada  $\mathbf{V}$  y luego asigna valor a  $\mathbf{V}(i, j)$  para cada fila  $i$  y columna  $j$ , ambas entre 0 y 3).

Informática Gráfica, curso 2021-22.

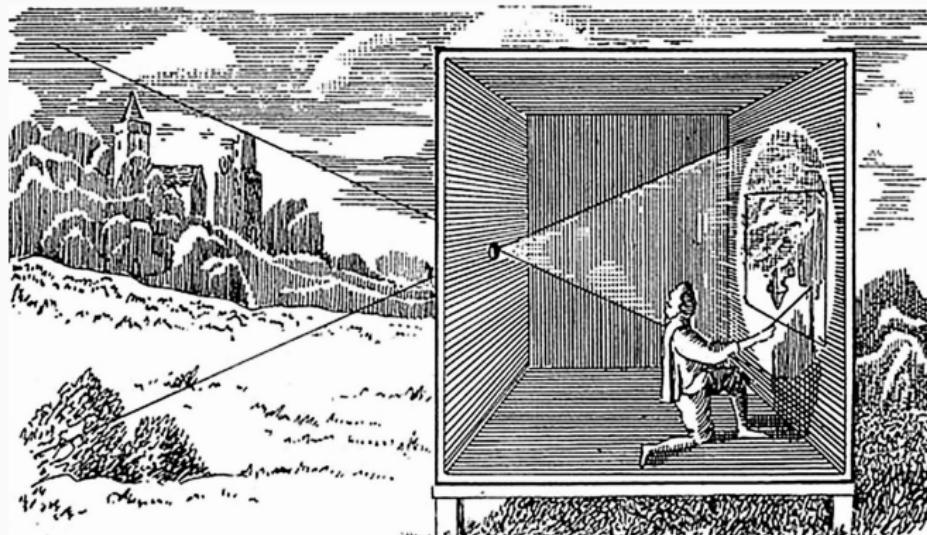
Teoría. Tema 3. Visualización.

Sección 1. Cauce gráfico y definición de la cámara.

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:

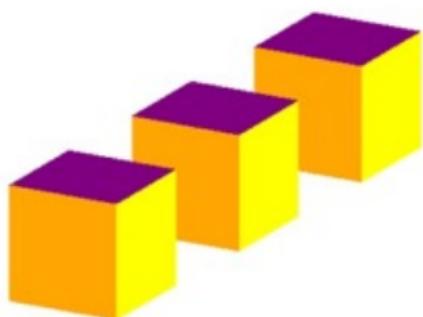
- ▶ Dicho plano se denomina **plano de visión (viewplane)**, es siempre perpendicular al eje Z del marco de vista.

La proyección puede ser de dos tipos

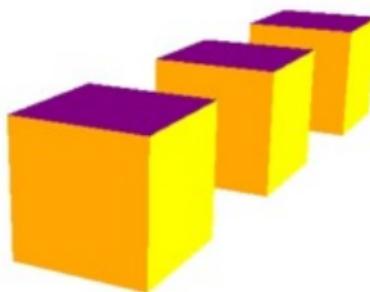
- ▶ **Proyección perspectiva:** los vértices se proyectan sobre el plano de visión usando líneas que van desde cada punto al origen del marco de coordenadas de la cámara (a esas líneas se les llama **proyectores**, el origen actua como **foco** de la proyección). La coordenada Z del plano de visión debe ser estrictamente positiva.
- ▶ **Proyección ortográfica (o paralela):** los proyectores 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



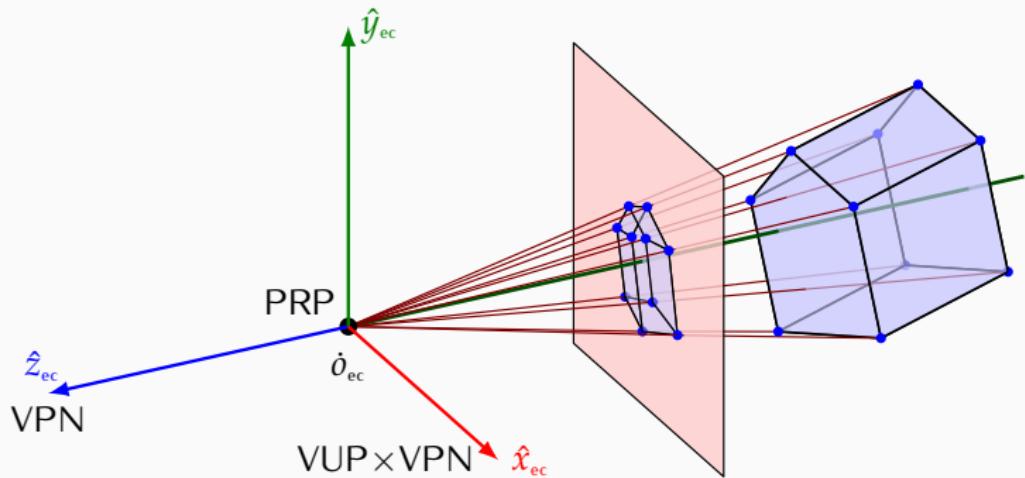
Perspective Projection

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

Microsoft WPF documentation: 3D Graphics Overview

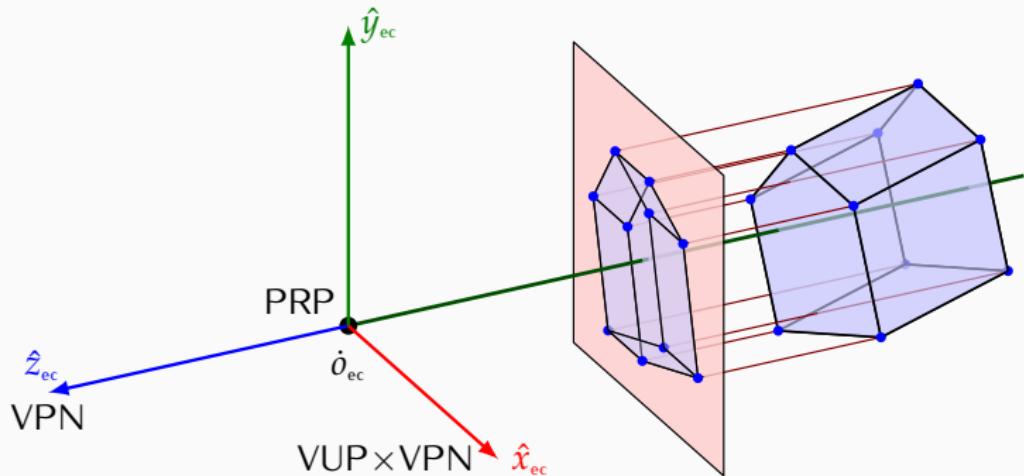
# Proyección perspectiva

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.).

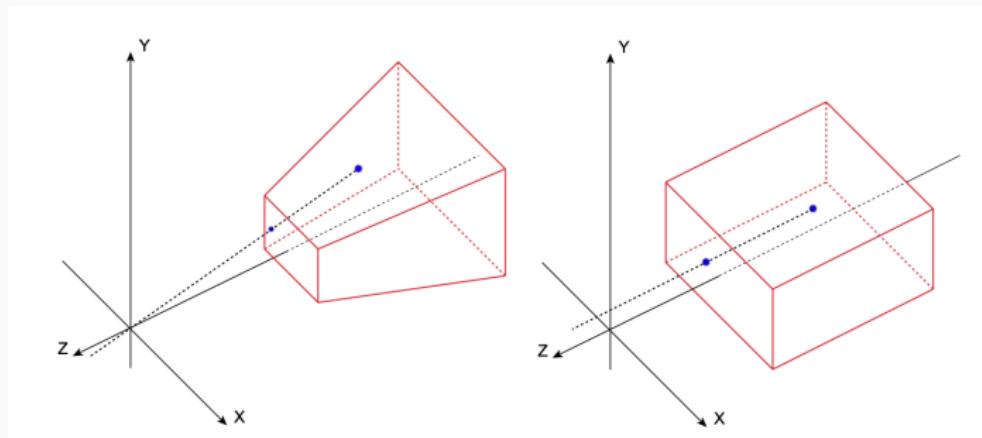


Imagen obtenida de: SGI® OpenGL Multipipe™ SDK User's Guide

## 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) determinan la transformación de la tupla  $(x_{ec}, y_{ec}, z_{ec})$ , que está en coordenadas de vista en la tupla  $(x_{ndc}, y_{ndc}, z_{ndc})$  en NDCC (*coordenadas normalizadas de dispositivo*, 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  $z_{ec} = -n$  en EC se transforma en el plano  $z_{ndc} = -1$  en NDC.
  - ▶ El plano  $z_{ec} = -f$  en EC se transforma en el plano  $z_{ndc} = +1$  en NDC.
- ▶ En la proyección perspectiva, se exige además  $0 < n$  y  $0 < f$ .
- ▶ 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  $z_{ec} = -n$  se llama **plano de recorte delantero**
- ▶ El plano  $z_{ec} = -f$  se llama **plano de recorte trasero**

## Parámetros del view-frustum. Extension 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  $x_{ec} = l$  en EC se transforma en el plano  $x_{ndc} = -1$  en NDC.
  - ▶ El plano  $x_{ec} = r$  en EC se transforma en el plano  $x_{ndc} = +1$  en NDC.
  - ▶ El plano  $y_{ec} = b$  en EC se transforma en el plano  $y_{ndc} = -1$  en NDC.
  - ▶ El plano  $y_{ec} = t$  en EC se transforma en el plano  $y_{ndc} = +1$  en NDC.
- ▶ En proy. perspectiva:
  - ▶ El plano  $-nx_{ec} = lz_{ec}$  (EC) se transf. en el plano  $x_{ndc} = -1$  en NDC.
  - ▶ El plano  $-nx_{ec} = rz_{ec}$  (EC) se transf. en el plano  $x_{ndc} = +1$  en NDC.
  - ▶ El plano  $-ny_{ec} = bz_{ec}$  (EC) se transf. en el plano  $y_{ndc} = -1$  en NDC.
  - ▶ El plano  $-ny_{ec} = tz_{ec}$  (EC) se transf. en el plano  $y_{ndc} = +1$  en NDC.

# Propiedades de la extensión en X e Y

Hay que tener en cuenta que:

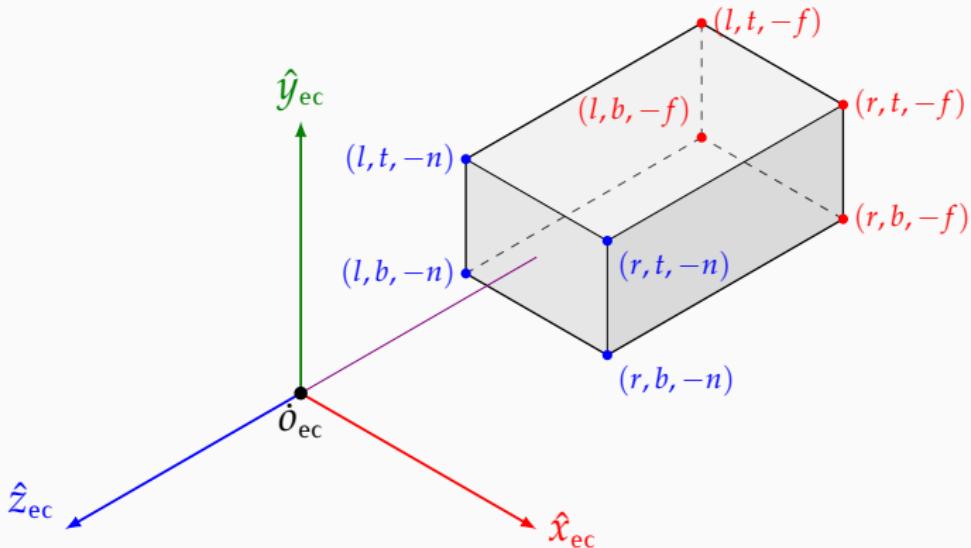
- ▶ Aunque esto no es requerido estrictamente, usualmente se seleccionan los parámetros de forma que  $l < r$  y  $b < t$ .
- ▶ Cuando se cumple  $l = -r$  y  $b = -t$ , decimos que el **view-frustum está centrado** (el eje Z pasa por el centro de las caras delantera y trasera). Esto es lo más usual, 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.

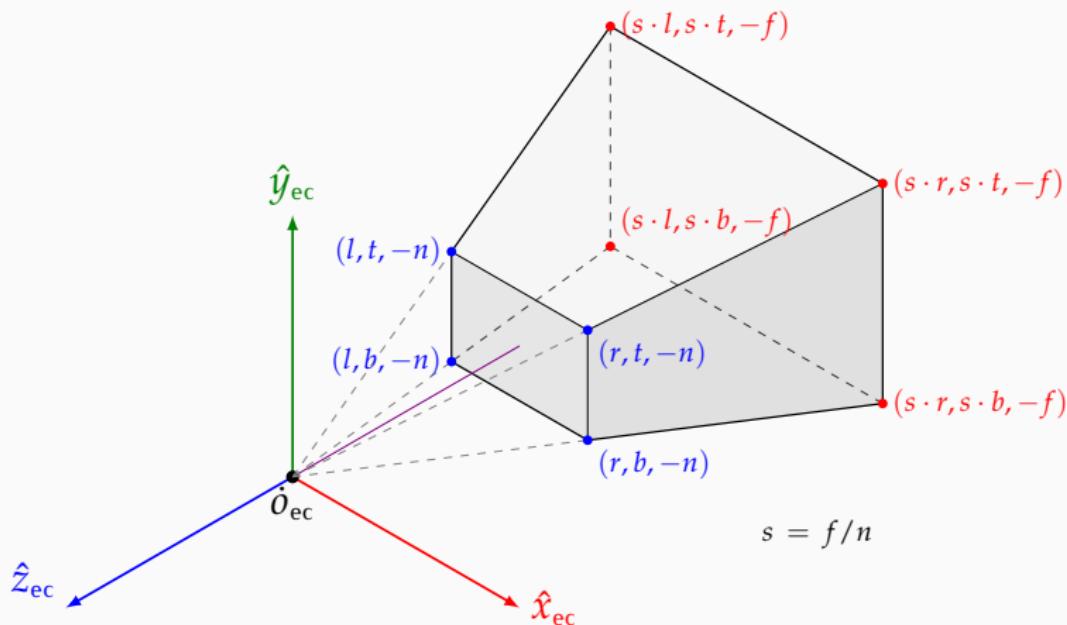
En pr. ortográfica el view-frustum es un **ortoedro**. Contiene los puntos cuyas coordenadas de cámara  $(x_{ec}, y_{ec}, z_{ec})$  cumplen:

$$l \leq x_{ec} \leq r \quad b \leq y_{ec} \leq t \quad -f \leq z_{ec} \leq -n$$



# Parámetros en la proyección perspectiva (1/2)

En perspectiva, el view-frustum es una pirámide rectangular truncada:



## Parámetros en la proyección perspectiva (2/2)

Los puntos dentro del view-frustum son aquellos cuyas coordenadas de cámara  $(x_{ec}, y_{ec}, z_{ec})$  cumplen:

En el eje X:

$$l \leq x_{ec} \left( \frac{n}{-z_{ec}} \right) \leq r$$

En el eje Y:

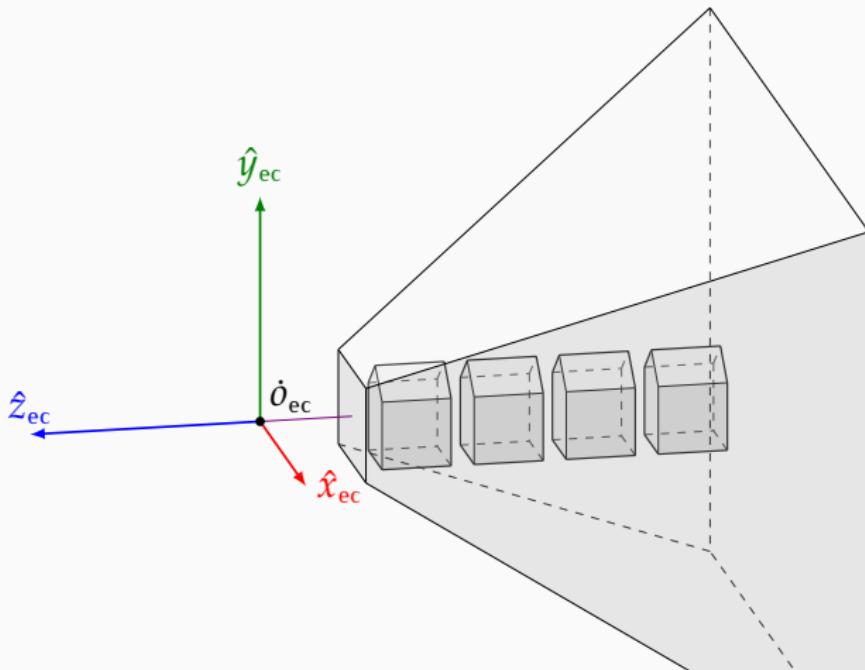
$$b \leq y_{ec} \left( \frac{n}{-z_{ec}} \right) \leq t$$

En el eje Z:

$$-f \leq z_{ec} \leq -n$$

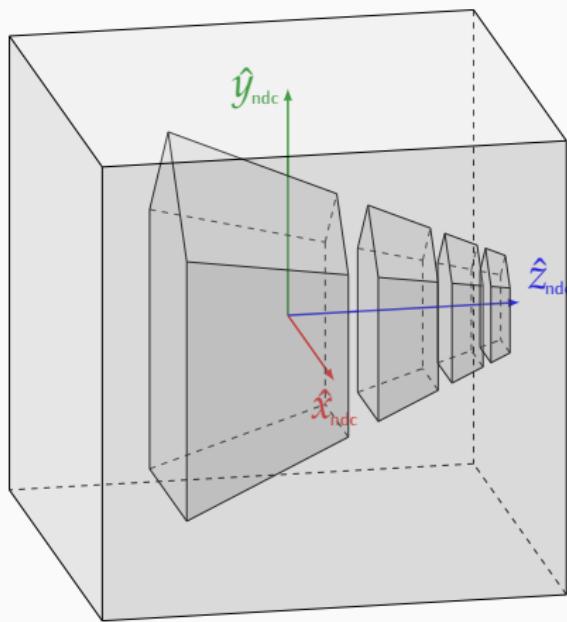
# Escena de ejemplo para transformación perspectiva

Suponemos que partimos de una escena que vamos a proyectar usando perspectiva. En el espacio de coordenadas de cámara, la escena es esta:



# Escena proyectada por transformación perspectiva

El efecto de la transformación de proyección perspectiva será hacer más pequeños los objetos más alejados del observador, y situar la escena en un cubo de lado 2:



# 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  $\mathbf{o}_{ec}$ :

- ▶ Dado un punto  $\mathbf{p} = \mathcal{V}(x_{ec}, y_{ec}, z_{ec}, w_{ec})$  queremos calcular las coordenadas de su proyección  $(x', y', z', w')$  (en principio, con  $w' = w_{ec} = 1$ ).
- ▶ Si se asume  $z_{ec} < 0$  (el punto está en la rama negativa del eje Z), podemos hacer:

$$x' \equiv \frac{x_{ec}n}{-z_{ec}} \quad y' \equiv \frac{y_{ec}n}{-z_{ec}} \quad z' \equiv \frac{z_{ec}n}{-z_{ec}} = -n$$

esta transformación tiene dos problemas:

- ▶ Las coordenadas resultado no están 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:

$$x'' \equiv 2 \left( \frac{x' - l}{r - l} \right) - 1 = \frac{a_0 x_{ec}}{-z_{ec}} - a_1 = \frac{a_0 x_{ec} + a_1 z_{ec}}{-z_{ec}}$$

$$y'' \equiv 2 \left( \frac{y' - b}{t - b} \right) - 1 = \frac{b_0 y_{ec}}{-z_{ec}} - b_1 = \frac{b_0 y_{ec} + b_1 z_{ec}}{-z_{ec}}$$

donde hemos usado estas cuatro constantes:

$$a_0 \equiv \frac{2n}{r - l} \quad a_1 \equiv \frac{r + l}{r - l} \quad b_0 \equiv \frac{2n}{t - b} \quad b_1 \equiv \frac{t + b}{t - b}$$

# 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  $c_0$  y  $c_1$ :

$$z'' \equiv \frac{c_0 z_{ec} + c_1}{-z_{ec}} \quad \text{donde:} \quad c_0 \equiv \frac{n+f}{n-f} \quad c_1 \equiv \frac{2fn}{n-f}$$

- ▶ los dos valores  $c_2$  y  $c_3$  se eligen de forma que, para  $z_{ec} = -n$ , se hace  $z'' = -1$ , y para  $z_{ec} = -f$ , 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:

$$x'' \equiv \frac{a_0 x_{ec} + a_1 z_{ec}}{-z_{ec}}$$

$$y'' \equiv \frac{b_0 x_{ec} + b_1 z_{ec}}{-z_{ec}}$$

$$z'' \equiv \frac{c_0 z_{ec} + c_1}{-z_{ec}} = \frac{c_0 z_{ec} + c_1 w_{ec}}{-z_{ec}}$$

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

## Obtención de las coordenadas de recortado

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

$$x_{cc} \equiv a_0 x_{ec} + a_1 z_{ec}$$

$$y_{cc} \equiv b_0 y_{ec} + b_1 z_{ec}$$

$$z_{cc} \equiv c_0 z_{ec} + c_1 w_{ec}$$

$$w_{cc} \equiv -z_{ec}$$

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

- ▶ se ha eliminado la división por  $-z_{ec}$ , el resto es igual
- ▶ esta división se hace más adelante en el cauce gráfico
- ▶ para ello, el denominador de la división ( $-z_{ec}$ ) queda guardado en  $w_{cc}$  (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  $w_{ec} = 1$ ) y produce coordenadas de recortado (con  $w_{cc} \neq 1$ ):

$$\begin{pmatrix} x_{cc} \\ y_{cc} \\ z_{cc} \\ w_{cc} \end{pmatrix} = \begin{pmatrix} a_0 & 0 & a_1 & 0 \\ 0 & b_0 & b_1 & 0 \\ 0 & 0 & c_0 & c_1 \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x_{ec} \\ y_{ec} \\ z_{ec} \\ 1 \end{pmatrix}$$

evidentemente, podemos definir entonces la matriz  $Q$  de esta forma:

$$Q \equiv \begin{pmatrix} a_0 & 0 & a_1 & 0 \\ 0 & b_0 & b_1 & 0 \\ 0 & 0 & c_0 & c_1 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

# 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)

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

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

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

(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:

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

de forma que ahora hacemos:

$$(x_{cc}, y_{cc}, z_{cc}, w_{cc})^t = O(x_{ec}, y_{ec}, z_{ec}, w_{ec})^t$$

donde  $w_{cc}$  sí vale 1 con seguridad.

## Matriz de proyección en OpenGL (cauce fijo)

En el estado de OpenGL, cuando se usa el cauce de funcionalidad prefijada, 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*, usando la constante **GL\_MATRIXMODE**, de forma que posteriores operaciones se realicen sobre la matriz  $P$ .
- ▶ A modo de ejemplo, estas llamadas hacen  $P$  igual a la matriz identidad:

```
glMatrixMode( GL_PROJECTION ); // entrar en modo 'matriz proyeccion'  
glLoadIdentity(); // hace P := Ide
```

## Definición de la matriz de proyección (cauce fijo)

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  $P := P \cdot Q$ 
```

Si queremos una proyección ortográfica (matriz  $O$ ), 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  $P := P \cdot O$ 
```

## Transf. de proyección con gluPerspective

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 β, GLdouble a, // calcula Q a partir de β,a,n,f  
                GLdouble n, GLdouble f ) ; // y luego hace P := P · Q
```

esta función equivale a **glFrustum** centrado con:

$$t \equiv n \tan\left(\frac{\beta}{2}\right) \quad b \equiv -t \quad r \equiv at \quad l \equiv -r$$

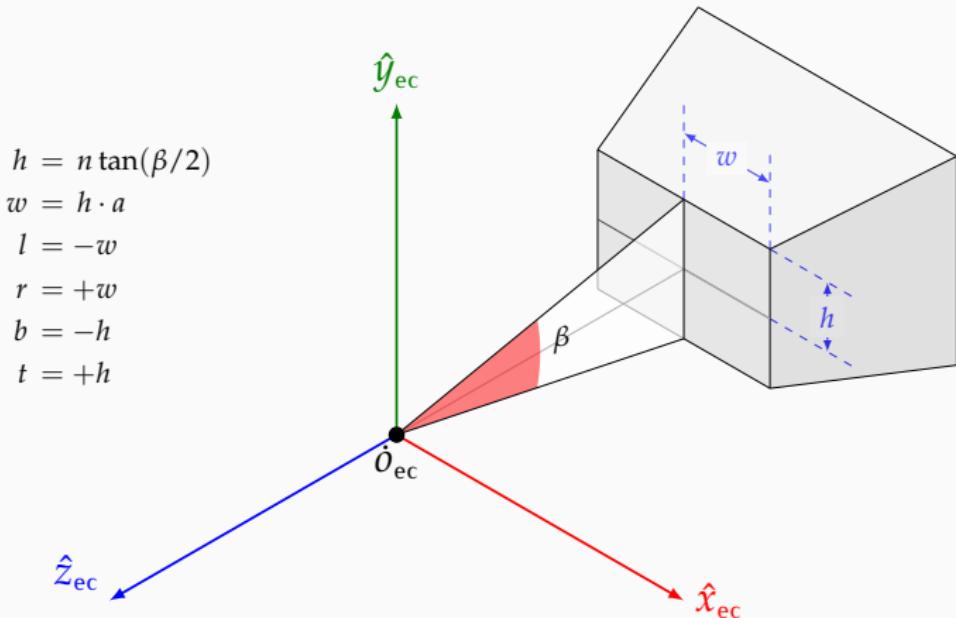
donde:

$\beta$   $\equiv$  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

$a$   $\equiv$  **relación de aspecto** (*aspect ratio*) de la imagen a producir: su ancho (n. columnas) dividido por el alto (n. filas).

# Parámetros de gluPerspective

El significado de los parámetros se aprecia en esta figura:



Esta perspectiva es *centrada*, ya que  $r = -l$  y  $t = -b$

# Construcción explícita de la matriz de proyección

Para construir explicitamente una matriz de proyección (para el cauce programable) se pueden usar funciones que devuelven **Matriz4f**. En el código de prácticas se encuentran disponibles estas:

```
#include <matrices-tr.h> // include de librería de generación de matrices

// construye matriz O (misma matriz que glOrtho)
Matriz4f MAT_Ortografica( const float l, const float r,
                           const float b, const float t,
                           const float n, const float f );

// construye matriz Q (misma matriz que glFrustum)
Matriz4f MAT_Frustum      ( const float l, const float r,
                            const float b, const float t,
                            const float n, const float f );

// construye matriz Q (misma matriz que gluPerspective)
Matriz4f MAT_Perspectiva( const float fovy, const float a,
                           const float n, const float f );
```

# Matriz de proyección en el cauce programable

Si se usa un cauce programable, es necesario:

- ▶ Declarar un parámetro *uniform* en el *vertex shader* de tipo **mat4**, que contendrá la matriz de proyección.
- ▶ Incluir código en el *vertex shader* de forma que la última etapa de transformación de la posición de un vértice sea multiplicar por esa matriz.
- ▶ Como parte de la inicialización del cauce, obtener y guardar la localización de dicho parámetro (con **glGetUniformLocation**).
- ▶ Al inicio de la aplicación (o al inicio de la visualización de un frame), se debe construir la matriz de proyección  $P$  (de tipo **Matriz4f**) usando **MAT\_Frustum**, **MAT\_Ortografica** u otras funciones parecidas.
- ▶ Finalmente, usando la localización y  $P$ , se debe fijar el valor del parámetro *uniform* con **glUniformMatrix4fv**.

## Código del *vertex shader*

En este ejemplo, el uniform **matrizP** tiene la matriz de proyección:

```
// parámetros de entrada uniform (iguales en todos los vértices de cada primitiva)
uniform mat4 matrizMV ;           // matriz 4x4 de transf. de coord. de vértices
uniform mat4 matrizMV_nor;        // matriz 4x4 de transf. de normales
uniform mat4 matrizP ;           // matriz 4x4 de proyección (produce coord.pantalla)

// variables de salida varying (atributos de vértice: serán interpolados a pixels)
varying vec4 var_posic_ec;       // posición (en coords de cámara)
varying vec3 var_normal_ec;      // normal (en coords. de camara)
varying vec4 var_color;          // color
varying vec2 var_coord_text;     // coordenadas de textura

// vars. de entrada predefinidas (posición + atributos, recibidos de la aplicación):
//      gl_Vertex, gl_Normal, gl_color, gl_MultiTexCoord0

void main() // escribe variables 'varying', más 'gl_Position'
{
    var_posic_ec  = matrizMV * gl_Vertex;           // transf. coord. recibida
    var_normal_ec = matrizMV_nor * gl_Normal;        // transf. normal recibida
    var_color     = gl_color ;                      // usar color enviado
    var_coord_text= gl_MultiTexCoord0.st ;           // usa cc.t. enviadas
    gl_Position    = matrizP * var_posic_ec;         // proyecta a pantalla
}
```

## Uso de la clase cauce

Para facilitar la gestión del cauce fijo y el cauce programable se puede usar el método **fijarMatrizProyeccion** del la clase **Cauce**. Las respectivas implementaciones son básicamente así:

```
void CauceFijo::fijarMatrizProyeccion( const Matriz4f & nue_mat_pro )
{
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    glMultMatrixf( nue_mat_pro );
}
```

```
void CauceProgramable::fijarMatrizProyeccion( const Matriz4f & nue_mat_pro )
{
    mat_proyeccion = nue_mat_pro; // no es estrictamente necesario
    glUseProgram( id_prog );
    glUniformMatrix4fv( loc_mat_proyeccion, 1, GL_FALSE, mat_proyeccion );
}
```

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 1. Cauce gráfico y definición de la cámara.

Subsección 1.4.

Recortado y división por  $W$ .

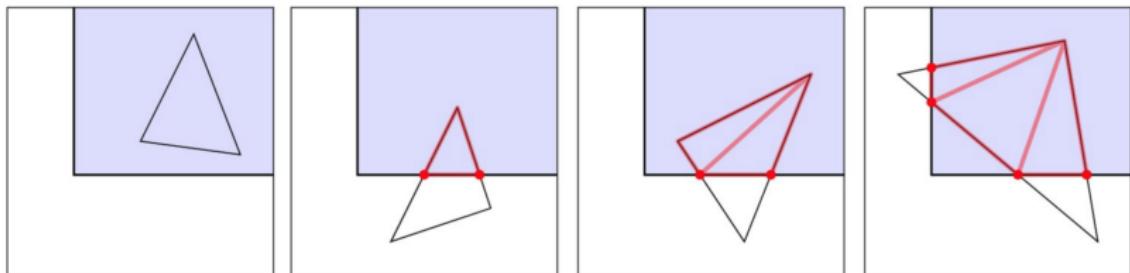
## Recortado

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 (se vis) 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.

# Inserción de nuevos vértices y triángulos

Varios ejemplos de recortado de triángulos:



- ▶ Las coordenadas y otros atributos de los nuevos vértices se interpolan a partir de los vértices en los dos extremos de la arista donde se inserta el nuevo.
- ▶ Se hace recortado independiente por cada uno de los 6 planos de recorte.

Figura obtenida de *CMU Computer Graphics Course (fall 2019)*:

☞ <http://15462.courses.cs.cmu.edu/fall2019/>

## División por W. Coordenadas normalizadas de dispositivo.

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

$$(x_{ndc}, y_{ndc}, z_{ndc}, 1) = \frac{1}{w_{cc}} (x_{cc}, y_{cc}, z_{cc}, w_{cc})$$

los valores  $x_{ndc}$ ,  $y_{ndc}$  y  $z_{ndc}$  están los tres en el intervalo  $[-1, 1]$  (los vértices ya han pasado el recortado).

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 1. Cauce gráfico y definición de la cámara.

Subsección 1.5.  
Transformación de viewport.

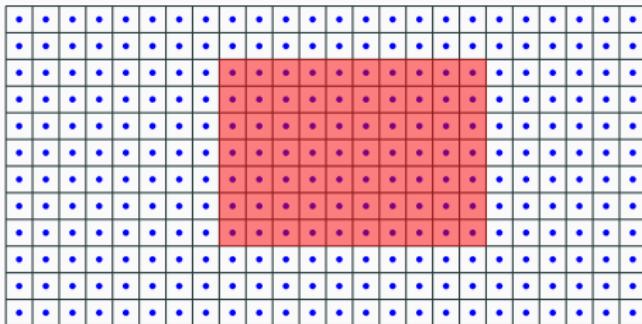
# Transformación de Viewport

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 y pixels del viewport

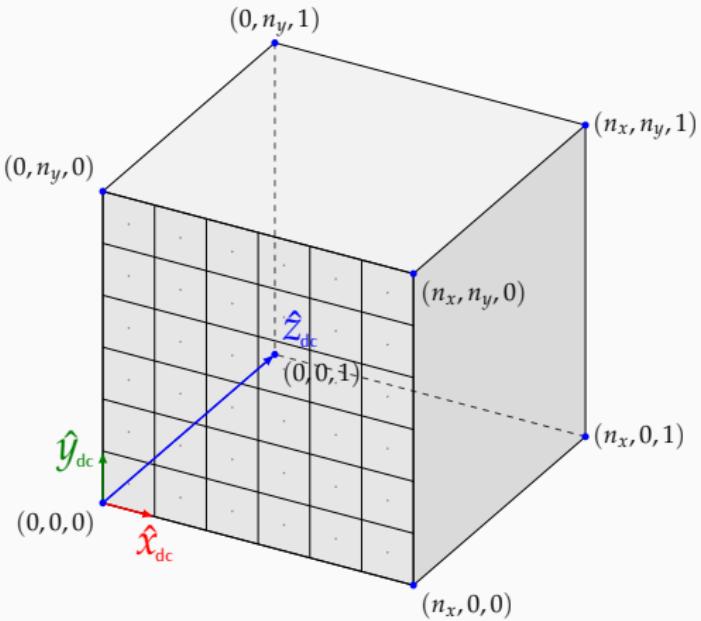
En coordenadas de dispositivo, podemos asociar una región cuadrada (de lado unidad) a cada pixel en el plano de la ventana. El viewport (en rojo) es un bloque rectangular de pixels, contenido en el bloque rectangular correspondiente a la ventana o imagen completa:



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).

# El espacio de coordenadas de dispositivo

En 3D el espacio de coordenadas de dispositivo es un ortoedro. Se puede visualizar como aparece aquí, incluyendo el marco de coordenadas de dispositivo:



## Matriz del viewport y parámetros en OpenGL.

OpenGL tiene en su estado una matriz  $4 \times 4$  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.

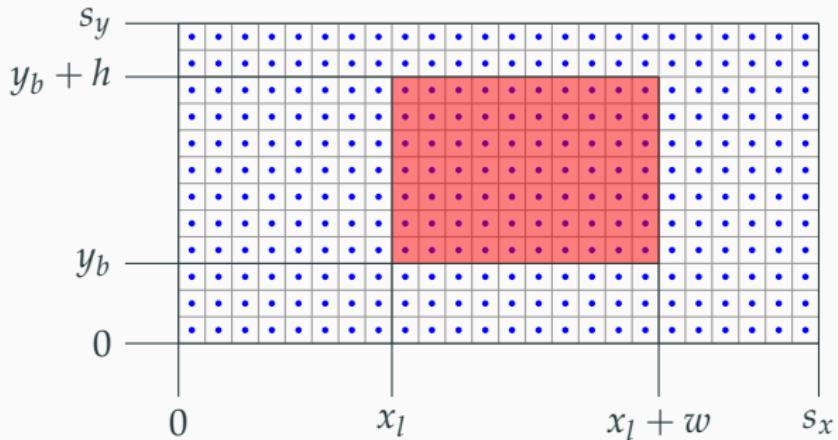
$w, h$  (*width* y *height*) número total (entero no negativo) de columnas y de filas de pixels (respectivamente) que ocupa el viewport.

$n_d, f_d$  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:

$$D = \text{Tra}[x_l, y_b, n_d] \cdot \text{Esc}[w, h, f_d - n_d] \cdot \text{Esc}[1/2] \cdot \text{Tra}[1, 1, 1]$$

por tanto, las **coordenadas de dispositivo**  $(x_{dc}, y_{dc}, z_{dc}, 1)$  se definen a partir de las normalizadas  $(x_{ndc}, y_{ndc}, z_{ndc}, 1)$  de esta forma:

$$\begin{aligned}x_{dc} &= (x_{ndc} + 1)w/2 + x_l \\y_{dc} &= (y_{ndc} + 1)h/2 + y_b \\z_{dc} &= (z_{ndc} + 1)(f_d - n_d)/2 + n_d\end{aligned}$$

## La matriz de viewport $D$

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

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

de forma que:

$$(x_{dc}, y_{dc}, z_{dc}, 1)^t = D (x_{ndc}, y_{ndc}, z_{ndc}, 1)^t$$

# 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.

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 1. Cauce gráfico y definición de la cámara.

Subsección 1.6.  
Representación de cámaras.

# Representación de cámaras

La clase **Camara** encapsula todos los parámetros relacionados con la matriz de vista y proyección.

- ▶ Es una clase base con funcionalidad mínima. Se derivan clases con funcionalidad más avanzada.
- ▶ Incluye una matriz  $4 \times 4$  de vista  $V$  y otra de proyección  $P$ .
- ▶ El método **activar(c)** permite activar una cámara en un cauce **c** (referencia a una instancia de una clase derivada de **Cauce**). Este método simplemente fija las matrices en el cauce usando las que hay en la instancia.
- ▶ El método **actualizarMatrices** es un método **virtual**, que se encarga de calcular  $V$  y  $P$  a partir de los parámetros específicos de cada tipo de cámara.
- ▶ Por defecto, esta clase base define una cámara ortográfica que visualiza un cubo de lado 2 unidades en X y centro en el origen (en coords. de  $\mathcal{W}$ ).

# Clase Camara: declaración

```
class Camara
{
public:
    // fija las matrices model-view y projection en el cauce
    void activar( Cauce & cauce ) ;
    // cambio el valor de 'ratio_vp' (alto/ancho del viewport)
    void fijarRatioViewport( const float nuevo_{cc}ratio ) ;
    // lee la descripción de la cámara (y probablemente su estado)
    virtual std::string descripcion() ;

protected:
    bool      matrices_actualizadas = false; // true si matrices actualizadas
    Matriz4f  matriz_vista        = MAT_Ident(),    // matriz de vista
              matriz_proye       = MAT_Ident();    // matriz de proyección
    float     ratio_vp            = 1.0 ;           // ratio viewport (alto/ancho)

    // actualiza matriz_vista y matriz_proye a partir de los parámetros
    // específicos de cada tipo de cámara
    virtual void actualizarMatrices() ;
};

}
```

El ratio  $Y/X$  se almacena siempre para evitar deformaciones.

# Activación y actualización de una cámara

Cualquier tipo de cámara se activa fijando las matrices en el cauce a partir de las que se guardan en la instancia. Antes de eso se actualizan las matrices (recalcula **matriz\_vista** y **matriz\_proyección** si no estaban actualizadas)

```
void Camara::activar( Cauce & cauce )
{
    actualizarMatrices(); // recalcula si no están actualizadas
    cauce.fijarMatrizVista( matriz_vista );
    cauce.fijarMatrizProyeccion( matriz_proye );
}
```

El método **fijarRatioViewport** permite cambiar **ratio\_vp** para adaptarlo a las proporciones del viewport en uso:

```
void Camara::fijarRatioViewport( const float nuevo_ratio )
{
    ratio_vp = nuevo_ratio ;           // registrar nuevo ratio
    matrices_actualizadas = false; // matrices deben actualizarse antes de activar
}
```

# Matrices de la clase base Camara

La cámara básica define un view-frustum de lado 2 en X y en Z, y de lado  $2r$  en Y (donde  $r$  es el ratio del viewport, **ratio\_vp**). Está centrado en el origen de  $\mathcal{W}$ .

Por tanto, el método **actualizarMatrices** queda así:

```
void Camara::actualizarMatrices() // método virtual: redefinido en derivadas.  
{  
    if ( matrices_actualizadas )  
        return ;  
    matriz_vista = MAT_Ident();  
    matriz_proye = MAT_Escalado( 1.0f, 1.0f/ratio_vp, 1.0f );  
    matrices_actualizadas = true ;  
}
```

# Cámara orbital simple

En prácticas usamos una instancia de **CamaraOrbitalSimple** (derivada indirectamente de **Camara**), que define una cámara centrada en el origen con tres parámetros: dos ángulos (**a** y **b**) y una distancia al origen (**d**):

```
void CamaraOrbitalSimple::actualizarMatrices()
{
    // matriz de vista:
    matriz_vista = MAT_Traslacion( 0.0, 0.0, -d ) * // (3) despl. en Z por d
                    MAT_Rotacion( b, 1.0,0.0,0.0 ) * // (2) rotación eje X por b
                    MAT_Rotacion( -a, 0.0,1.0,0.0 ) ; // (1) rotacion eje Y por a
    // matriz de proyección:
    constexpr float          // parámetros de la matriz perspectiva (fijos)
        fovy_grad = 60.0,    // apertura vertical de campo, en grados
        near      = 0.05,    // distancia al plano de recorte delantero
        far       = near+1000.0 ; // dist. al plano de recorte trasero
    matriz_proye = MAT_Perspectiva( fovy_grad, ratio_vp, near, far );

    matrices_actualizadas = true; // registra que las matrices están ya actualizadas
}
```

# Problemas: parámetros de matriz de proyección (1)

## Problema 3.5.

Queremos visualizar una escena con mallas indexadas de la cual sabemos que tiene todos los vértices dentro de un cubo de lado  $s$  unidades cuyo centro es el punto de coordenadas del mundo  $\mathbf{c} = (c_x, c_y, c_z)$ .

Para construir la matriz de vista, se situa el observador en el punto  $\mathbf{o}_{\text{ec}} = (c_x, c_y, c_z + s + 2)$ , el punto de atención  $\mathbf{a}$  se hace igual a  $\mathbf{c}$  (el centro del cubo se ve en el centro de la imagen), y el vector  $\mathbf{u}$  es  $(0, 1, 0)$ . Se visualizará en un viewport cuadrado.

(continua en la siguiente página)

## Problemas: parámetros de matriz de proyección (2)

### Problema 3.5. (continuación)

Queremos construir la matriz de proyección perspectiva  $Q$  de forma que se cumplan estos requerimientos:

1. No se recorta ningún triángulo.
2. El tamaño aparente de los objetos (proyectados en pantalla) es el mayor posible.
3. El valor del parámetro  $n$  es el mayor posible.
4. El valor del parámetro  $f$  es el menor posible.
5. Los objetos no aparecen deformados.

Con estos requerimientos, indica como calcular los valores  $l, r, t, b, n$  y  $f$  (para obtener la matriz  $Q$  de proyección), en función de  $s$  y  $(c_x, c_y, c_z)$ .

# Problemas: parámetros de matriz de proyección (3)

## Problema 3.6.

Repite el problema anterior 3.5, con exactamente los mismos requerimientos y suposiciones, excepto que ahora la escena, en lugar de estar contenida en un cubo de lado  $s$  unidades, está contenida en una esfera de radio  $r$  unidades (con centro igualmente en  $\mathbf{c}$ ).

## Problema 3.7.

Repite el problema 3.5, con exactamente los mismos requerimientos y suposiciones, excepto que ahora, en lugar de suponer que el *viewport* es cuadrado, sabemos que tiene  $w$  columnas de pixels y  $h$  filas de pixels, y no podemos suponer que  $w = h$ .

## Problema 3.8.

Repite el problema 3.5, con exactamente los mismos requerimientos y suposiciones, excepto que ahora se nos da un ángulo  $\beta$  en grados que debe ser la apertura de campo vertical de la proyección perspectiva. Para ello, ahora tenemos libertad para situar al observador en la línea paralela al eje Z que pasa por  $\mathbf{c}$ , de forma que la apertura de campo vertical sea exactamente  $\beta$ .

Indica como calcular la coordenada Z que debemos usar ahora para  $\mathbf{o}_{ec}$  (la X y la Y son las mismas que antes), de forma que se cumpla lo dicho, también indica como debemos de calcular ahora los valores de  $l, r, t, b, n$  y  $f$  (todo ello en función de  $\beta, s$  y  $\mathbf{c} = (c_x, c_y, c_z)$ ).

## Sección 2. Modelos de Iluminación, texturas y sombreado..

- 2.1. Radiación visible: características, percepción y reproducción.
- 2.2. Emisión y reflexión de la radiación.
- 2.3. Modelos computacionales simplificados.
- 2.4. Texturas
- 2.5. Métodos de sombreado para rasterización.

# 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.

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 2. Modelos de Iluminación, texturas y sombreado.

Subsección 2.1.

Radiación visible: características, percepción y reproducción..

# 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  $\lambda$  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  $\lambda$  que pasan por **p** en una determinada dirección **v** (un vector libre)

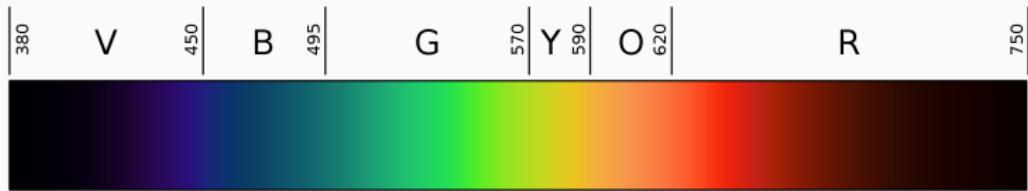
Esa energía se denomina **radiancia** y se nota como  $L(\lambda, \mathbf{p}, \mathbf{v})$ .

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](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  $\lambda$ , 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 o percibe en el cerebro (SVH) como de color rojo.
- ▶ la señal  $(0, x, 0)$  se percibe de color verde.
- ▶ la señal  $(0, 0, x)$  se percibe de color azul.

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

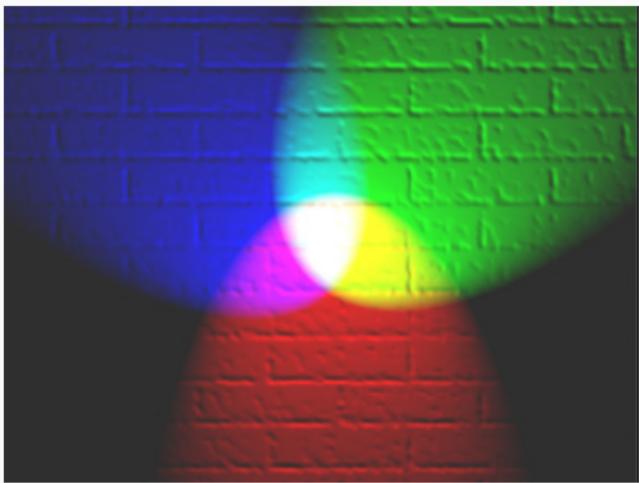
$$f(L_r) \approx (1, 0, 0) \quad f(L_g) \approx (0, 1, 0) \quad f(L_b) \approx (0, 0, 1) \quad (1)$$

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](http://en.wikipedia.org/wiki/RGB_color_model))

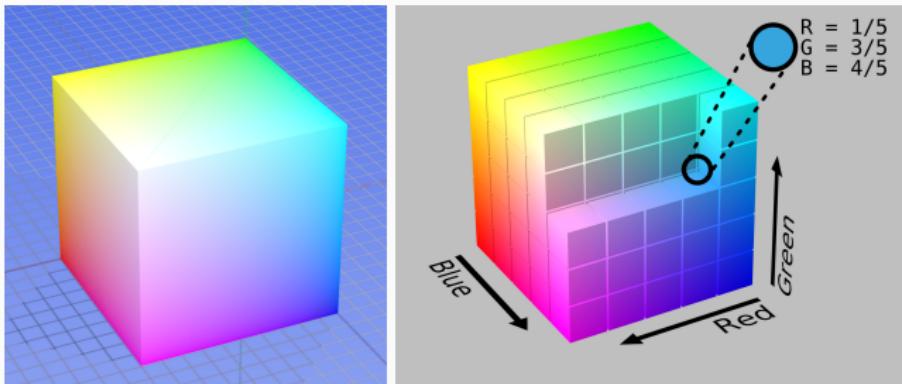
# Reproducción de ternas RGB

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](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)

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 2. Modelos de Iluminación, texturas y sombreado.

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  $\mathbf{u}_i$

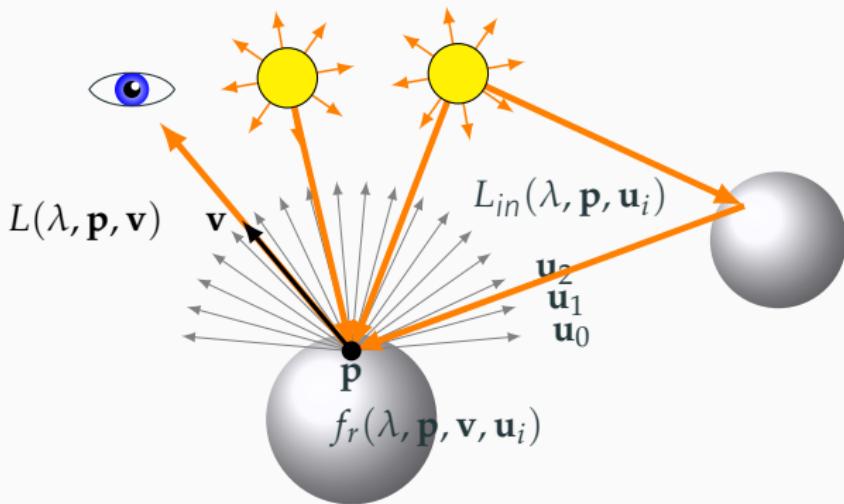
$f_r(\lambda, \mathbf{p}, \mathbf{v}, \mathbf{u}_i) \equiv$  fracción de radiancia que se refleja desde  $\mathbf{p}$  en la dirección  $\mathbf{v}$ , respecto del total incidente sobre  $\mathbf{p}$  proveniente de la dirección  $\mathbf{u}_i$  (con l.o.  $\lambda$ )

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:



Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 2. Modelos de Iluminación, texturas y sombreado.

Subsección 2.3.

Modelos computacionales simplificados..

# 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  $\lambda$  posibles, usamos el modelo RGB.

# Efecto de las simplificaciones.

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

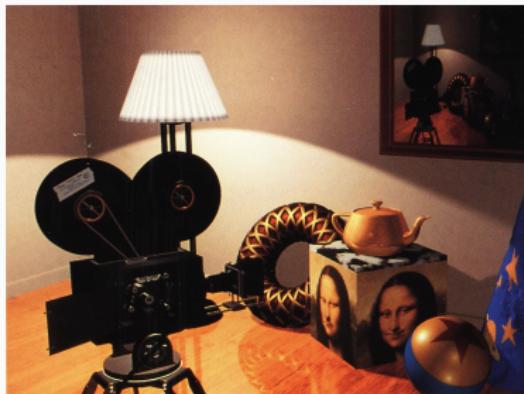


Imagen obtenida de: Computer Graphics: Principles and Practice in C (2nd Edition) Foley, van Dam, Feiner, Hughes.

## 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)

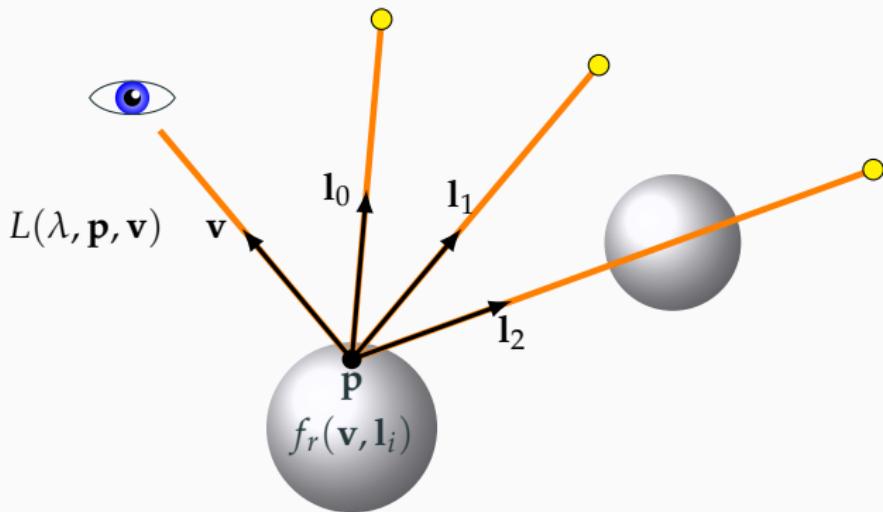
Por tanto:

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

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)

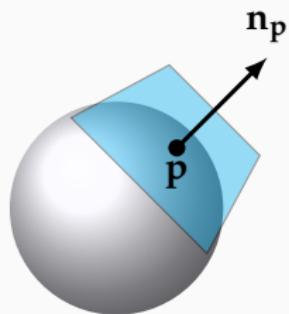
Ahora solo consideramos trayectorias desde las luminarias hacia  $\mathbf{p}$ , las luminarias se cuentan aunque la trayectoria esté bloqueada (no hay sombras arrojadas)



# 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  $f_r$



# 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:

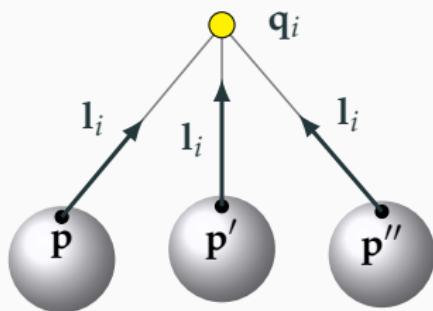
$$\mathbf{l}_i = \frac{\mathbf{q}_i - \mathbf{p}}{\|\mathbf{q}_i - \mathbf{p}\|}$$

- ▶ Fuentes de luz **direccionales**: 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

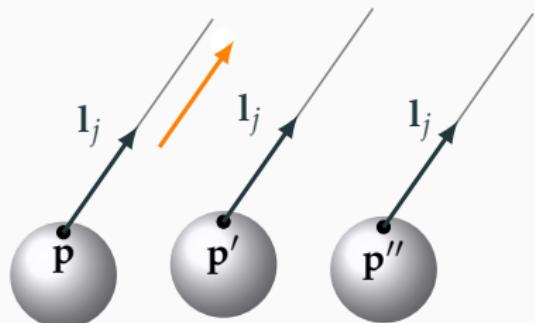
Además de esto, cada fuente de luz emite una radiancia  $S_i = (r, g, b)$  (en general no acotada).

# Posición o dirección de las luminarias

Fuente posicional ( $i$ )



Fuente direccional ( $j$ )



- ▶ Posicional: la dirección  $\mathbf{l}_i$  es distinta para cada punto  $\mathbf{p}$  considerado. Es necesario recalcularla cada vez que se evalua el MIL.
- ▶ Direccional: La dirección  $\mathbf{l}_j$  es igual para todos los puntos  $\mathbf{p}$  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:

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

# 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:

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

## 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.

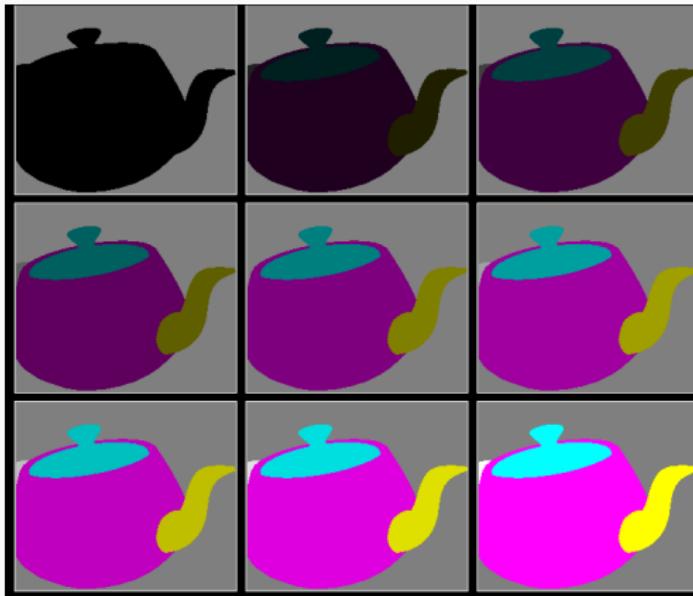
Para poder hacer esto, se asocia a cada polígono o punto una reflectividad difusa del material, una terna RGB que notamos como  $M_A(\mathbf{p})$  (con valores entre 0 y 1 pues se trata de una reflectividad), y hacemos:

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

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  $\mathbf{n_p}$  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:

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

## Orientación de la superficie:

La orientación de la superficie respecto de la fuente de luz viene determinada por el valor  $\alpha$ , 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  $\alpha$ ). 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:

$$90^\circ < \alpha$$

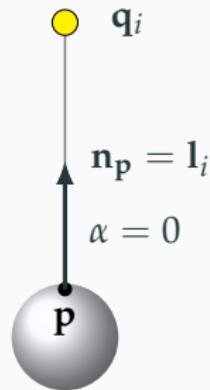
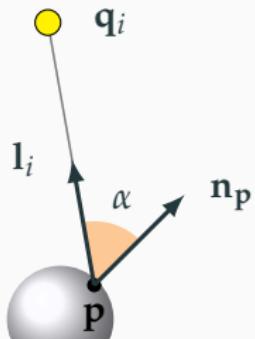
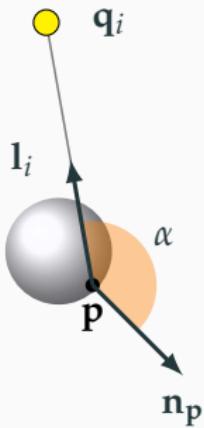
$$0 > \cos(\alpha)$$

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

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

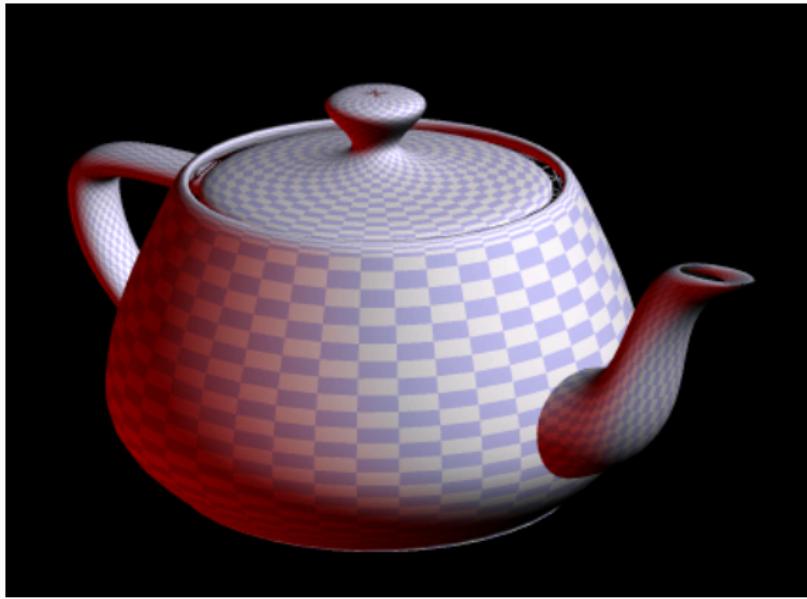
$$\alpha = 0^\circ$$

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



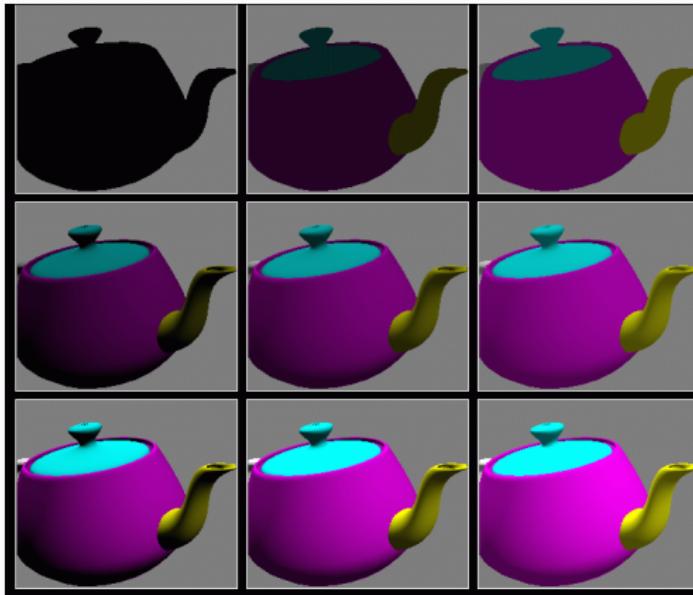
# Material difuso

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:

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

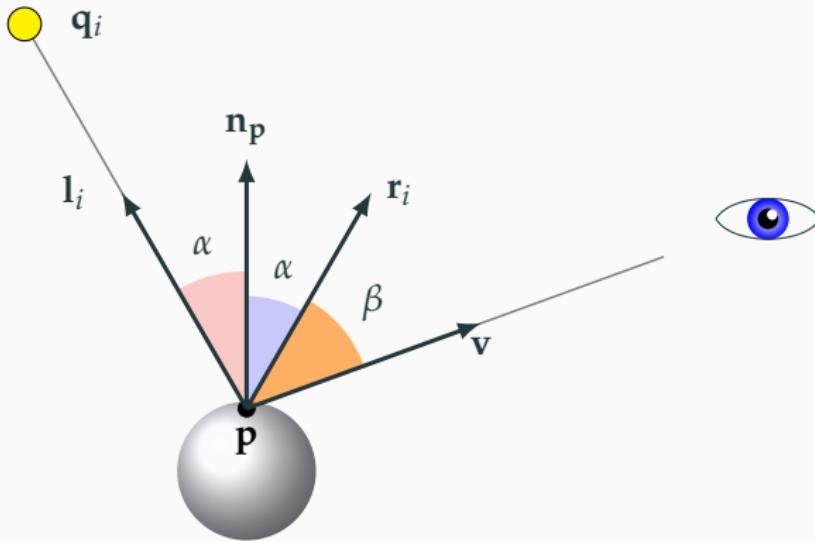
el vector  $\mathbf{r}_i$  indica la dirección desde  $\mathbf{p}$  en la cual la  $i$ -ésima fuente de luz produce el máximo brillo.

$e$   $\equiv$  **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)

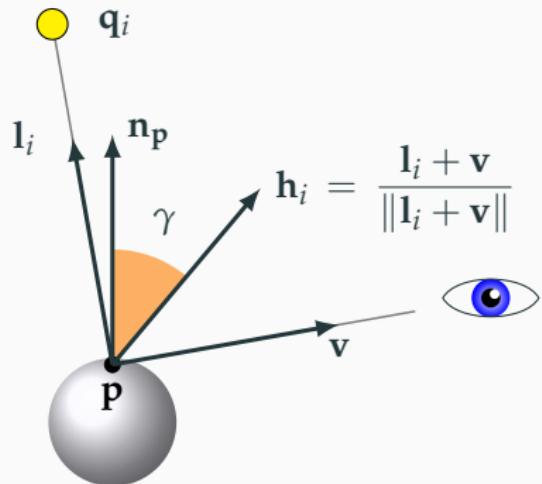
## Vectores del modelo de Phong

El valor  $\mathbf{r}_i \cdot \mathbf{v}$  es el coseno del ángulo  $\beta$  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*  $\mathbf{h}_i$  (bisectriz de  $\mathbf{l}_i$  y  $\mathbf{v}$ , normalizado). Ahora el brillo es proporcional al coseno del ángulo  $\gamma$  entre  $\mathbf{h}_i$  y  $\mathbf{n}_p$  (máximo cuando coinciden)



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

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

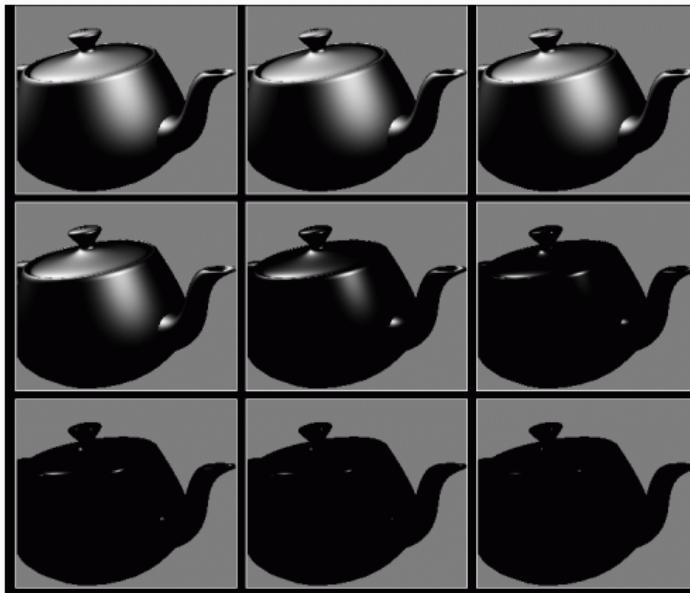
## Ejemplo de material pseudo-especular

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:



## 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:

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

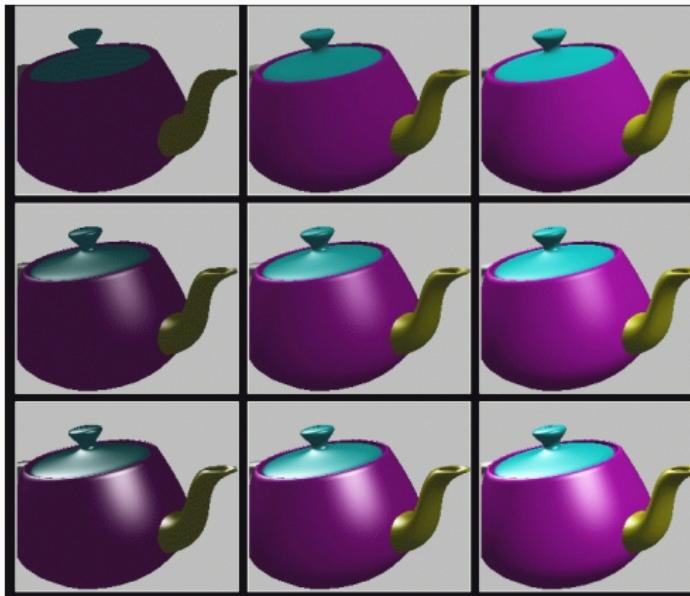
# Ejemplo de material combinado

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:



Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 2. Modelos de Iluminación, texturas y sombreado.

Subsección 2.4.

Texturas.

## Detalles a pequeña escala

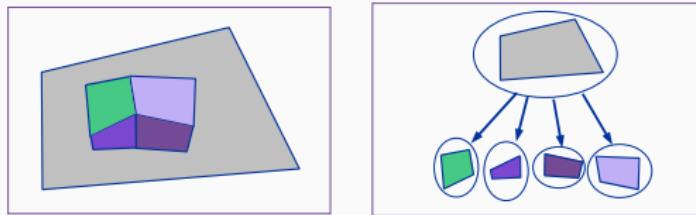
Los objetos reales presentan a veces detalles a pequeña escalar, como son manchas, defectos, motivos ornamentales, rugosidades, o, en general, cambios en el espacio de los atributos que determinan su apariencia:



- ▶ 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).

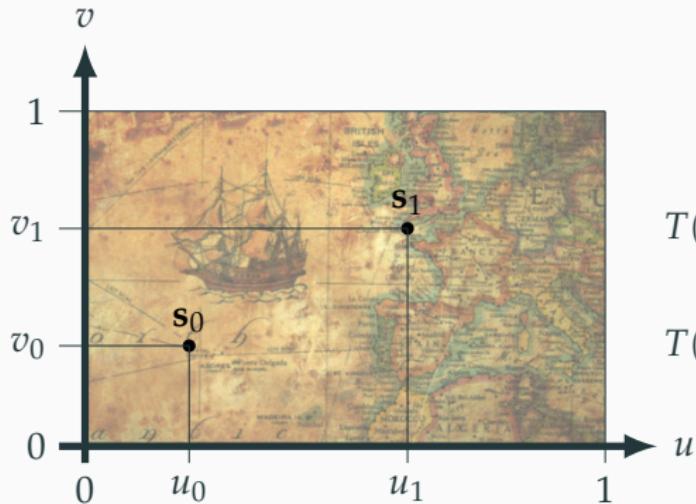
# Texturas

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.



$$T(\mathbf{s}_1) = T(u_1, v_1) = (r_1, g_1, b_1)$$

$$T(\mathbf{s}_0) = T(u_0, v_0) = (r_0, g_0, b_0)$$

# 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.)



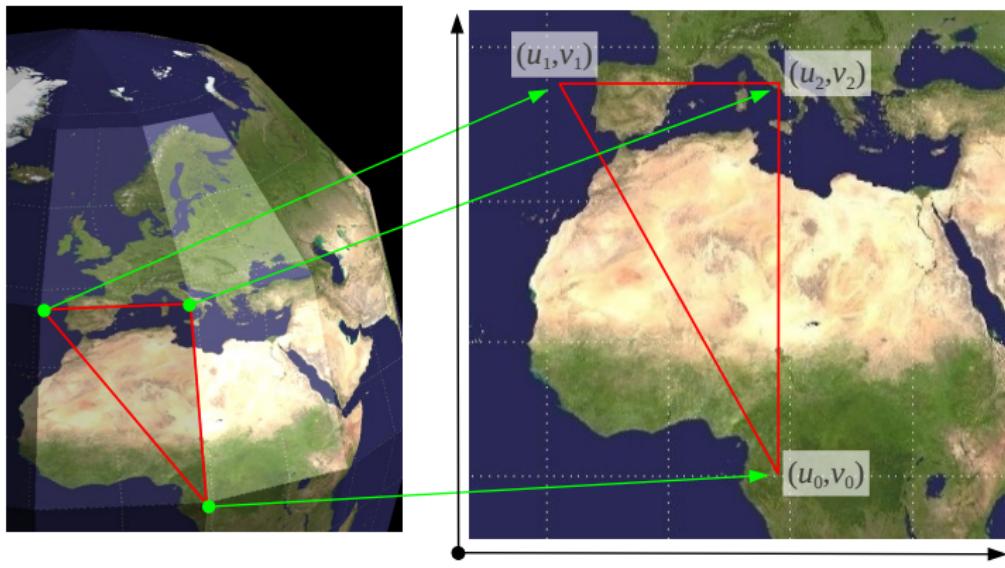
## Coordenadas de textura

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 coord. de text. se puede hacer usando:

- ▶ **Asignación explícita a vértices:** las coordenadas forman parte de la definición del modelo de escena, y son un dato de entrada al cauce gráfico, en forma de un vector o tabla de coordenadas de textura de vértices  $(v_0, u_0), (v_1, u_1), \dots (v_{n-1}, u_{n-1})$ .
  - ▶ se puede hacer manualmente en objetos sencillos, o bien
  - ▶ de forma asistida usando software para CAD.
- ▶ hace necesario realizar una interpolación de coords. de text. en el interior de los polígonos.
- ▶ **Asignación procedural:**  $f$  se implementa como un subprograma `CoordText(p)` que calcula las coordenadas de textura (para un punto  $p$  devuelve el par  $(u, v) = f(p)$  con las coords. de textura de  $p$ ).

# Ejemplo de asignación explícita.

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

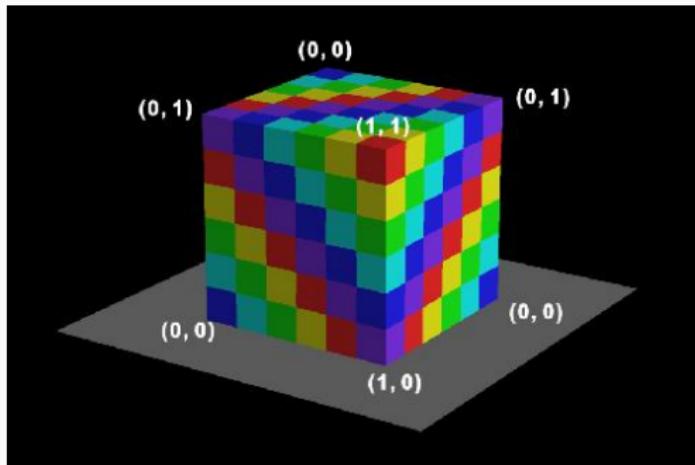
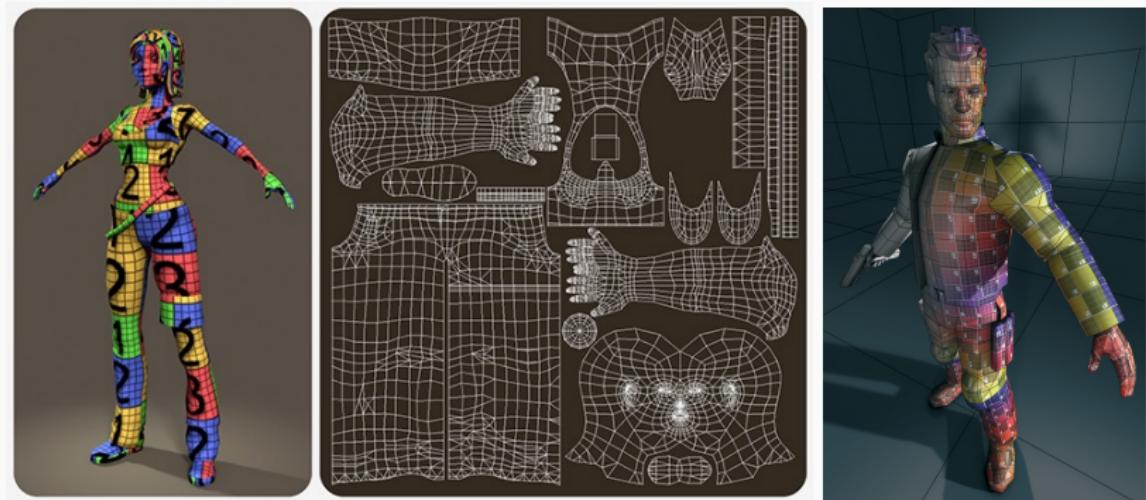


Imagen de Rosalee Wolfe, en [Teaching Texture Mapping Visually](#) (SIGGRAPH 97 Education Slide Set).

# Ejemplo de uso de herramientas CAD.

En objetos complejos es necesario el uso de herramientas CAD:



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

# Tipos de asignación procedural

Hay dos opciones:

- ▶ **Asignación procedural a vértices:** se invoca `CoordText( $v_i$ )` para calcular las coordenadas de textura en cada vértice  $v_i$ , y las coordenadas obtenidas se almacenan y después se interpolan linealmente en el interior de los polígonos de la malla.
  - ▶ Funciona de forma totalmente correcta (exacta) solo cuando  $f$  es lineal, en otro caso es una aproximación lineal a trozos.
- ▶ **Asignación procedural a puntos:** se invoca `CoordText( $p$ )` cada vez que sea necesario evaluar el MIL en un punto de la superficie  $p$ .
  - ▶ Permite exactitud incluso aunque  $f$  sea no lineal.
  - ▶ 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  $(\alpha, \beta, r)$ , los valores  $u$  y  $v$  se obtienen de  $\alpha$  y  $\beta$ .
- ▶ **Coordenadas cilíndricas** (proyección en un cilindro): el punto  $\mathbf{p}$  se expresa en coordenadas cilíndricas como una terna  $(\alpha, y, r)$ , los valores  $u$  y  $v$  se obtienen de  $\alpha$  e  $y$ .

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

## Funciones 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:

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

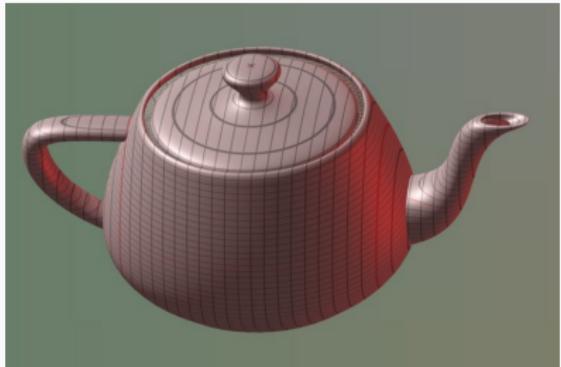
como casos particulares, y a modo de ejemplo, podemos hacer  $\mathbf{q}$  igual al origen  $(0, 0, 0)$ ,  $\mathbf{e}_u = \mathbf{x} = (1, 0, 0)$  y  $\mathbf{e}_v = \mathbf{y} = (0, 1, 0)$ , y en este caso es una proyección paralela el eje Z, sobre el plano XY (descarta la Z)

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

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

## 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).

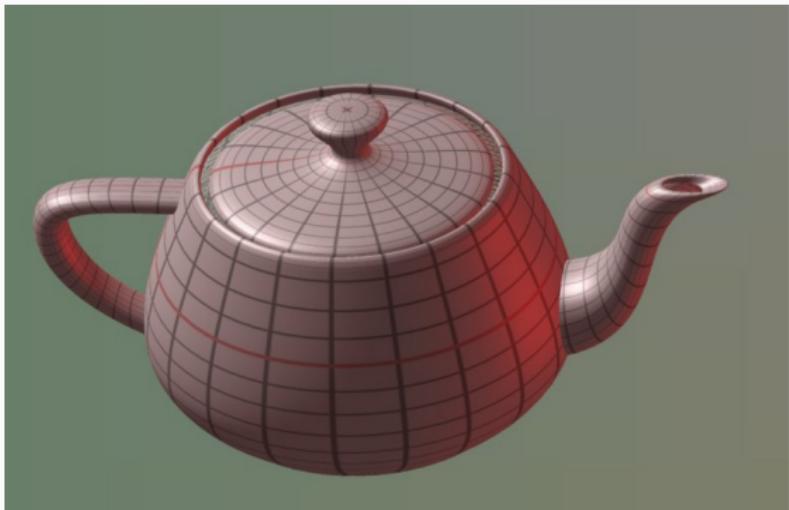
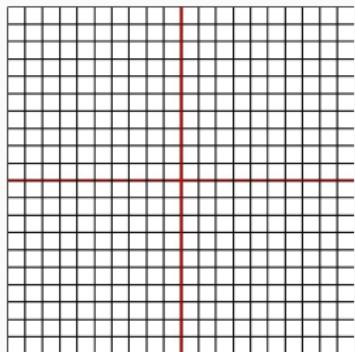
## Coordenadas paramétricas.

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 explicitamente en el programa a cada vértice sus coordenadas de textura, usando para ello sus coordenadas paramétricas.

# Coordenadas esféricicas

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

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

Hacemos:

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

- ▶ Se obtiene  $\alpha$  en el rango  $[-\pi, \pi]$  y  $\beta$  en el rango  $[-\pi/2, \pi/2]$ . 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.)

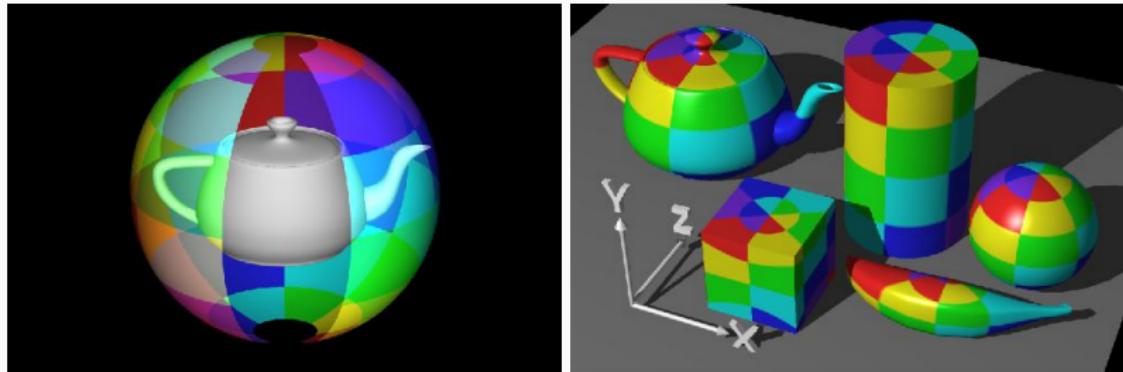


Imagen de Rosalee Wolfe, en [Teaching Texture Mapping Visually](#) (SIGGRAPH 97 Education Slide Set).

# Coordenadas cilíndricas

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

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

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

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

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

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

# Ejemplo de coordenadas cilíndricas

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

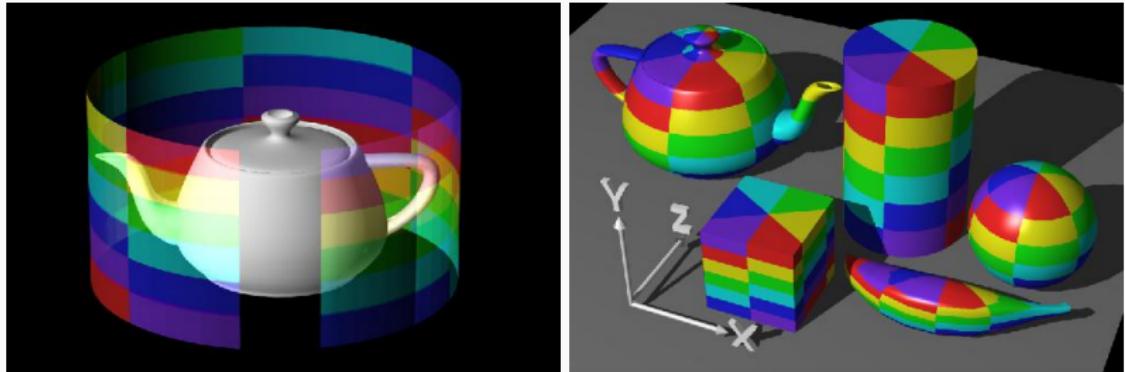


Imagen de Rosalee Wolfe, en [Teaching Texture Mapping Visually](#) (SIGGRAPH 97 Education Slide Set).

## Consulta de texels en texturas de imagen.

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

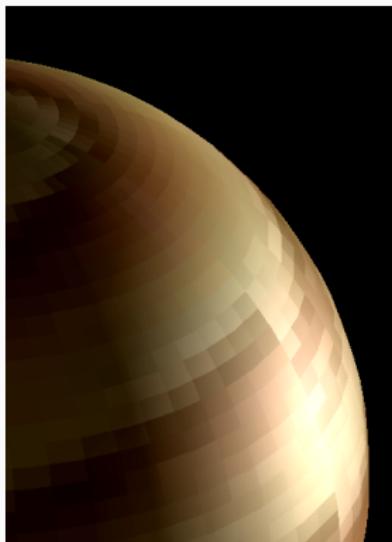
La consulta del color de la textura en un punto  $(u, v)$  puede hacerse de dos formas:

- ▶ **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

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 2. Modelos de Iluminación, texturas y sombreado.

Subsección 2.5.

Métodos de sombreado para rasterización..

## Alternativas

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

- ▶ **Sombreado plano:** (*flat shading*) una vez por cada polígono que forma el modelo, asignando el resultado (una terna RGB única) a todos los pixels donde se proyecta el polígono.
- ▶ **Sombreado de vértices:** (*smooth shading* o *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

## 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).

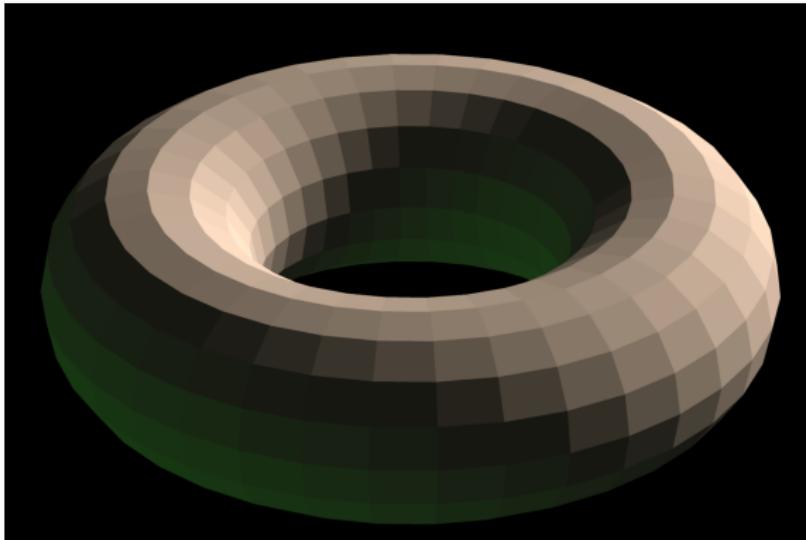
- ▶ Se debe seleccionar un punto cualquiera  $\mathbf{p}$  de cada polígono, típicamente se usa un vértice, pero podría ser cualquier otro.
- ▶ Se usa la normal al polígono  $\mathbf{n}_p$ .
- ▶ Se calcula el vector al observador  $\mathbf{v}$  en  $\mathbf{p}$ .

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).



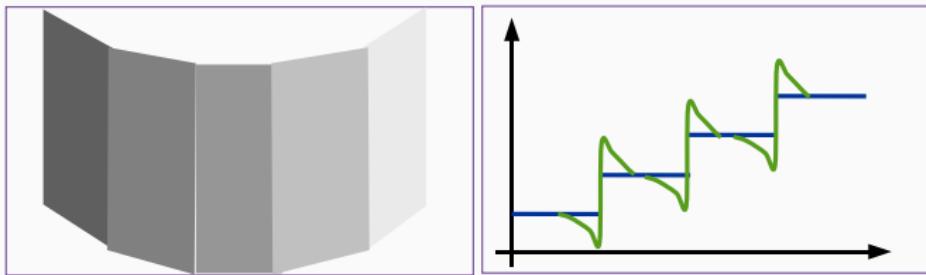
# Resultados del sombreado plano

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

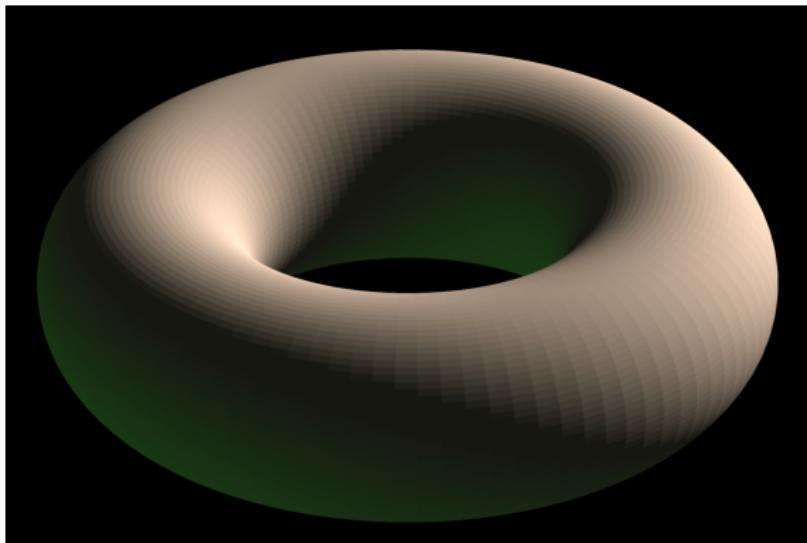
La **Bandas Mach** son una ilusión visual producida por la *inhibición lateral de las neuronas de la retina*, que es un mecanismo desarrollado evolutivamente para resaltar el contraste en aristas entre colores planos:



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:



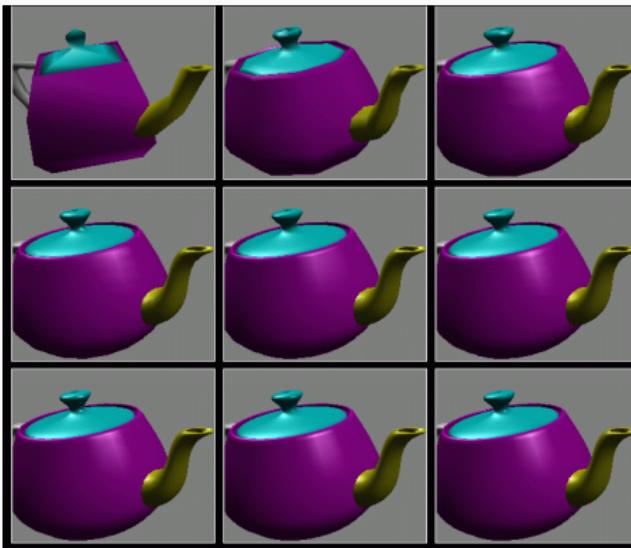
## Sombreado en los vértices

En esta modalidad (*vertex shading*), el MIL se evalua 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.

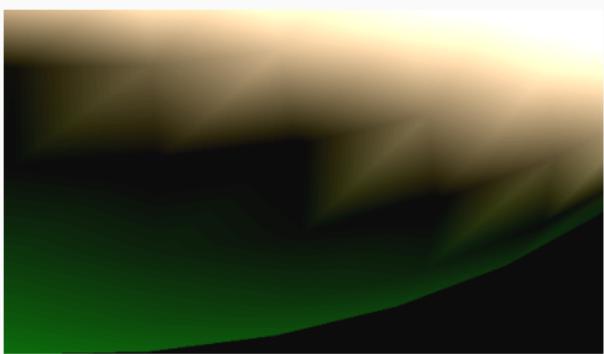
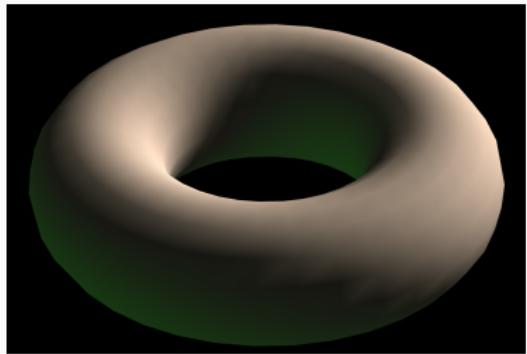
## Pérdida de zonas brillantes

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)

## 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.

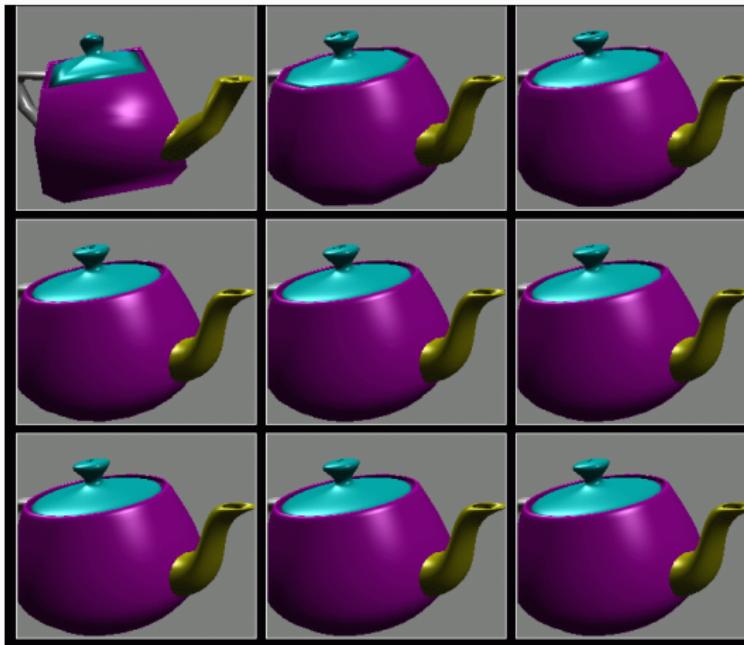
# Ejemplo de sombreado

Esta imagen se ha creado con sombreado en los pixels:



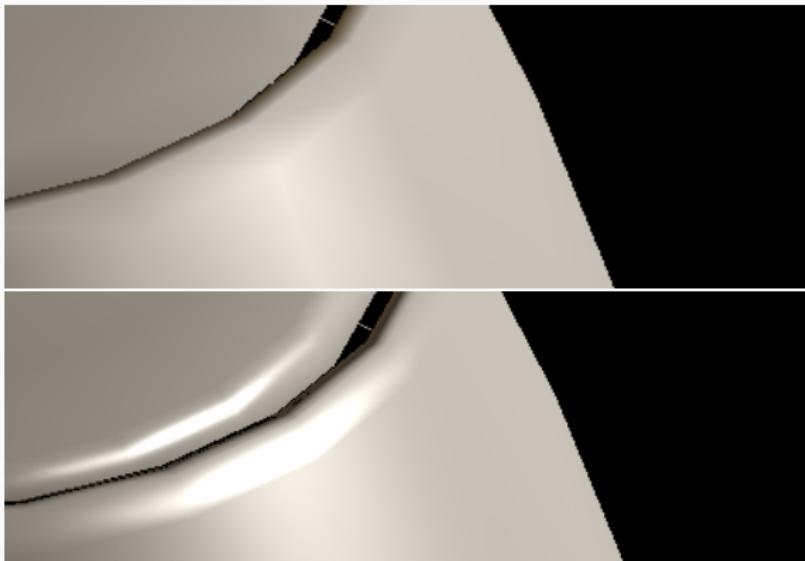
# Reproducción de zonas de brillo

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:



## Sección 3. Iluminación y texturas con el cauce fijo.

- 3.1. Introducción: activación, iluminación versus colores prefijados.
- 3.2. Definición de fuentes de luz: tipos y atributos.
- 3.3. Definición de atributos de materiales.
- 3.4. Configuración de sombreado en el cauce fijo.
- 3.5. Carga de texturas en el sistema gráfico.
- 3.6. Configuración de texturas en el cauce fijo.
- 3.7. Implementación de la clase **CauceFijo**

# 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).

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 3. Iluminación y texturas con el cauce fijo

Subsección 3.1.

Introducción: activación, iluminación versus colores prefijados..

# 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:

$M_E$   $\equiv$  emisividad del material.

$A_G$   $\equiv$  término ambiente global.

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

$e$   $\equiv$  exponente de la componente pseudo-especular.

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

$\mathbf{q}_i, \mathbf{l}_i$   $\equiv$  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:

**p** ≡ posición del vértice donde se evalua el MIL, en EC.

**v** ≡ vector normalizado desde **p** hacia el observador en EC. Si la proyección es perspectiva, se usa  $\mathbf{v} = -\mathbf{p}/\|\mathbf{p}\|$ , si es ortogonal se usa  $\mathbf{v} = (0, 0, 1)$ .

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 3. Iluminación y texturas con el cauce fijo

Subsección 3.2.

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  $M = \text{Ide}$ .
- ▶ 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  $\alpha$  y  $\beta$  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()
```

## 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 );
```

## 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
```

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 3. Iluminación y texturas con el cauce fijo

Subsección 3.3.

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

**glColor** actualiza una terna RGB (la llamamos  $C$ ) en el 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:

```
// 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 recorrellos 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:

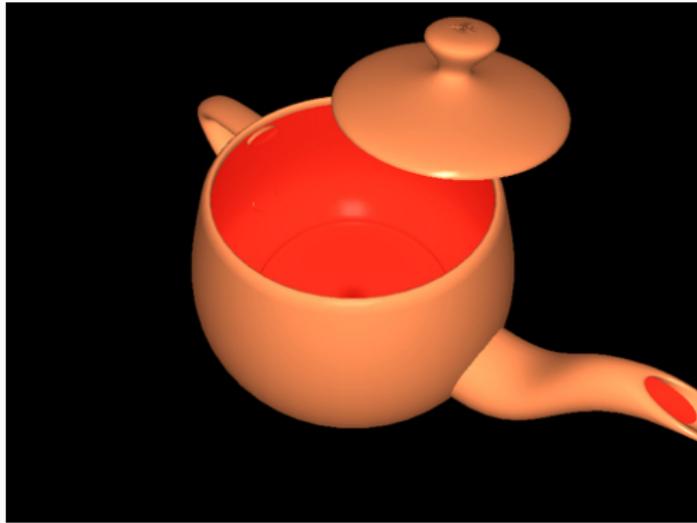


Imagen obtenida de:  editorial Packt

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 3. Iluminación y texturas con el cauce fijo

Subsección 3.4.

Configuración de sombreado en el cauce fijo..

# 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 con el cauce fijo.

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 el color especificado en la tabla de colores o 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.

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 3. Iluminación y texturas con el cauce fijo

Subsección 3.5.

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 ;  
glGenTextures( 1, &idTex ); // idTex = nuevo identificador
```

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

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

# Unidades de textura

OpenGL permite configurar distintas *unidades de textura*, y activar distintas texturas en las distintas unidades.

Para cambiar la unidad de texturas activa podemos usar:

```
// activa textura con identificador 'idTex' :  
glActiveTexture( GL_TEXTUREi );
```

- ▶ el parámetro debe ser **GL\_TEXTURE0**, **GL\_TEXTURE1**,  
**GL\_TEXTURE2**, etc....
- ▶ inicialmente la unidad activa es la unidad 0
- ▶ nosotros siempre usaremos la unidad 0

# Unidades de textura y textura activa

En el estado interno de OpenGL, por cada unidad de textura, 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:  $2^n$ ) (GLsizei)  
    alto,        // núm de filas (potencia de dos:  $2^m$ ) (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)

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 3. Iluminación y texturas con el cauce fijo

Subsección 3.6.

Configuración de texturas en el cauce fijo..

## Texturas en el cauce fijo

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. El color obtenido es  $T$ . Además, en dicho pixel se calcula un color interpolado  $C$ , a partir de **glColor** o la tabla de colores de vértice. Entonces:

- ▶ Con iluminación **desactivada**, el color del pixel puede ser:
  - ▶ igual a  $T$ : se ignora el color  $C$  (usaremos siempre esta opción)
  - ▶ igual a  $T \cdot C$ : el color  $C$  modula al color de la textura.
- ▶ Con iluminación **activada**, el color de la textura  $T$  sustituye a las reflectividades del material (usualmente a la difusa y la ambiental) en la evaluación del MIL.

todas las operaciones de texturas en el cauce fijo requieren que esta funcionalidad esté activada.

# Activación y desactivación

Las ordenes **glEnable** y **glDisable** se pueden usar para activar o desactivar, en el cauce de funcionalidad fija, toda la funcionalidad de OpenGL relacionada con las texturas

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

(las texturas están inicialmente desactivadas en el cauce fijo).

## Parámetros 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):

```
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 ) ;
```

- ▶ No es posible usar para esto las coordenadas del mundo en el cauce fijo.

(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 ) ;
```

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 3. Iluminación y texturas con el cauce fijo

Subsección 3.7.

Implementación de la clase **CauceFijo**.

# Iluminación usando CauceFijo

La clase **CauceFijo** implementa el interfaz **Cauce** para facilitar implementaciones portables:

El método **fijarEvalMIL** activa o desactiva la iluminación:

```
void CauceFijo::fijarEvalMIL( const bool nue_eval_mil )  
{  
    if ( nue_eval_mil ) glEnable( GL_LIGHTING );  
    else                 glDisable( GL_LIGHTING );  
}
```

El método **fijarModoSombrPlano** permite activar sombreado plano o desactivarlo (activa modo suave)

```
void CauceFijo::fijarModoSombrPlano  
    ( const bool nue_sombr_plano )  
{  
    const GLenum modo = nue_sombr_plano ? GL_FLAT : GL_SMOOTH ;  
    glShadeModel( modo );  
}
```

## Fijar parámetros de las fuentes (1/2)

El método **fijarFuentesLuz** se invoca una vez por cuadro:

```
void CauceFijo::fijarFuentesLuz
    ( const std::vector<Tuple3f> & color,
      const std::vector<Tuple4f> & pos_dir_wc )
{
    const GLfloat color_negro[4] = { 0.0, 0.0, 0.0, 1.0 };

    // (1) fijar parámetros de iluminación y materiales
    glEnable( GL_NORMALIZE );
    glEnable( GL_COLOR_MATERIAL );
    glColorMaterial( GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE );

    glLightModeli( GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE );
    glLightModeli( GL_LIGHT_MODEL_COLOR_CONTROL,
                    GL_SEPARATE_SPECULAR_COLOR );
    glLightModeli( GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE );
    glLightModelfv( GL_LIGHT_MODEL_AMBIENT, color_negro );
    ....
```

## Fijar parámetros de las fuentes (2/2)

```
.....  
// (2) habilitar fuentes, desde 0 hasta color.size()-1,  
// para cada una de ellas, se envía su posición (el color de la fuente se enviará  
// cuando se especifiquen los parámetros del material, más adelante)  
for( unsigned i = 0 ; i < color.size() ; i++ )  
{ glEnable( GL_LIGHT0+i );  
    glLightfv( GL_LIGHT0+i, GL_POSITION, pos_dir_wc[i] );  
}  
  
// (3) deshabilitar restantes fuentes, hasta 'maxNumFuentesLuz' (no incluido).  
for( unsigned i = color.size() ; i < maxNumFuentesLuz(); i++ )  
    glDisable( GL_LIGHT0 + i );  
  
// (4) registrar los colores de las fuentes de luz, se usarán más adelante cuando  
// se active el material y se conozcan todos los parámetros del M.I.L.  
colores_fuentes.clear();  
for( unsigned i = 0 ; i < color.size() ; i++ )  
    colores_fuentes.push_back( color[i] );
```

# Fijar parámetros del MIL

El método **fijarParamsMIL** configura las reflectividades del MIL:

```
void CauceFijo::fijarParamsMIL( const Tupla3f &amb,
                                const Tupla3f &dif, const Tupla3f &esp,
                                const float exp )
{
    // definir emisividad (nula), comp. especular (1,1,1) y exponente
    glMaterialfv( GL_FRONT_AND_BACK, GL_EMISSION, Tupla4f{0,0,0,1});
    glMaterialfv( GL_FRONT_AND_BACK, GL_SPECULAR, Tupla4f{1,1,1,1});
    glMaterialf ( GL_FRONT_AND_BACK, GL_SHININESS, exp );
    // definir los colores de las fuentes de luz en OpenGL: son el producto de
    // los colores de las fuentes que hay registradas por las reflectividades
    for( unsigned i = 0 ; i < colores_fuentes.size() ; i++ )
    { glLightfv( GL_LIGHT0+i, GL_AMBIENT, pro(amb,colores_fuentes[i]));
      glLightfv( GL_LIGHT0+i, GL_DIFFUSE, pro(dif,colores_fuentes[i]));
      glLightfv( GL_LIGHT0+i, GL_SPECULAR, pro(esp,colores_fuentes[i]));
    }
}
```

**pro** acepta dos tuplas y devuelve la tupla producto comp. a comp.

## Habilitar una textura

El método **fijarEvalText** permite habilitar (o deshabilitar) las texturas, y en el primer caso requiere el identificador de la textura a activar:

```
void CauceFijo::fijarEvalText( const bool nue_eval_text,
                               const int   nue_text_id  )
{
    if ( nue_eval_text )
    {
        glEnable( GL_TEXTURE_2D );
        glBindTexture( GL_TEXTURE_2D, nue_text_id );
        glColor3f( 1.0, 1.0, 1.0 );
    }
    else
        glDisable( GL_TEXTURE_2D );
}
```

## Fijar parámetros de generación de cc.t. (1/2)

El método **fijarTipoGCT** fija los parámetros relacionados con generación automática de coordenadas de textura (generación activada sí/no, tipo de generación, coeficientes):

```
void CauceFijo::fijarTipoGCT( const int nue_tipo_gct,
                               const float * coefs_s, const float * coefs_t )
{
    switch ( nue_tipo_gct )
    {
        case 0 : // ==> generación de cc.t. desactivada
            glDisable( GL_TEXTURE_GEN_S );
            glDisable( GL_TEXTURE_GEN_T );
            break ;
        .....
    }
}
```

## Fijar parámetros de generación de cc.t. (2/2)

```
.....  
  
case 1 : // ==> generación activada, en coords de objeto.  
    glEnable( GL_TEXTURE_GEN_S );  
    glEnable( GL_TEXTURE_GEN_T );  
    glTexGeni( GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR );  
    glTexGeni( GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR );  
    glTexGenfv( GL_S, GL_OBJECT_PLANE, coefs_s ) ;  
    glTexGenfv( GL_T, GL_OBJECT_PLANE, coefs_t ) ;  
    break ;  
  
case 2 : // ==> generación activada, en coordenadas de cámara  
    glEnable( GL_TEXTURE_GEN_S );  
    glEnable( GL_TEXTURE_GEN_T );  
    glTexGeni( GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR );  
    glTexGeni( GL_T, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR );  
    glTexGenfv( GL_S, GL_EYE_PLANE, coefs_s ) ;  
    glTexGenfv( GL_T, GL_EYE_PLANE, coefs_t ) ;  
    break ;  
}  
}
```

## Sección 4.

### Iluminación y texturas en el cauce programable.

- 4.1. Estructura del *vertex shader*.
- 4.2. Estructura del *fragment shader*.
- 4.3. Implementación de la clase **CauceProgramable**.

## Illum. y texturas en el cauce programable

En OpenGL moderno la evaluación del Modelo de Iluminación Local (MIL) sencillo y la consulta de texturas se hacen principalmente en los *shaders*

- ▶ En el *vertex shader* se hace:
  - ▶ Generación de coordenadas de textura (si está activada, en otro caso se pasan las coords. de textura del vértice)
- ▶ En el *fragment shader* se hace:
  - ▶ Obtención del vector normal y el vector al observador
  - ▶ Obtención de los parámetros de las fuentes de luz.
  - ▶ Obtención de los parámetros del material.
  - ▶ Consulta de la textura
  - ▶ Evaluación del MIL.

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 4. Iluminación y texturas en el cauce programable

Subsección 4.1.

Estructura del *vertex shader*..

# Generación de coords. de textura en el Vertex Shader

Aquí vemos los parámetros de entrada y las variables relacionadas:

```
.....
// parámetros de entrada (uniforms)
uniform int eval_text; // 0 -> no evaluar texturas, 1 -> sí eval. texturas
uniform int tipo_gct; // tipo gen.cc.tt. (0=desact, 1=objeto, 2=camara)
uniform vec4 coefs_s; // coefficientes para G.CC.TT. (coordenada 's')
uniform vec4 coefs_t; // coefficientes para G.CC.TT. (coordenada 't')
// variables calculadas en el vertex shader (varying)
varying vec4 var_posic_ec; // posicion del vert. en EC (coords.cámara)
varying vec2 var_coord_text; // coords. de textura del vértice
// función de cálculo de las coords. de textura
vec2 CoordsTextura() { .... }
...
void main()
{ ....
    var_posic_ec = ..... ; // calcula posición en EC
    var_coord_text = CoordsTextura(); // calcula coords. de text.
    gl_Position = ..... ; // calcula posición en CC
}
```

# Generación de coordenadas de textura

La función **CoordsTextura** se encarga de devolver las coordenadas de textura según los disntintos *uniform* y la posición en coordenadas de cámara o de objeto:

```
vec2 CoordsTextura() // calcula las coordenadas de textura
{
    if ( eval_text == 0 )          // si no se están evaluando las cc.t.
        return vec2( 0.0, 0.0 );  // devuelve cualquier cosa
    if ( tipo_gct == 0 )          // texturas activadas, generación desact:
        return gl_MultiTexCoord0.st; // devuelve las cc.t. del vértice

    vec4 pos_ver ;
    if ( tipo_gct == 1 )          // generacion en coordenadas de objeto
        pos_ver = gl_Vertex;     // usar las coords originales (objeto)
    else                          // generacion en coords de cámara
        pos_ver = var_posic_ec; // usar las coordenadas de cámara

    return vec2( dot(pos_ver,coefs_s), dot(pos_ver,coefs_t) );
}
```

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 4. Iluminación y texturas en el cauce programable

Subsección 4.2.

Estructura del *fragment shader*..

## Evaluación del MIL en el *fragment shader*. Parámetros.

El *fragment shader* se encarga de evaluar el Modelo de Iluminación Local (MIL) sencillo que hemos introducido antes (el mismo que el cauce fijo). La evaluación produce como resultado el color del pixel.

Usa estos parámetros:

- ▶ Un valor lógico que indica si se debe evaluar el MIL o no. Otro que indica si se debe usar la textura o no.
- ▶ Vector normal (normalizado), el vector al observador (posición en EC negada), coordenadas de textura.
- ▶ El número de fuentes de luz activas.
- ▶ Parámetros de las fuentes de luz: para cada una, su posición o dirección y el color, (en sendos arrays).
- ▶ Parámetros del material: reflectividades difusa, ambiental y especular, color y exponente de brillo.
- ▶ Textura activa actualmente, coordenadas de textura calculadas en el fragment shader.

# Eval. MIL: parámetros de entrada *uniform*

Son los siguientes:

```
// MIL y texturas
uniform int      eval_mil;    // evaluar el MIL: sí (1) o no (0)
uniform int      sombr_plano; // Sombreado: 0 -> Gouroud, 1 -> plano
uniform int      eval_text;   // usar textura sí (1) o no (0)
uniform sampler2D tex;       // textura activa en la unidad 0

// datos del material activo
uniform vec3     mil_ka;     // reflectividad ambiental ( $M_a$ )
uniform vec3     mil_kd;     // reflectividad difusa
uniform vec3     mil_ks;     // reflectividad pseudo-especular
uniform float    mil_exp;    // exponente de brillo

// datos de las fuentes de luz activas
const int max_num_luces = 8 ; // máx. númer. de fuentes de luz activas
uniform int      num_luces;  // número de luces activas
uniform vec4     pos_dir_luz_ec[max_num_luces]; // posic./direc.
uniform vec3     color_luz[max_num_luces] ;      // colores
```

## Evaluación del MIL: parámetros de entrada *varying*

En el *fragment shader*: las variables *varying* son parámetros de entrada, con datos interpolados desde el *vertex shader*. Usaremos las siguientes declaraciones:

```
varying vec4 var_posic_ec ;      // posicion del punto  
varying vec3 var_normal_ec;     // normal del punto (no norm.)  
varying vec3 var_vec_obs_ec ;   // vector hacia el observador  
varying vec4 var_color;         // color de la primitiva en el pixel  
varying vec2 var_coord_text;    // coordenadas de textura
```

La posición, la normal (no normalizada), y el vector al observador se expresan en coordenadas de cámara (EC)

# Vector hacia el observador y normal

Se calculan usando estas funciones:

```
vec3 VectorHaciaObs() // devuelve el vector hacia el observador normalizado
{
    return normalize( var_vec_obs_ec );
}

vec3 NormalTriangulo() // calcula la normal al triángulo
{
    vec4 tx = dFdx( var_posic_ec ); // tangente al tri. en horizontal
        ty = dFdy( var_posic_ec ); // tangente al tri. en vertical
    return normalize( cross( tx.xyz, ty.xyz ) ); // producto vectorial
}

vec3 Normal() // devuelve la normal que se debe usar en el MIL
{
    vec3 nor = (sombr_plano == 1) ? NormalTriangulo()
                                    : normalize( var_normal_ec );
    return gl_FrontFacing ? nor // es una cara delantera
                          : -nor; // es una cara trasera
}
```

# Vector hacia una fuente de luz

El vector hacia la  $i$ -ñesima fuente de luz (normalizado), en coordenadas de cámara, se calcula en función de la componente W de la posición o dirección a dicha fuente.

Usamos esta función:

```
vec3 VectorHaciaFuente( int i )
{
    return ( pos_dir_luz_ec[i].w == 1.0 ) ?
        normalize( pos_dir_luz_ec[i].xyz - var_posic_ec.xyz ) :
        normalize( pos_dir_luz_ec[i].xyz ) ;
}
```

# Evaluación del MIL

```
vec3 EvalMIL( vec3 col ) // col ≡ color interpolado o de textura
{
    vec3 n = Normal(), // normal (EC)
        v = VectorHaciaObs(), // vector hacia observador (EC)
        s = vec3( 0.0, 0.0, 0.0 ); // suma color debido a cada fuente

    for( int i = 0 ; i < num_luces ; i++ ) // para cada fuente
    {
        vec3 l = VectorHaciaFuente( i ); // vector hacia fuente (EC)
        float nl = dot( n, l ); // coseno de ángulo entre n y l

        if ( 0.0 < nl ) // si normal está de cara a la luz
        {
            float hn = max( 0.0, dot( n, normalize( l+v ) ) );
            vec3 c = mil_kd*col*nl + mil_ks*pow(hn,mil_exp);
            s = s + c*color_luz[i]; // sumar componentes difusa + especular
        }
        s = s + mil_ka*col*color_luz[i]; // sumar componente ambiental
    }
    return s ; // devolver la suma de todas las fuentes
}
```

# Función principal

La función **main** calcula el color del pixel en **gl\_FragColor**, invocando **EvalMIL** si es necesario:

```
void main()
{
    vec4 color_obj = ( eval_text == 1 ) ?
        texture2D( tex, var_coord_text ) : // color de textura
        var_color ;                      // color de la primitva

    gl_FragColor = ( eval_mil == 1 ) ?
        vec4( EvalMIL( color_obj.rgb ), 1.0 ) : // ilum. activada
        color_obj ;                          // ilum desactivada
}
```

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 4. Iluminación y texturas en el cauce programable

Subsección 4.3.

Implementación de la clase **CauceProgramable..**

# Iluminación usando CauceProgramable

La clase **CauceProgramable** implementa el interfaz **Cauce** para facilitar el uso de los shaders.

El método **fijarEvalMIL** actualiza el uniform **eval\_mil**:

```
void CauceProgramable::fijarEvalMIL( const bool nue_eval_mil )  
{  
    glUseProgram( id_prog );  
    glUniform1i( loc_eval_mil, eval_mil ? 1 : 0 );  
}
```

El método **fijarModoSombrPlano** actualiza **sombr\_plano**:

```
void CauceProgramable::fijarModoSombrPlano  
    ( const bool nue_sombr_plano )  
{  
    glUseProgram( id_prog );  
    glUniform1i( loc_sombr_plano, sombr_plano ? 1 : 0 );  
}
```

# Parámetros de las fuentes

El método **fijarFuentesLuz** se invoca una vez por cuadro:

```
void CauceProgramable::fijarFuentesLuz
    ( const std::vector<Tuple3f> & color,
      const std::vector<Tuple4f> & pos_dir_wc )
{
    const unsigned nl = color.size(); // número de fuentes
    std::vector<Tuple4f> pos_dir_ec; // vector con pos./dir.
    for( unsigned i = 0 ; i < nl ; i++ ) // para cada fuente
        pos_dir_ec.push_back( mat_vista * pos_dir_wc[i] ); // añadir p/d
    // copiar vectores en los uniforms:
    glUseProgram( id_prog );
    glUniform1i( loc_num_luces, nl );
    glUniform3fv( loc_color_luz, nl, (const float *)color.data() );
    glUniform4fv( loc_pos_dir_luz_ec, nl,
                  (const float *)pos_dir_ec.data() );
}
```

transforma las direcciones o posiciones por la matriz de vista actual, así que las interpreta en coords. de mundo y produce E.C.

## Parámetros del MIL (material)

El método **fijarParamsMIL** fija los parámetros del material  
(reflectividades ambiente, difusa y especular, y el exponente)

```
void CauceProgramable::fijarParamsMIL
    ( const Tupla3f & mil_ka, const Tupla3f & mil_kd,
      const Tupla3f & mil_ks, const float exp )
{
    glUseProgram( id_prog );
    glUniform3fv( loc_mil_ka, 1, mil_ka );
    glUniform3fv( loc_mil_kd, 1, mil_kd );
    glUniform3fv( loc_mil_ks, 1, mil_ks );
    glUniform1f ( loc_mil_exp, exp );
}
```

se debe invocar cada vez que se quiera activar un nuevo material

## Habilitar una textura

El método **fijarEvalText** permite habilitar (o deshabilitar) las texturas, y en el primer caso requiere el identificador de la textura a activar:

```
void CauceProgramable::fijarEvalText( const bool nue_eval_text,
                                      const int  nue_text_id )
{
    glUseProgram( id_prog );
    if ( eval_text ) // activar
    {
        glBindTexture( GL_TEXTURE0 );
        glActiveTexture( GL_TEXTURE_2D, nue_text_id );
        glUniform1i( loc_eval_text, 1 );
        glColor3f( 1.0, 1.0, 1.0 );
    }
    else // desactivar
    {
        glUniform1i( loc_eval_text, 0 );
    }
}
```

## Fijar parámetros de generación de cc.t.

El método **fijarTipoGCT** fija los parámetros relacionados con generación automática de coordenadas de textura (generación activada sí/no, tipo de generación, coeficientes):

```
void CauceProgramable::fijarTipoGCT( const int nue_tipo_gct,
                                      const float * coefs_s, const float * coefs_t )
{
    glUniform1i( loc_tipo_gct, tipo_gct );

    if ( tipo_gct == 1 || tipo_gct == 2 )
    { glUniform4fv( loc_coefs_s, 1, coefs_s );
      glUniform4fv( loc_coefs_t, 1, coefs_t );
    }
}
```

## Sección 5.

### Representación de materiales, texturas y fuentes..

- 5.1. Las clases **Textura** y **Material**
- 5.2. Fuentes de luz. La clase **ColeccionFuentes**
- 5.3. Materiales en el grafo de escena

# Clases relacionadas con iluminación y texturas.

En esta sección veremos como representar en una aplicación los diversos parámetros relacionados con la iluminación y texturas:

- ▶ Clase **Textura**: incluye un puntero a los texels en RAM, y los parámetros de generación de coords. de text.
- ▶ Clase **Material**: incluye los parámetros del MIL (reflectividades ambiente, difusa y especular, exponente de brillo), y opcionalmente un puntero a una instancia de una textura.
- ▶ Clase **FuenteLuz** y **ColFuentesLuz**: parámetros que definen cada fuente de luz (posición o dirección, color), y conjunto de fuentes de luz a usar en una escena.
- ▶ Clase **NodoGrafoEscena**: en esta clase se podrán incluir entradas de tipo material.
- ▶ Clase **Escena**: contendrá una colección de fuentes, y un material inicial.

## Activación

Las instancias de estas las clases **Textura**, **Material** y **ColFuentesLuz** incorporan un método para *activarlas*:

- ▶ La **activación** es el proceso por el cual el cauce se configura para que en la siguientes operaciones de visualización se use una textura, un material, o una colección de fuentes.
- ▶ En todos los casos se usa un método llamado **activar**, que tiene como único parámetro una referencia al cauce actual.
- ▶ Veremos como se implementa la activación usando el interfaz de la clase **Cauce**.

Para las texturas, materiales y colecciones de fuentes, se definen clases derivadas con constructores que implementan distintas variantes.

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 5. Representación de materiales, texturas y fuentes.

Subsección 5.1.

Las clases Textura y Material.

# Clase Textura

La clase textura se declara así:

```
class Textura
{
public:
    Textura( const std::string & nombreArchivoJPG ) ; // constructor
    ~Textura() ; // libera memoria ocupada por texels
    void activar( Cauce & cauce ) ; // activación

protected:
    void enviar() ; // envia la imagen a la GPU (gluBuild2DMipmaps)
    unsigned char * imagen = nullptr; // texels en mem. dinámica
    bool enviada = false; // true si enviada
    GLuint ident_textura = -1 ; // 'nombre' o identif. de text.
    unsigned ancho = 0, // número de columnas
            alto = 0 ; // número de filas de la imagen
    ModoGenCT modo_gen_ct = mgct_desactivada ; // modo gen. cc.t.
    float coefs_s[4] = {1.0,0.0,0.0,0.0}, // coeficientes (S)
          coefs_t[4] = {0.0,1.0,0.0,0.0}; // coeficientes (T)
};
```

# Clase Material

Encapsula una textura y los cuatro parámetros del MIL (reales)

```
class Material
{
public:
    Material() {} ; // usa valores por defecto (en las declaraciones)
    // constructores (sin textura y con textura)
    Material( const float p_k_amb, const float p_k_dif,
               const float p_k_pse, const float p_exp_pse );
    Material( Textura * p_textura,
               const float p_k_amb, const float p_k_dif,
               const float p_k_pse, const float p_exp_pse );
    void activar( Cauce & cauce ) ; // activación
    ~Material() ; // libera la textura, si hay alguna
protected:
    Textura * textura = nullptr; // textura, si != nullptr
    float k_amb     = 0.2f,   // coeficiente de reflexión ambiente
          k_dif     = 0.8f,   // coeficiente de reflexión difusa
          k_pse     = 0.0f,   // coeficiente de reflexión pseudo-especular
          exp_pse   = 0.0f;   // exponente de brillo para reflexion pseudo-especular
};
```

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 5. Representación de materiales, texturas y fuentes.

Subsección 5.2.

Fuentes de luz. La clase **ColecciónFuentes**.

# Clase FuenteLuz

Ejemplo de fuente de luz direccional y manipulable interactivamente:

```
class FuenteLuz
{
public:
    FuenteLuz( GLfloat p_longi_ini, GLfloat p_lati_ini,
                const Tupla3f & p_color ) ;
    bool gestionarEventoTeclaEspecial( int key ); // procesa L+tecla
    void actualizarLongi( const float incre ); // actualiza longi.
    void actualizarLati( const float incre ); // actualiza lati.

protected:
    float    longi,           // longitud actual en WC (grados, 0 a 360)
            lati ;          // latitud actual en WC (grados, -90 a +90)
    float    longi_ini,        // valor inicial de 'longi'
            lati_ini,        // valor inicial de 'lati'
    Tupla3f  col_ambiente,   // color de la fuente para componente ambiental
            col_difuso,     // color de la fuente para componente difusa
            col_especular; // color de la fuente para componente especular
    friend class ColFuentesLuz ;
};
```

# Clase ColFuentesLuz

Colección de fuentes manipulable (con una fuente actual)

```
class ColFuentesLuz
{
public:
    ColFuentesLuz() ; // crea la colección vacía
    ~ColFuentesLuz() ; // libera memoria ocu
    void insertar( FuenteLuz * pf ) ; // inserta nueva fuente (copia ptr)
    void activar( Cauce & cauce ) ; // activación fuentes e ilum.
    void sigAntFuente( int d ) ; // cambiar fuente act. (d==+1 o -1)
    FuenteLuz * fuenteLuzActual() ; // devuelve (puntero a) fuente act.

private:
    std::vector<FuenteLuz *> vpf ; // vector de punteros a fuentes
    GLint max_num_fuentes = 0 ; // máximo número de fuentes
    unsigned i_fuente_actual = 0 ; // índice de fuente actual
};
```

# Visualización con materiales y luces

El código de la aplicación, para visualizar una escena con iluminación activada, debe:

- ▶ Antes de visualizar los objetos:
  1. activar la evaluación del MIL
  2. activar una colección de fuentes de luz
  3. activar un material por defecto
- ▶ Durante la visualización, para visualizar un objeto que tiene un material específico, se debe
  1. guardar un puntero al material activado antes
  2. activar el material específico
  3. visualizar el objeto
  4. activar el material anterior

Para recordar un puntero al material activo, podemos usar el contexto de visualización.

## Ejemplo de visualización

Si queremos visualizar un objeto (**objeto**) usando un puntero a un material (**material**), podemos usar la variable **material\_act** del contexto de visualización (**cv**):

```
// (1) registrar material activo actualmente (suponemos que no es nullptr)
Material * material_previo = cv.material_act ;
// (2) activar material actual:
cv.material_act = material ;
cv.material_act->activar( cv.cauce_act );
// (3) visualizar el objeto
objeto->visualizarGL( cv );
// (4) reactivar material activo al inicio
cv.material_act = material_previo ;
cv.material_act->activar( cv.cauce_act );
```

Antes de visualizar una escena, debemos de activar algún material por defecto y guardar el puntero en el contexto de visualización.

Informática Gráfica, curso 2021-22.

Teoría. Tema 3. Visualización.

Sección 5. Representación de materiales, texturas y fuentes.

Subsección 5.3.

Materiales en el grafo de escena.

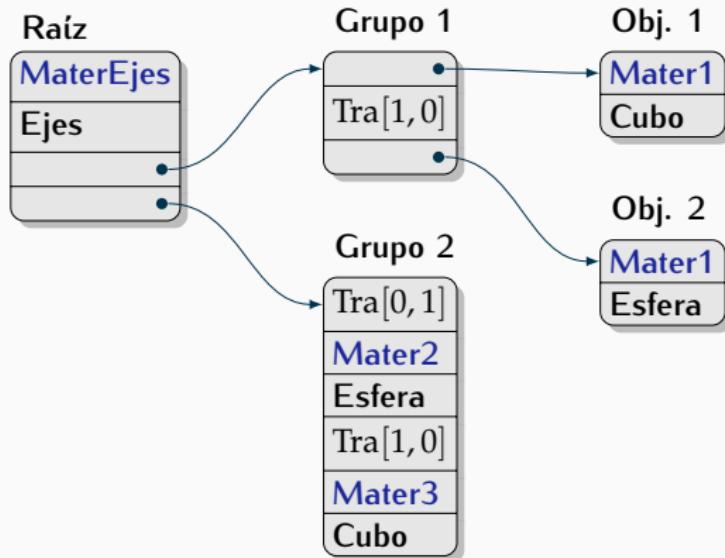
## Materiales en los grafos de escena

En los grafo de escena vamos a incorporar la posibilidad de asignar distintos materiales a distintas partes o nodos del grafo:

- ▶ Hay un nuevo tipo de entrada en los nodos: las **entradas de tipo material**: es un puntero a una instancia de **Material**.
- ▶ 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 materiales en el grafo de escena

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



# 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

En el método **visualizarGL** de la clase **NodoGrafoEscena**:

- ▶ Al inicio, registrar material activo

```
Material * material_pre = cv.iluminacion ?  
                                cv.material_act : nullptr;
```

- ▶ En el bucle, al encontrar una entrada de tipo material, se activa:

```
case TipoEntNGE::material : // si la entrada es de tipo 'material'  
    if ( cv.iluminacion ) // y si está activada la iluminación  
    {  
        cv.material_act = entradas[i].material ; // registrar material  
        cv.material_act->activar( cauce ); // activar material  
    }  
    break ;
```

- ▶ Al finalizar, reactivamos el material activo originalmente

```
if ( material_pre != nullptr )  
{  
    cv.material_act = material_previo ; // copiar el previo en 'cv'  
    cv.material_act->activar( cauce ); // activar el previo  
}
```