TEMA 3. VISUALIZACIÓN

Objetivos:

- *Conocer los distintos parámetros de la transformación de vista*
- *Comprender los procesos de proyección de la escena*
- *Conocer la influencia de la iluminación y los materiales en la apariencia de los objetos, y las distintas posibilidades existentes.*
- *Comprender el mecanismo de texturización y los parámetros que intervienen en el mismo.*

INTRODUCCIÓN

En la Ilustración 71 se representa de una forma esquemática el proceso que ocurre desde que tenemos las coordenadas geométricas de los objetos de la escena en su sistema de coordenadas local hasta que generamos una imagen 2D.

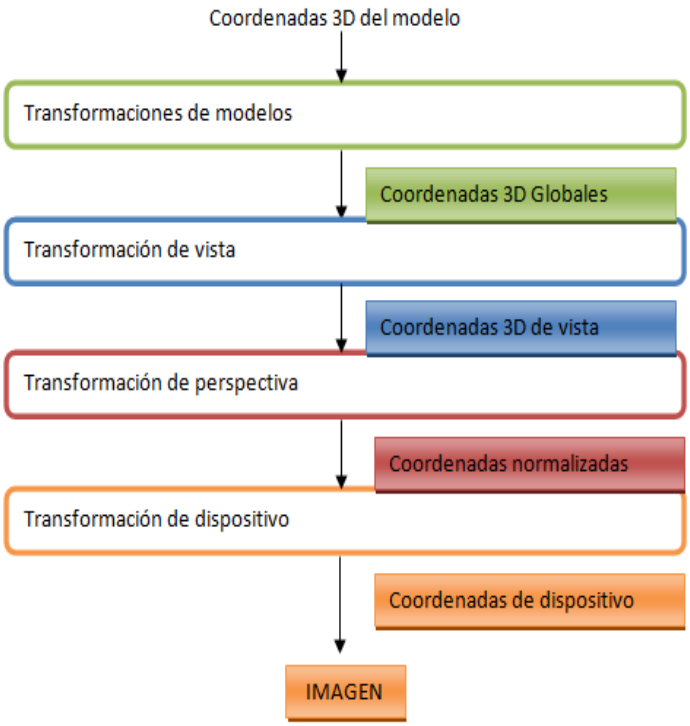


Ilustración 71: Esquema del Pipeline Fijo

A estas alturas de la asignatura hemos visto todo lo necesario para realizar las transformaciones de modelos, y hemos visto de pasada que existe una matriz V, que forma parte de la matriz *modelview*, que se encarga de realizar la transformación de vista, si bien no hemos entrado en profundidad en cómo funciona.

Lo que tradicionalmente entendemos como "cámara" en el mundo real, en realidad supone dos tareas:

- Posicionarla y orientarla en el mundo, lo que se gestiona con la matriz *modelview*
- Generar la imagen, esto es, aplicar la transformación de perspectiva. Para ello influyen muchos parámetros, como la distancia focal, el tipo de lente, etc.

En el caso de los gráficos por ordenador, además de posicionar la cámara y realizar las proyecciones adecuadas, hay otra serie de algoritmos que se han de ejecutar y que en el mundo físico pasan inadvertidos:

- Ignorar partes ocultas
- Ignorar elementos fuera del campo de visión
- Ignorar objetos o partes de objetos obstruídos por otros.

Además, el proceso físico –real- de generación de la imagen, es un largo viaje de los fotones de luz desde las distintas fuentes de luz hasta la CCD de la cámara digital, donde éstos rebotan en la escena numerosas veces y van cambiando de longitud de onda (color), según las propiedades físicas de los objetos de la escena y su comportamiento frente a la luz.

En los entornos gráficos donde se generan imágenes por rasterización, intervienen tres elementos fundamentales para simular este fenómeno físico:

- Luces
- Materiales
- Texturas

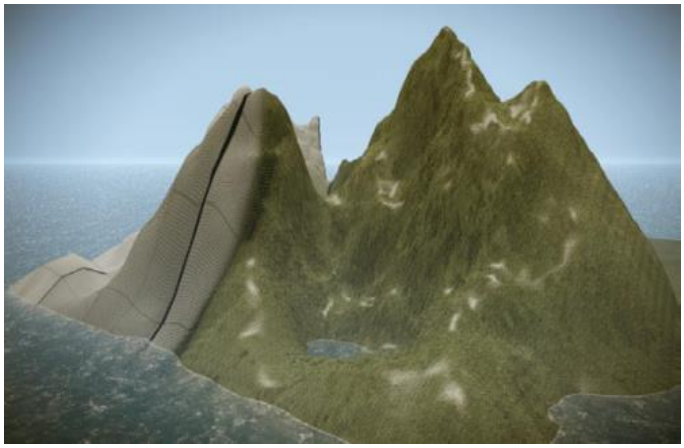


Ilustración 72 Modelo digital de terreno. A la izquierda de la isla, sin textura. A la derecha, con textura. ©CryEngine

En la Ilustración 72 se puede ver cómo con el uso de texturas se pueden conseguir imágenes mucho más realistas que con el simple uso del color plano o los materiales. El mar también tiene una textura.

Por tanto, en este tema vamos a centrarnos en las tres últimas etapas (marcadas en azul, rojo y naranja) de la Ilustración 71.

La Cámara: Transformaciones de Vista y de Proyección

La transformación de Vista

Una cámara, o un observador, tienen una posición en el espacio y una orientación. Estos parámetros son los que hacen que la escena se vea desde un punto de vista u otro, y son definidos mediante la transformación de vista.

Hay varias formas de definir esos parámetros, en función de la librería gráfica que se use o del sistema de movimiento de cámara que estemos usando en cada momento.

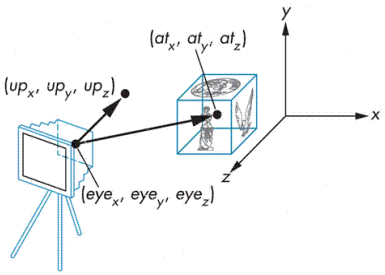


Ilustración 73: Cámara definida con EYE, AT y UP

En la Ilustración 73 se muestra esquemáticamente los parámetros de la función gluLookAt:

```
void gluLookAt(GLdouble eyeX,  GLdouble eyeY,  GLdouble eyeZ,
               GLdouble atX,  GLdouble atY,  GLdouble atZ,
               GLdouble upX,  GLdouble upY,  GLdouble upZ);
```

Esta función posiciona al observador de forma que el *ojo* está en la posición *eye*, mirando hacia el punto *at* y con el sentido "hacia arriba" definido por el vector *up*. Normalmente este vector es (0,1,0) cuando queremos ver la escena en una posición natural.

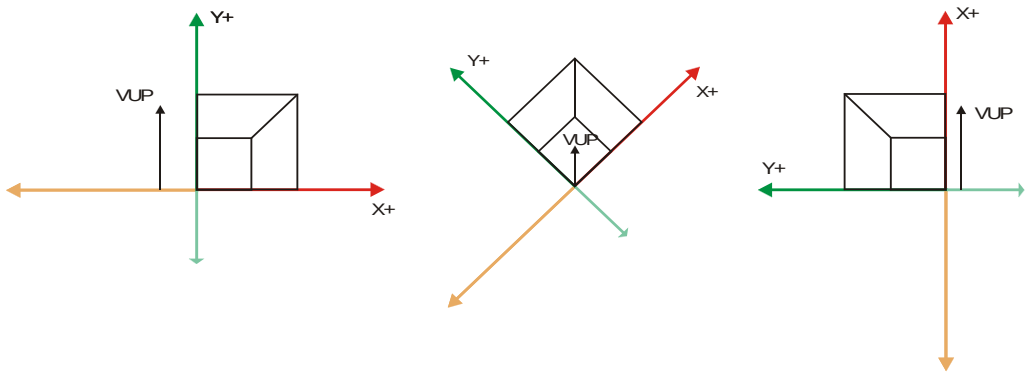


Ilustración 74 Visualización de una escena con UP (0,1,0), (1,1,0) y (1,0,0).

De hecho, en la Ilustración 74 podemos ver la imagen que se genera con idéntico valor de *eye* y *at*, pero distinto valor de *up:*

Otra forma de definir la posición y orientación de la cámara es con los siguientes parámetros, mostrados gráficamente en la Ilustración 75:

- VRP (*view reference point)*, es el punto del espacio donde está el ojo, el observador ficticio que contempla la escena. Está en el plano de proyección.
- VPN (*view plane normal*) es un vector libre perpendicular al plano de proyección de la imagen, es decir, el que indica la dirección en la que se está mirando.
- VUP *(view up*) indica la dirección "hacia arriba" en la imagen.

Nótese cómo VPN no apunta a donde se está mirando, sino hacia la cámara, y como VUP no tiene por qué ser perpendicular a éste.

En realidad, podemos ver la transformación de vista como un cambio de sistema de referencia, de forma que el mundo, la escena, pasan a estar definidos con respecto a un origen de coordenadas que sería el VRP y unos ejes cartesianos generados a partir de VPN y VUP. Representamos esa transformación en la Ilustración 76, donde vemos que los ejes de este sistema de coordenadas vienen dados por **n**, **v** y **u.**

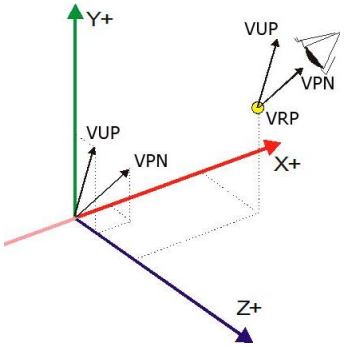


Ilustración 75 VRP, VPN y VUP

¿De dónde salen estos ejes? Veamos:

- ***n*** es el vector VPN, y sería, por decirlo de una forma intuitiva, el eje Z de nuestro sistema de coordenadas de vista.
- ***u*** sería el eje X de nuestro sistema de coordenadas de vista (digamos *"el que marca el sentido hacia la derecha"*) y es perpendicular al plano que forman VUP y VPN, por tanto, se calcula como el resultado normalizado del producto vectorial entre éstos dos vectores. ¡Ojo que el orden afecta!

$$u = \frac{VUP \times VPN}{\|VUP \times VPN\|}$$

- ***v*** sería el eje Y del sistema de coordenadas de vista, y es ortogonal a ***n*** y ***u***, por lo que su cálculo es también mediante el producto vectorial de estos dos ejes:

$$v = \frac{n \times u}{\|n \times u\|}$$

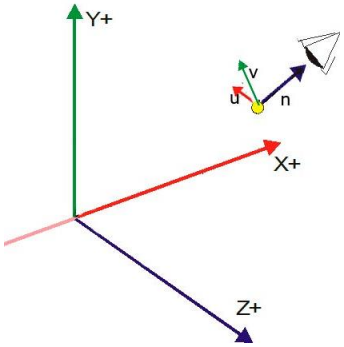


Ilustración 76: Sistema de Referencia de la Cámara

Ahora ya sabemos cómo se define la posición y orientación del observador, pero ¿en qué consiste la transformación de vista? Bueno, en realidad no es más que un cambio de sistema de referencia, y al ser **XYZ** y **uvn** sistemas ortonormales, podemos definir la matriz de vista como:

$$V = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -VRP_x \\ 0 & 1 & 0 & -VRP_y \\ 0 & 0 & 1 & -VRP_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} u_x & u_y & u_z & d_x \\ v_x & v_y & v_z & d_y \\ n_x & n_y & n_z & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Donde

$$d_x = -VRP \cdot u$$
$$d_y = -VRP \cdot v$$
$$d_z = -VRP \cdot n$$

Intuitivamente podríamos describirlo cómo trasladar y rotar la cámara para que se su sistema de coordenadas coincida con el del mundo, o lo que es lo mismo, expresar la geometría no ya en el sistema de coordenadas del mundo si no en el de la cámara.

Estos pasos se describen visualmente en la Ilustración 77, y consisten en la creación de la matriz V como una traslación seguida de una rotación de eje arbitrario, que a su vez se puede poner como tres rotaciones en torno a los ejes del mundo, tras las cuales se consigue alinear los dos sistemas de coordenadas:

$$V = R[\Upsilon,Z] \cdot R[\beta, Y] \cdot R[\alpha, X] \cdot T[-VRP_x, -VRP_y, -VRP_z]$$

Estos ángulos α, β y Υ son también conocidos como **ángulos de Euler.**

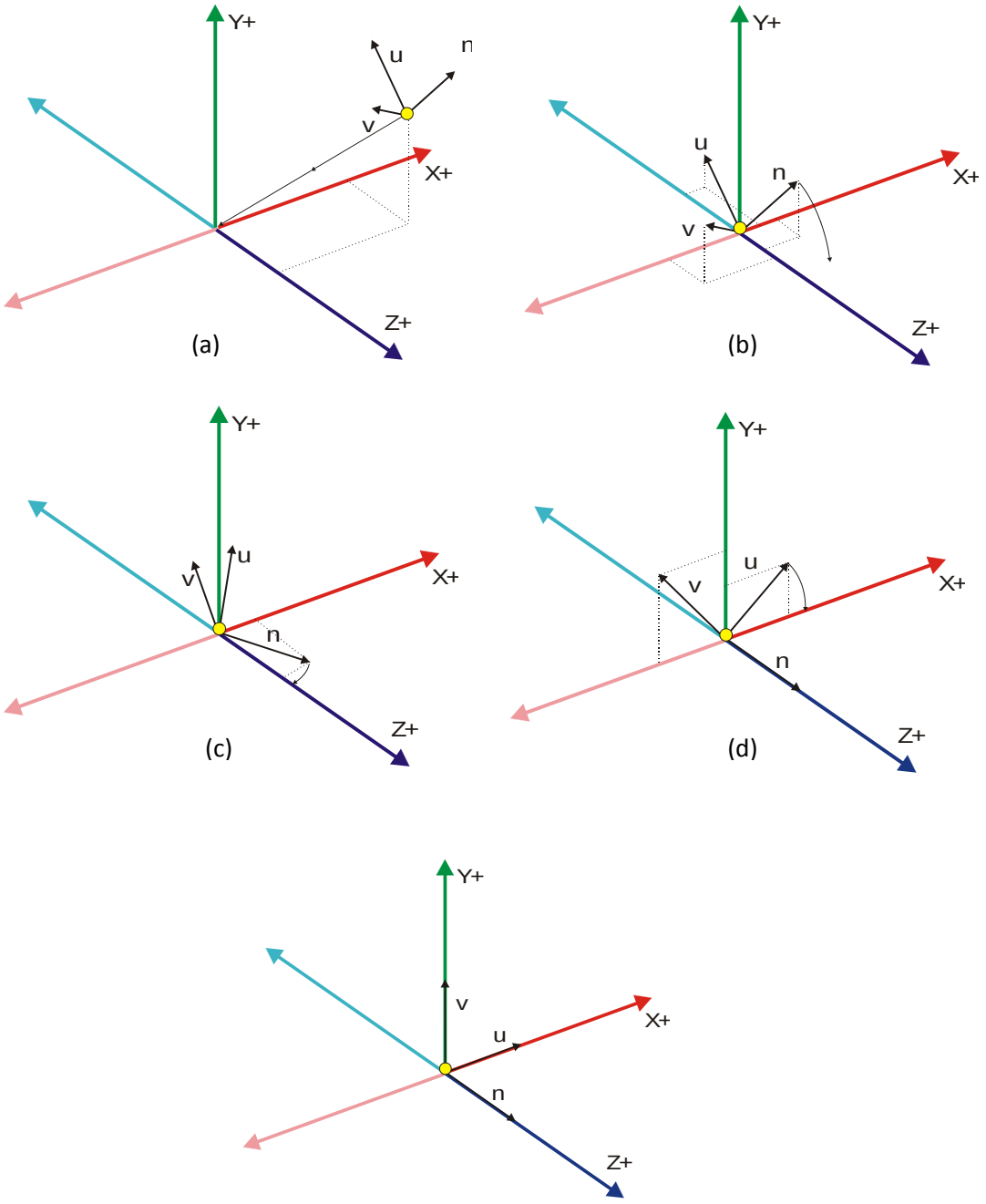


Ilustración 77: Transformación de Vista:
a). Traslación al Origen; b) Rotación en X; c) Rotación en Y; d) Rotación en Z.

EJERCICIOS

62. *Dado un cubo unitario, situado en el primer cuadrante y con una de sus esquinas en el origen:*
    1) *Dibujarlo*
    2) *Calcular el valor de las coordenadas de sus ocho vértices con los siguientes sistemas de coordenadas de vista:*
        - *VRP (1,1,1); VPN(1,1,1); VUP(0,1,0)*
        - *VRP(-2,0,0); VPN(-1,0,0); VUP(0,-1,0)*
        - *VRP(0,10,0);VPN(0,1,0); VUP(0,0,1)*
    - *Dibujar la escena según las posiciones de cámara anteriores.*
63. *Documente en sus apuntes la definición de la orientación de una cámara mediante los parámetros* **pitch**, **roll** y **yaw** (ángulos de Tait-Bryan).

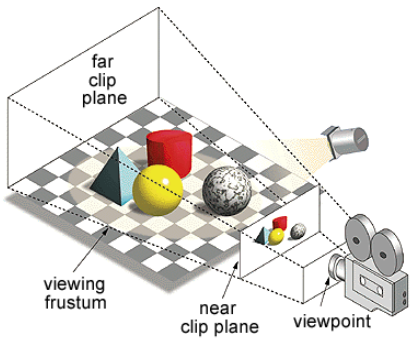


Ilustración 78: El suelo no está completamente incluido en el volumen de Visualización.

El volumen de visualización

El proceso de síntesis de una imagen, como hemos dicho infinidad de veces a esta altura del curso, emula lo que ocurre en una toma de fotografías. Pero no siempre es así, pues con una cámara fotográfica es imposible obtener imágenes ortogonales, ya que las leyes de la física hacen que toda proyección sea en perspectiva.

Además, en el proceso de síntesis de imagen, para acelerar el proceso de rasterización, se incluye el concepto de *volumen de visualización* o **frustum**, que no es más que la parte de la escena virtual que será considerada a la hora de generar la imagen 2D.

Este volumen de visualización es una pirámide truncada que se define en el sistema de coordenadas de vista, y es atravesada por el eje Z de este sistema de coordenadas. Los parámetros que definen este volumen de visualización, se ven gráficamente en la Ilustración 77Ilustración 79, y son:

- ***near***, distancia del observador al plano más cercano. Todo lo que esté más cerca que ese plano, se ignora.
- ***far***, distancia del observador al plano más lejano. Lo que esté más alejado en la escena, se ignora.
- ***bottom***, distancia desde el centro del plano cercano (eje Z de la cámara) hasta el borde inferior del frustum en su parte más cercana a la cámara (valor mínimo de coordenada Y en el S.C. de vista)
- ***top***, distancia desde el centro del plano cercano (eje Z de la cámara)

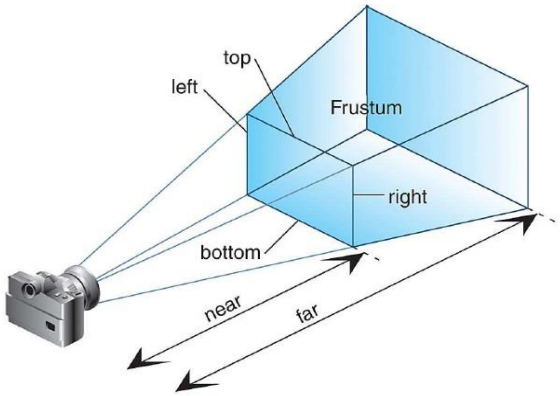


Ilustración 79: Parámetros del volumen de visualización (frustum)

hasta el borde superior del frustum en su parte más cercana a la cámara (valor máximo de Y en el S.C. de vista)

- ***right***, distancia desde el centro del plano cercano (eje Z de la cámara) hasta el borde derecho del frustum en su parte más cercana a la cámara (valor mínimo de coordenada X en el S.C. de vista)
- ***left***, distancia desde el centro del plano cercano (eje Z de la cámara) hasta el borde izquierdo del frustum en su parte más cercana a la cámara (valor mínimo de coordenada X en el S.C. de vista).

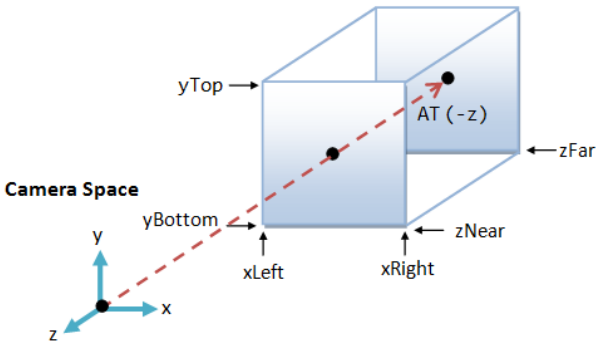


Ilustración 80 Frustum en proyección paralela.

En la Ilustración 80 podemos ver estos mismos parámetros pero con un *frustum* distinto, ya no es una pirámide truncada sino un paralelepípedo. Esto ocurre porque es una proyección ortográfica, que veremos a continuación.

Es importante tener claro que aunque se mueva la cámara, como en la Ilustración 81 los valores que definen el *frustum* **no cambian**, pues están expresados en el sistema de coordenadas de la cámara, y el "*mover la cámara*" es una transformación que pertenece al sistema de coordenadas del mundo.

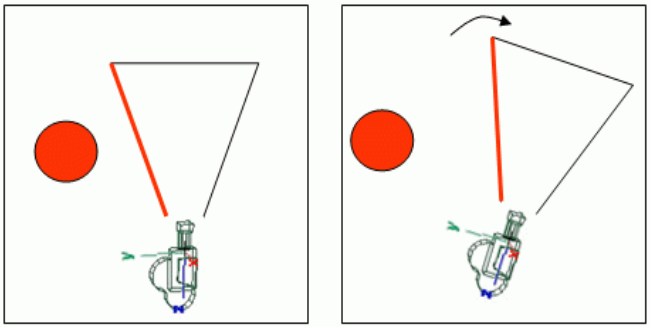


Ilustración 81 Mover la cámara no altera los parámetros del frustum.

La transformación de Proyección

Como se ha dicho hace unos instantes, la forma del volumen de visualización depende de la transformación de proyección que se vaya a utilizar, o lo que es lo mismo: la forma en que pasamos de 3D a 2D.

Si usted ha cursado Dibujo Técnico en su Bachiller, conocerá la clasificación de proyecciones de la Ilustración 82. Si bien todas se pueden llegar a representar virtualmente, las dos que vienen por defecto en OpenGL son la *perspectiva con un punto de fuga* y la *ortográfica.*

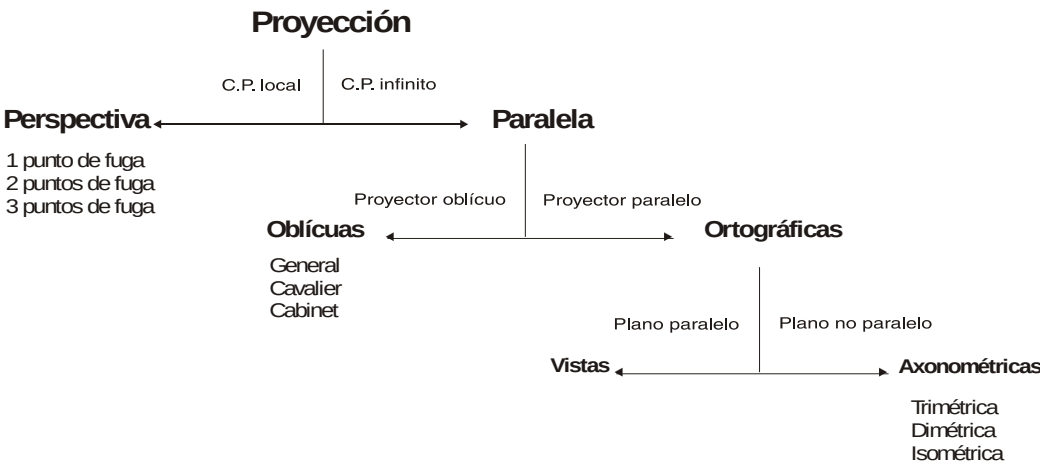


Ilustración 82: Tipos de proyecciones

¿Qué elementos conforman una operación de proyección? Se muestran gráficamente en la Ilustración 83:

- El *proyector* es una recta que pasa por el punto a y el centro de proyección.
- El *centro de proyección* es un punto por donde pasan, convergen, todos los proyectores
- El *plano de proyección* es el plano 2D donde se forma la imagen, mediante la intersección de los proyectores con este plano.

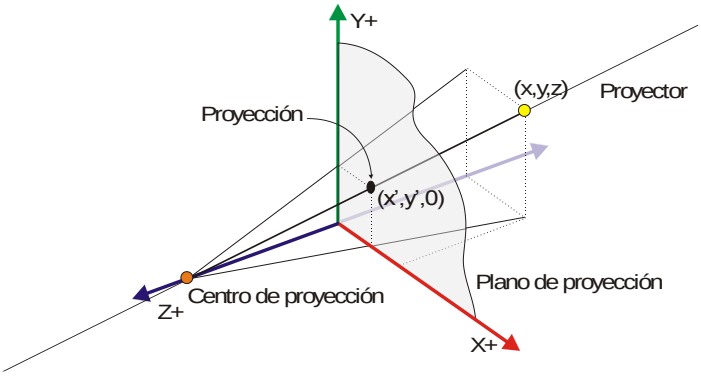


Ilustración 83 Elementos de la proyección

Lo que determinará si estamos ante uno u otro tipo de proyección es la configuración de estos elementos:

- Una *proyección perspectiva* se genera con un único centro de proyección en un punto concreto y finito del eje Z de la cámara.
- Una *proyección ortográfica* se consigue ubicando el centro de proyección en el infinito del eje Z de la cámara, de forma que todos los proyectores son paralelos

En la Ilustración 85 se puede ver los proyectores y la imagen generada con las dos proyecciones comentadas.

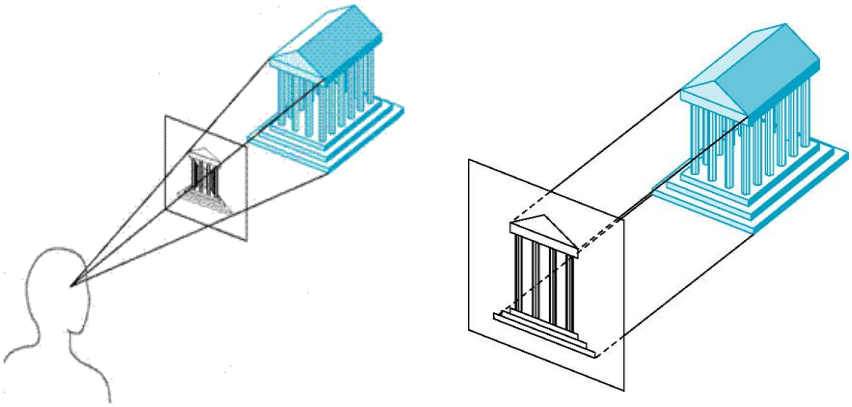


Ilustración 85 Proyección perspectiva (izda) vs Proyeccíon Ortográfica o Paralela (dcha)

¿Cómo se consigue una proyección oblicua? Pues situando el proyector en el infinito pero fuera del eje Z de la cámara, como se puede ver en la Ilustración 84

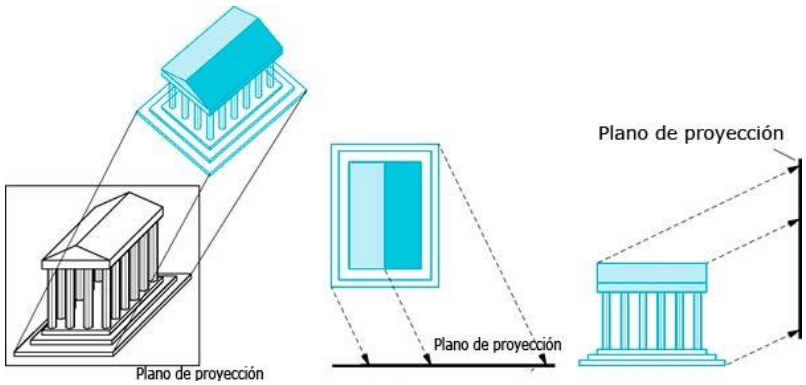


Ilustración 84 Proyección oblicua.

La proyección perspectiva

El valor de las coordenadas 2D de un punto 3D cuando se aplica una proyección de perspectiva se puede calcular fácilmente usando las propiedades de semejanza de triángulos, como se muestra en la Ilustración 86, que supone el plano de proyección en el origen de coordenadas:

$$\frac{y}{z+d} = \frac{y'}{d}$$

donde z es la distancia desde el punto al plano de proyección en coordenada z, y n la distancia desde el punto de proyección al plano *near*.

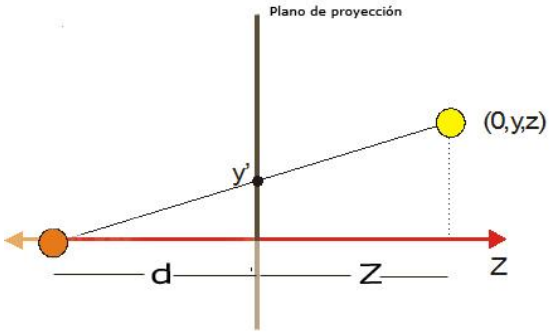


Ilustración 86 Cálculo de coordenada 2D en una proyección perspectiva.

Se puede deducir fácilmente que

$$y' = \frac{d \cdot y}{z+d}$$

Y esto hace evidente el por qué los objetos más alejados se ven más pequeños: la distancia es un factor reductor del valor de las coordenadas en 2D.

Obviamente, la coordenada x' se calcula de manera análoga a la aquí mostrada para y'.

$$x' = \frac{d \cdot x}{z+d}$$

Pero ¿no sería posible hacerlo más fácil. Vamos a reorganizar las ecuaciones anteriores:

$$x' = \frac{d \cdot x}{z+d} = \frac{x}{\frac{z+d}{d}} = \frac{x}{\frac{z}{d}+1}$$

$$y' = \frac{d \cdot y}{z+d} = \frac{y}{\frac{z+d}{d}} = \frac{y}{\frac{z}{d}+1}$$

¿No estamos realizando una transformación de coordenadas dividiendo el valor original por un término dependiente de z? Esto se podría expresar en coordenadas homogéneas como

$$[x'y'z'w'] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1/d \\ 0 & 0 & 0 & 1 \end{bmatrix} [x \quad y \quad z \quad 1]$$

Lo que da a un punto

$$\left[x \quad y \quad 0 \quad \frac{z}{d}+1\right]$$

Que al pasar a 3D, dan los valores anteriormente vistos.

La proyección ortogonal

Las proyecciones ortográficas u ortogonales, como ya hemos dicho antes, son un caso especial de proyecciones paralelas, en las cuales los proyectores son perpendiculares al plano de proyección, esto es, cuando el centro de proyección está a una distancia infinita del plano.

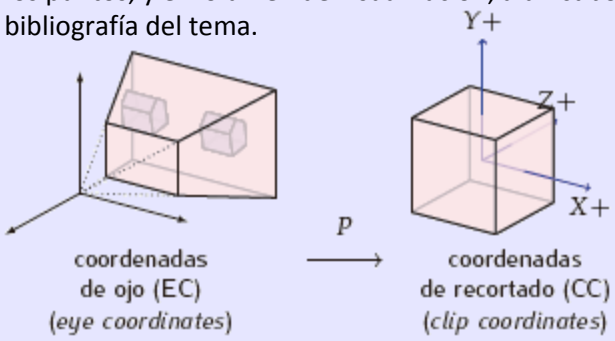
La matriz que realiza esta proyección puede ser trivialmente deducida a partir de lo especificado en el párrafo anterior: si $d=\infty$, entonces podemos crear la siguiente matriz:

$$[x'y'z'w'] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} [x \quad y \quad z \quad 1]$$

Estas matrices, tan sencillas, no son las que realmente utiliza OpenGL. Son un poco más complejas porque es necesario conservar algún tipo de información sobre el valor de la coordenada Z de los puntos. Para comprender todo el proceso, haga el ejercicio 64.

EJERCICIOS

64. *Documente en sus apuntes el paso de coordenadas de vista a coordenadas de recortado.* Esta operación transforma los puntos, y el volumen de visualización, a un cubo 2x2x2 centrado en el origen. Consulte en la bibliografía del tema.



65. *¿Cómo se implementaría una proyección oblicua en OpenGL?*
66. *¿Qué efectos se aplican en la imagen generada de la escena las siguientes acciones:*
    - *Mover el centro de proyección hacia el plano de proyección?*
    - *Alejar el plano de proyección del centro de proyección?*

Definición de la matriz de proyección en OpenGL

En la máquina de estados de OpenGL hay una matriz de proyección (*projection matrix*) que puede ser manipulada mediante varias llamadas.

La función que primero hay que invocar antes de manipular esta matriz de proyección es `glMatrixMode`, con el parámetro `GL_PROJECTION`. Esto indica a OpenGL que todas las operaciones sobre matrices que se apliquen desde ese instante hasta que se cambie el `glMatrixMode`, se aplicarán sobre la matriz de proyección. Tras esta llamada, inicializamos al valor identidad:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
```

Si queremos usar en nuestra escena una proyección perspectiva, usaremos `glFrustum`:

```
void glFrustum(GLdouble  left,  GLdouble  right,
               GLdouble  bottom,  GLdouble  top,
               GLdouble  near,  GLdouble  far);
```

Los valores near y far son siempre positivos, mientras que los demás están en el sistema coordenado de la cámara (normalmente left y bottom son negativos).

Si queremos usar en nuestra escena una proyección ortográfica, usaremos `glOrtho` con los mismos parámetros que `glFrustum`

```
void glOrtho(GLdouble  left,  GLdouble  right,
             GLdouble  bottom,  GLdouble  top,
             GLdouble  near,  GLdouble  far);
```

La librería GLU (GL Utilities), que viene "de serie" con OpenGL se nos proporciona otra función que define la proyección de perspectiva con otros parámetros, mostrados en la

```
void gluPerspective(GLdouble  fovY,
                    GLdouble  aspect,
                    GLdouble  near,  GLdouble  far)
```

Se puede ver que se utiliza unos parámetros más habituales a los parámetros de una cámara fija:

- Distancia focal (near)
- Ángulo de visión (*Field Of View* en el eje Y), o dicho de otra forma, la apertura vertical del campo de visión
- Ratio de aspecto del frustum

Una diferencia con respecto a glFrustum es que esta proyección está centrada obligatoriamente con respecto al observador, ya que no hay posibilidad de definir las posiciones de los planos del volumen de visualización.

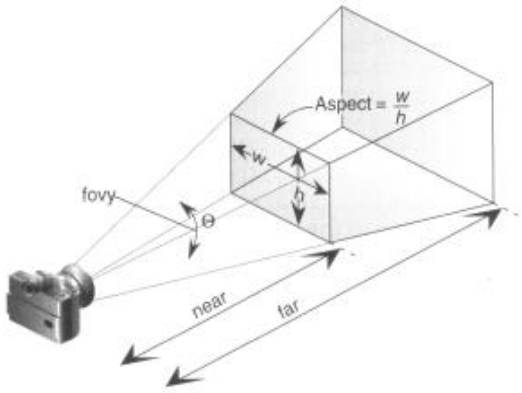


Ilustración 87 Definición de la proyección con gluPerspective

Recortado

Fijémonos en las matrices de las secciones anteriores. El valor de Z desaparece en esas matrices, pero si ha hecho el ejercicio 64, verá que no es así, que se conserva cierto valor que permite pasar de coordenadas de vista a coordenadas de recorte.

Una vez que se tienen las coordenadas de recorte de los vértices, se comprueba qué primitivas están dentro o fuera del volumen de visualización (recordemos que tras el recorte es un cubo de tamaño 2x2x2 centrado en el origen). En este paso:

- 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 otras primitivas más pequeñas de forma que:
    - Unas están dentro del volumen de visualización y se conservan.
    - El resto se descartan por estar completamente fuera.

Gráficamente se ve este proceso en la Ilustración 88.

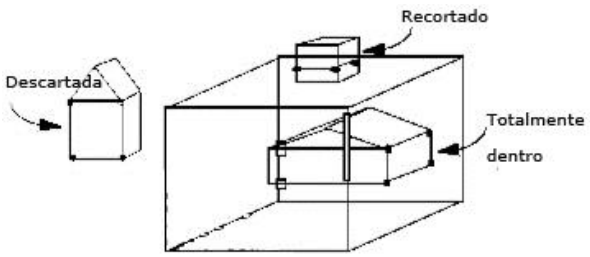


Ilustración 88 Operación de recortado

Eliminación de partes ocultas

Si un proyector atraviesa varias primitivas en su camino al centro de proyección, parece evidente que el pixel proyectado tendrá el color de la primitiva más cercana al plano de proyección, pero, ¿cómo se calcula cuál es el más cercano?

Una opción es pintar las primitivas siguiendo un orden, desde la más alejada hasta la más cercana, y así nos garantizamos que la última en pintar cada pixel será la más cercana. Pero esto es muy poco eficiente.

Lo más usado es el algoritmo de ZBuffer, que se representa visualmente en la Ilustración 89. Para cada primitiva que se dibuje, se almacena en el pixel del ZBuffer su valor de profundidad normalizado dentro del volumen de visualización (0 lo más cercano al plano *near*, y 1 lo más cercano al plano *far*). De esta forma, si el valor de Z de la primitiva es menor que el actualmente existente en esa posición del buffer, se sustituye el color del pixel por ese nuevo valor y se actualiza el ZBuffer.

La principal ventaja de este algoritmo es que en el peor de los casos su complejidad es proporcional al número de primitivas.

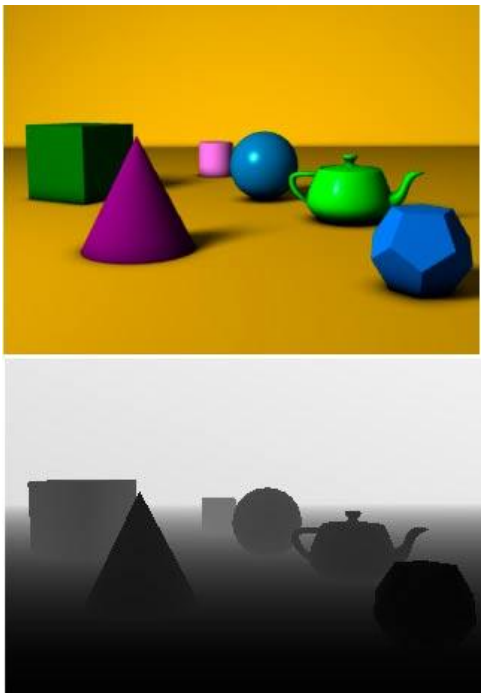


Ilustración 89: Escena (arriba) y su Z-Buffer (abajo)

¿Cómo se hace en OpenGL? En las prácticas se da hecho, y quizá ahora se comprendan mejor las siguientes líneas:

```
glutInitDisplaymode( GLIT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
glEnable(GL_DEPTH_TEST);
```

Cada vez que se dibuja, igual que limpiamos el buffer de color, hay que limpiar el ZBuffer:

```
glClear(GL_DEPTH_BUFFER);
```

EJERCICIOS

67. *Documente en sus apuntes la transformación de dispositivo*, que permite saber en qué pixel de la imagen se proyecta cada vértice. Su resultado son las coordenadas de dispositivo.

ILUMINACIÓN

Para crear escenas realistas, es necesario simular la interacción de la luz con los objetos en el mundo, y también tener en cuenta cómo ese mundo es percibido por los usuarios. OpenGL utiliza un modelo de iluminación y visualización que es una aproximación de la realidad, obviamente no lo suficientemente preciso como para que parezca hiperrealista, pero sí lo bastante bueno como para que el usuario comprenda el mundo 3D que se está representando.

Esta técnica de rasterización, anteriormente descrita, es mucho más rápida que otras técnicas que ofrecen resultados más realistas, pero inviables para la visualización y manipulación en tiempo real de las escenas 3D. Ya hemos visto en temas anteriores los fundamentos del raytracing, en los cuales no profundizaremos.

La luz

La luz es una radiación es una radiación electromagnética, un tipo de ondas que se propagan por el espacio, de una naturaleza similar a las ondas que se usan para las comunicaciones móviles, wifi, radio y televisión. La propiedad física que corresponde a la percepción de color es la ***longitud de onda***.

El sistema visual humano puede percibir esta radiación sólo cuando su longitud de onda está aproximadamente entre 390 y 750 nanómetros. Es lo que se llama el espectro visible, y se muestra en la Ilustración 90.

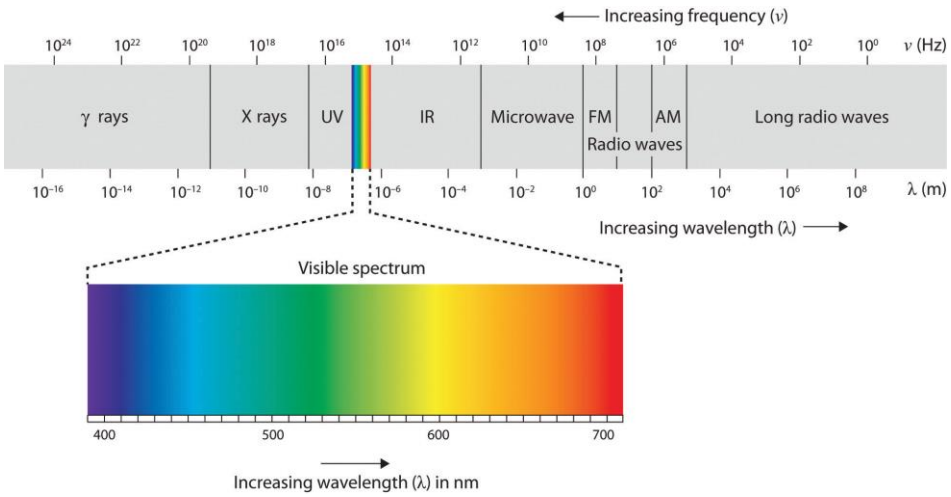


Ilustración 90: Espectro electromagnético.

Como se puede ver, es una banda muy estrecha de todo el espectro electromagnético, algunos de cuyos espectros pueden ser capturados visiblemente con aparatos especiales.

La emisión e interacción de las ondas electromagnéticas en los átomos nos permite percibir el entorno, y si bien la radiación tiene propiedades duales (onda/corpúsculo), en Informática Gráfica se suele usar más frecuentemente el modelo de partículas.

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 de estos fotones tiene una *energía radiante*, que depende de su longitud de onda. En un punto **p** de la superficie de un objeto podemos medir la densidad de energía radiante en un instante de tiempo de los fotones de una longitud de onda concreta (λ) que pasan en una determinada dirección (v). Este valor **L(λ,p,v)** se denomina ***radiancia***, y es lo que determina el tono de color y el brillo con que observamos el punto **p** cuando lo vemos desde la dirección **v**.

Desde un punto **p** con una dirección **v** pueden emitirse o reflejarse una gran cantidad de fotones con longitudes de onda distintas. El **brillo** o intensidad dependerá de la cantidad de fotones. El **color** dependerá de la distribución de las longitudes de onda de dichos fotones.

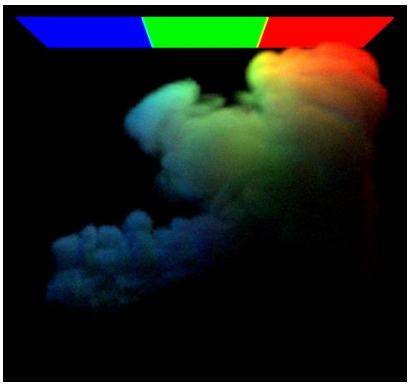


Ilustración 91 Nube iluminada con tres luces distintas.

Cuando la luz de alguna fuente choca con una superficie, parte de la energía es absorbida y parte es reflejada (y puede que alguna atraviese el objeto). La apariencia del objeto dependerá de la naturaleza de la luz que lo ilumina y de las propiedades de la superficie con la que los fotones chocan. Un objeto que refleja la mayor parte de la luz roja que lo ilumina es porque absorbe la mayoría del resto de fotones con una longitud de onda distinta al color rojo. Obviamente, esto ocurre porque la luz que incide sobre el objeto contiene fotones "rojos", si no sería imposible que se viera de dicho color. Este efecto se ilustra en la Ilustración 91.

El sistema visual humano

El ojo es la parte del *sistema visual humano* capaz de enviar señales eléctricas al cerebro que dependen de las características de la luz que incide sobre sus sensores, situados en la retina. Estas células, denominadas conos y bastones, son las encargadas de obtener toda la información lumínica posible.

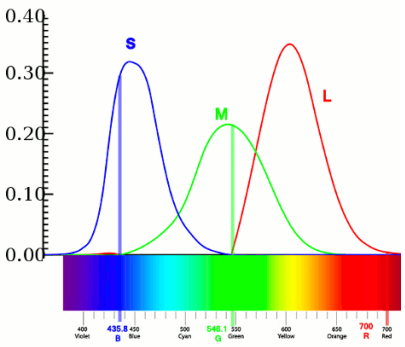


Ilustración 92: Sensibilidad de cada tipo de cono.

- Los bastones se encargan de la visión en blanco y negro, y están conectados en paralelo al nervio. Permiten la visión periférica por su gran extensión a lo largo de la retina (hay de 75 a 150 millones).
- Los conos son los encargados de detectar las distintas longitudes de onda, y por tanto el color. Hay unos 6 o 7 millones en la fóvea, teniendo una conexión nerviosa cada uno de ellos. Permiten ver con gran detalle, y a todo color. En concreto hay tres tipo de conos que, simplificando mucho, podríamos decir que son los que se encargan de detectar las longitudes de onda *"azul", "verde"* y *"roja"* (aunque realmente hay un gran solapamiento entre los rangos de longitudes de onda que cada uno puede detectar, como se aprecia en la Ilustración 92).

Aunque es imposible obtener toda la gama de colores que el ojo humano es capaz de percibir, tradicionalmente en los dispositivos de visualización como las pantallas de ordenador se utiliza el comportamiento simplificado de los conos para, mediante la teoría aditiva de estos colores, denominados **primarios**, obtener una gran gama de colores. Es lo que se denomina el color RGB, por las iniciales en inglés de *red, green* y *blue.*

El espacio RGB

Como consecuencia de lo anterior, cualquier color reproducible en un dispositivo se puede representar por una terna *(r,g,b)* con los tres valores comprendidos entre 0 y 1. El valor 0 indica que el correspondiente color no aparece y el valor 1 significa la máxima potencia para ese color.

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, como se muestra en la Ilustración 93.

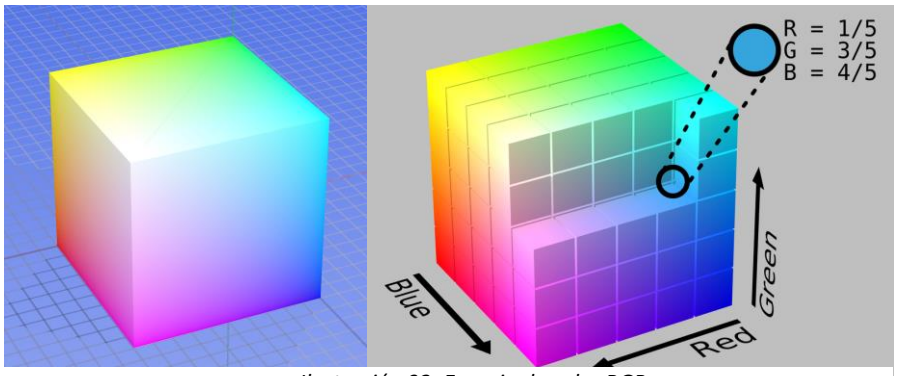


Ilustración 93: Espacio de color RGB

EJERCICIOS

68. *El espacio de color RGB es aditivo. ¿Existen los espacios de color sustractivos? Documéntelo en sus apuntes.*
69. *El espacio RGB no es el único esquema para representar computacionalmente los colores, pero sí el más usado. Documente en sus apuntes la teoría de color que subyace bajo el modelo HSV.*
70. *¿Sabría reproducir las imágenes que ve un daltónico con su ordenador? ¿Cómo?*

Un modelo de iluminación simplificado

La radiación electromagnética visible se genera en las fuentes de luz, que pueden ser:

- Fuentes naturales: sol o estrellas, fuego, objetos incandescentes, órganos de algunos animales, etc…
- Fuentes artificiales o luminarias: filamentos incandescentes, tubos fluorescentes, LEDs, etc.

Los fotones creados en las fuentes de luz interactúan con los átomos de la materia que encuentran a su paso, de forma que:

- Parte de la energía recibida se convierte en **calor**
- Parte de la energía recibida se convierte en **radiación reflejada**

Esta radiación reflejada puede reflejarse varias veces en distintos objetos e incluso a distintos niveles de profundidad del objeto intersectado. Como esto genera una complejidad de rebotes, difracciones, intersecciones, etc. imposibles de calcular de forma eficiente, OpenGL realiza en su forma más básica diferentes simplificaciones (mostradas gráficamente en la Ilustración 94) :

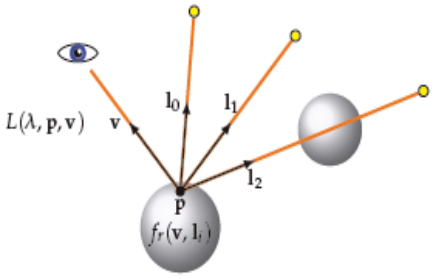


Ilustración 94: Modelo simplificado de iluminación.

1. Las fuentes de luz son puntuales o unidireccionales, y hay un número finito de ellas (hasta 8)
2. No se considera la luz incidente en una superficie que no provenga directamente de las fuentes de luz. Para suplir esos fotones que andan "rebotando" por la escena, se crea una radiancia ***ambiente*** constante.
3. Los objetos o polígonos son totalmente opacos (no hay transparencias ni materiales translúcidos).
4. No se consideran sombras arrojadas, es decir, un objeto no impide la trayectoria de la luz.

5. El espacio entre los objetos no dispersa la luz (ésta tiene igual densidad independientemente de la distancia a la fuente)
6. En lugar de considerar todas las longitudes de onda posibles, se usa el modelo RGB.

La consecuencia de esta simplificación se puede ver en la Ilustración 95 (además de que en la figura de la izquierda se usan texturas y otros efectos visuales).

Ilustración 95 Iluminación compleja (izda.) vs Iluminación simplificada (dcha.)

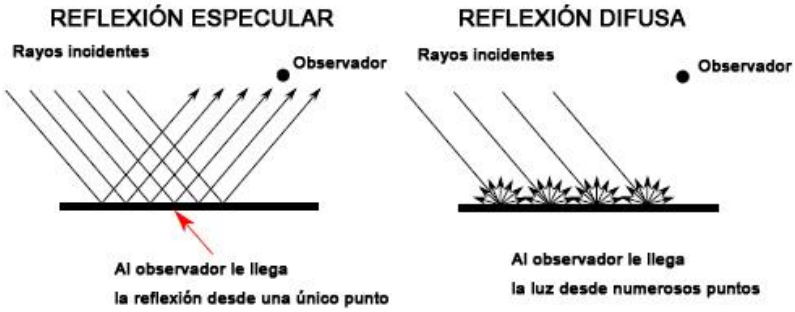


Ilustración 96: Reflexión Especular vs Reflexión Difusa

Para este modelo simplificado, podemos considerar dos tipos genéricos de reflexiones en la superficie: la reflexión **especular** y la **difusa**, mostradas visualmente en la Ilustración 96.

En un modelo de reflexión especular perfecto (como un espejo), un rayo de luz es reflejado al 100% cuando interseca la superficie, formando el mismo ángulo de salida que el ángulo incidente sobre la superficie. Esto supone que el observador sólo verá ese rayo reflejado si está en la posición adecuada, justo en la trayectoria de ese rebote de luz. Incluso si la superficie entera está iluminada por esa fuente de luz, el observador sólo verá la reflexión desde los puntos por los que pasan esos rayos reflejados. Esas reflexiones se suelen denominar **brillos especulares**, y en la práctica se suele pensar en un rayo que es reflejado no como una línea sino como un cono de luz, de forma que superficies muy brillantes producen conos de reflexión muy estrechos y nítidos.

Por el contrario, en un modelo de reflexión difuso perfecto, los rayos de luz son rebotados con igual intensidad en todas las direcciones, por lo que el observador vería ese rayo de luz reflejado desde cualquier punto de vista. Esto da una apariencia de iluminación homogénea de toda la superficie.

Fuentes de luz

En el modelo de la escena se pueden incluir un conjunto de *n* fuentes de luz, pudiendo ser cada una de ellas de dos tipos:

- Fuentes de luz **posicionales**, que ocupan un punto en el espacio y emiten en todas direcciones.
- Fuentes de luz **direccionales**, que están en un punto a distancia infinita y tienen un vector director en el que se emiten los fotones.

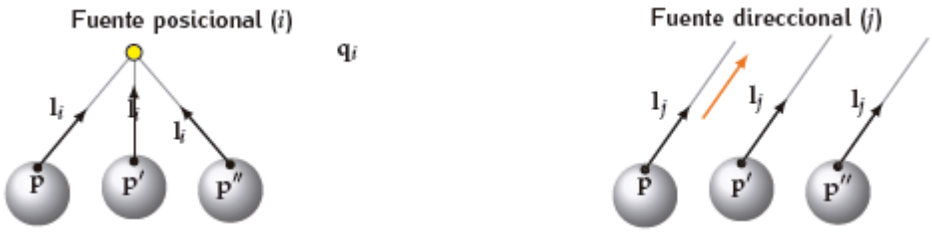


Ilustración 97 Fuente de luz posicional vs direccional

Normales

La iluminación en un punto *p* depende la orientación de la superficie en dicho punto. Esta orientación está caracterizada por el vector normal n_p asociado a dicho punto. n_p es un vector de longitud unidad, que idealmente es perpendicular al plano tangente a la superficie en el punto p (en azul en la Ilustración 98)

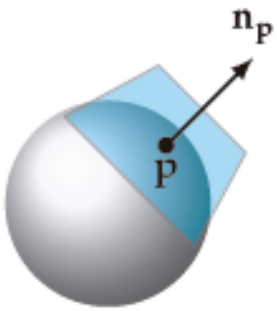


Ilustración 98: Vector normal en un punto p

En modelos de fronteras, puede calcularse de varias formas:

- Considerar que n_p es la normal del triángulo que contiene a *p*
- Aproximar el valor de n_p en cada vértice como la media de las normales de los triángulos que lo comparten.

Materiales

Cuando un rayo de luz choca con una superficie, los fotones de cierta longitud de onda son absorbidos, otros se reflejan de forma difusa y otros son reflejados especularmente. La forma en que la superficie reacciona a las diferentes longitudes de onda es lo que constituirá el color en que se percibe.

Llegados a este punto, debemos tener claro que el color no es una simple terna RGB que le pasamos a un vértice o primitiva, sino que está compuesto de dos elementos y su combinación: las características de la luz incidente y las propiedades de la superficie, o lo que es lo mismo, del *material*.

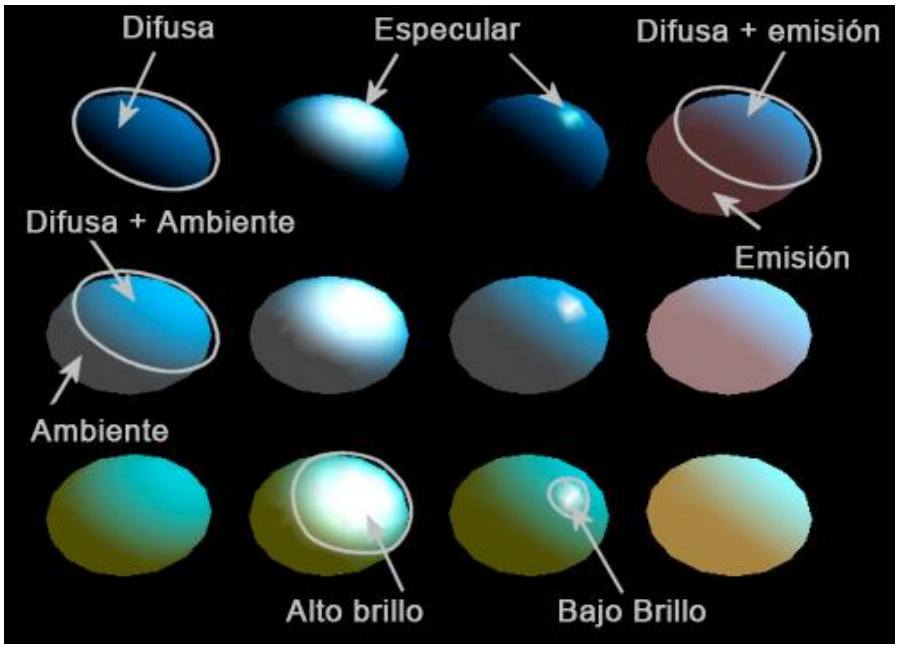


Ilustración 99 Las cuatro componentes de un material.

El modelo de reflectancia de Phong propone que un material se pueda definir mediante cuatro componentes:

- Color **difuso**, que indica qué longitudes de onda refleja el material de forma difusa. Podemos decir que es el color "base" del material.
- Color **especular**, que indica qué longitudes de onda refleja el material de forma especular, o dicho de otra forma, el color de los brillos.
- Color **ambiente**, que se refiere al comportamiento del objeto cuando no le incide directamente ninguna fuente de luz.
- Color de **emisión**, y se refiera a un comportamiento "artificial" que hace que se vea el objeto incluso cuando no hay fuentes de luz activas. El objeto no iluminará a otros, no se convierte en una fuente de luz, pero sí se vería en una escena a oscuras (algo así como las pegatinas fluorescentes).

Además, hay una quita componente que es el brillo, es decir, la cantidad de luz que es rebotada de forma especular.

Recordamos, aún a fuerza de ser insistentes, que el resultado final, el color del pixel, no depende únicamente de las propiedades del material, sino también de los valores de color de las fuentes de luz que intervienen. Simplificando al máximo lo anteriormente expuesto, podríamos decir de manera informal que:

- Un material "blanco" iluminado por una fuente de luz roja se verá rojo.
- Un material "blanco" iluminado por una fuente de luz verde, se verá verde
- Un material "verde" iluminado por una fuente de luz roja se verá naranja (Ilustración 100).

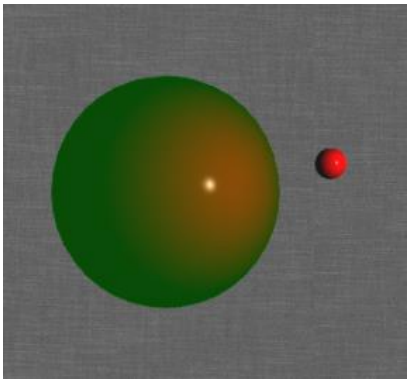


Ilustración 100 Material ambiental verde iluminado con luz difusa roja.

Iluminación en OpenGL

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 el modelo de iluminación local
glDisable(GL_LIGHTING); / / desactiva el modelo de iluminación local
```

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

Colores vs iluminación

El comportamiento de OpenGL depende de si la evaluación del modelo de iluminación local 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 color resultante se calcula a partir de los parámetros existentes en la máquina de estados relativos a las características de las luces y los materiales, en lugar del especificado con glColor.

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

- Una luz ambiente global, que "viene de la nada". Se especifica con:

```
GLfloat lmodel_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
```

- Para las luces:
  - S_{iA}, S_{iD}, S_{iE} componentes ambiental, difusa y especular de la i-ésima fuente de luz.
  - q_i, l_i posición o dirección de la i-ésima fuente de luz.
- para los materiales, en el momento de dibujar una primitiva se consulta:
  - M_E emisividad del material.
  - M_A,M_D,M_S reflectividad difusa, ambiente y especular del material.
  - e exponente de la componente pseudo-especular.

Estos parámetros se pueden modificar en cada momento.

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 modelo de iluminación local (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 SiA := (ra, ga, ba)
glLightfv( GL_LIGHTi, GL_DIFFUSE, cdf ) ; / / hace SiD := (rd, gd, bd)
glLightfv( GL_LIGHTi, GL_SPECULAR, csf ) ; / / hace SiS := (rs, gs, bs)
```

Donde GL_LIGHTi puede ser GL_LIGHT0, GL_LIGHT1… o cualquiera de las constantes que identifican a las luces.

Para la luz GL_LIGHT0, el valor por defecto para los parámetros GL_DIFFUSE y GL_SPECULAR es (1.0, 1.0, 1.0, 1.0), mientras que para el resto de luces es (0.0, 0.0, 0.0, 1.0). El valor por defecto de GL_AMBIENT es (0.0, 0.0, 0.0, 1.0) para todas las luces. Por eso, las escenas se ven con sólo "encender" la luz GL_LIGHT0, que es blanca "de serie".

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:

```
const GLfloat posf[4] = { px, py, pz, 1.0 } ;  // (x,y,z,w)
glLightfv( GL_LIGHTi, GL_POSITION, posf );

const GLfloat dirf[4] = { vx, vy, vz, 0.0 } ; // (x,y,z,w)
glLightfv( GL_LIGHTi, GL_POSITION, dirf );
```

El valor de w, el cuarto valor del vector, determina el tipo de fuente de luz. Si es 0, es una luz direccional (p.ej. el sol). Si w es distinto de cero, estamos en una luz posicional (p.ej. una bombilla de una lámpara), e irradia en todas direcciones.

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.

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

- **Coordenadas de cámara**: si se especifica cuando M = Identidad.
- **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)

Esto permite crear luces que permanezcan fijas (como si estuviera en el casco del observador por ejemplo), estacionarias en un lugar del mundo (como una farola, o el sol), o moviéndose con un objeto (como los faros del coche). Todo depende del momento en el que se indique su posición.

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 ángulos α y β de longitud y latitud, respectivamente, en grados), podríamos hacer:

```
const float[4] ejeZ = { 0.0, 0.0, 1.0, 0.0 } ;
glMatrixMode( GL_MODELVIEW ) ;
glPushMatrix() ;
glLoadIdentity() ; / / hacer M = Identidad

glMultMatrix( A ) ; / / A podría ser Ide,V o VN

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

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

// luz paralela eje Z+
glLightfv( GL_LIGHT0, GL_POSITION, ejeZ );
glPopMatrix() ;
```

EJERCICIOS

71. *En la url* http://fly.srk.fer.hr/~unreal/theredbook/theredbook.zip está disponible la edición PDF del OpenGL Programming Guide (libro rojo). Completa en tus apuntes y estudia los ejemplos de mover la luz alrededor de un objeto de forma independiente o acompañando a un objeto de la escena.
72. Accede a http://3dfd.ujaen.es/cgex/v2/#typesoflights y prueba al menos tres tipos de luces con dos posiciones y direcciones distintas.

Observador local o en el infinito

OpenGL debe calcular el vector *v* que va desde el punto *p* hacia el observador. Esto debe hacerse en función del tipo de proyección activo. La función `glLightModel` permite configurar este comportamiento

Si la proyección es ortogonal, *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 proyección es perspectiva, se dice que el observador es local (no está en el infinito), y en este caso debemos indicar:

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

Especificación de normales de vértices

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

```
glBegin(GL_TRIANGLES);
glNormal3f( nx0, ny0, nz0 ); glVertex3f( x0, y0, z0 );
glNormal3f( nx1, ny1, nz1 ); glVertex3f( x1, y1, z1 );
glNormal3f( nx2, ny2, nz2 ); glVertex3f( x2, y2, z2 );
/ / ..... otros triángulos ....
glEnd() ;
```

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 modelo de iluminación local. 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
  ```

Al igual que `glVertex`, `glNormal` está obsoleto, y se puede definir la normal para cada vértice utilizando un vector de normales, pues como hemos indicado, la normal es un atributo asociado a cada vértice, como lo era el color o lo será la coordenada de textura:

```
glVertexPointer( 3, GL_FLOAT, 0, v );
glNormalPointer( GL_FLOAT, 0, nv );

glEnableClientState( GL_VERTEX_ARRAY );
glEnableClientState( GL_NORMAL_ARRAY );

glDrawElements( GL_TRIANGLES, 3*num_tri, GL_UNSIGNED_INT, tri ) ;
glDisableClientState( GL_VERTEX_ARRAY );
glDisableClientState( GL_NORMAL_ARRAY );
```

Definición de atributos de materiales.

Las propiedades del material también forman parte del estado y se modifican con llamadas a la función `glMaterial`.

Para modificar la emisividad del material hacemos:

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

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 MA := (r, g, b), inicialmente (0.2,0.2,0.2)
glMaterialfv( GL_FRONT_AND_BACK, GL_AMBIENT, color ) ;
// hace MD := (r, g, b), inicialmente (0.8,0.8,0.8)
glMaterialfv( GL_FRONT_AND_BACK, GL_DIFFUSE, color ) ;
// hace MS := (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 ) ;
```

Con la función glColorMaterial podemos hacer que el valor de alguna de las reflectividades del material se haga igual a C cada vez que C cambie (por una llamada a glColor)

```
/ / asociar ME (emisión) con C :
glColorMaterial( GL_FRONT_AND_BACK, GL_EMISSION ) ;
/ / asociar MA (refl. ambiente) con C :
glColorMaterial( GL_FRONT_AND_BACK, GL_AMBIENT ) ;
/ / asociar MD (refl. difusa) con C :
glColorMaterial( GL_FRONT_AND_BACK, GL_DIFFUSE ) ;
/ / asociar MS (refl. pseudo-especular) con C :
glColorMaterial( GL_FRONT_AND_BACK, GL_SPECULAR ) ;
/ / asociar MA y MD con C :
glColorMaterial( GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE ) ;
```

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

Por defecto, se consideran polígonos delanteros aquellos en cuya proyección los vértices aparecen en sentido anti-horario al recorrerlos en el orden en el que se proporcionan a OpenGL con glVertex. El resto son traseras (este comportamiento es configurable).

Todas las llamadas que permiten cambiar el material tienen un primer parámetro que permite discriminarsobre 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

Esta funcionalidad de OpenGL permite asignar distinto color a un polígono en función de si estamos viendo una cara de dicho polígono o la otra, es decir, el material puede tener distinto aspecto en cada una de las dos caras de un polígono. Esto se corresponde con el aspecto de objetos reales bidimensionales que pueden ser observados por ambos lados.

Para ello, es relevante el orden en el que se proporcionan los vértices a OpenGL, pues dicho orden determina qué cara es considerada delantera y qué cara es considerada trasera. Se puede ver gráficamente en la Ilustración 101.

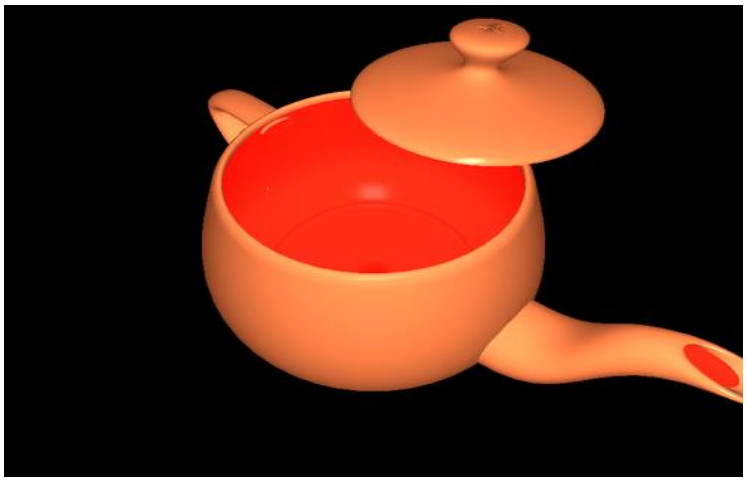


Ilustración 101 Material frontal naranja, y trasero rojo.

EJERCICIOS

73. Accede a http://3dfd.ujaen.es/cgex/v2/#lightsandmaterials y prueba distintas configuraciones de luces y materiales, anota los valores utilizados y pon capturas de pantalla de los resultados obtenidos. Para cada material, utiliza distintos valores de luces, y para cada luz, utiliza distintos valores de material.

74. *Suponga que visualizamos una esfera de radio unidad centrada en el origen. Se ilumina con una fuente de luz puntual situada en el punto p = (0, 2, 0). El observador está situado en o = (2, 0, 0). En estas condiciones:*

    - *Describe razonadamente en que punto de la superficie el brillo será máximo si el material es puramente difuso (MD = (1, 1, 1), y el resto de reflectividades a 0) ¿ es ese punto visible para el observador ?*
    - *Repita el razonamiento anterior asumiendo ahora que el material es puramente especular (M_S = (1, 1, 1), resto a cero). Indique si dicho punto es visible para el observador.*

TEXTURAS

Los objetos reales presenta a veces detalles a pequeña escala, como son manchas, defectos, motivos ornamentales, rugosidades, o, en general, cambios en el espacio de los atributos que determinan su apariencia. En la Ilustración 102 podemos ver un motivo ornamental de una tela, un papel arrugado, un veteado de madera o un mapa. Todos son ejemplos de información de colores distintos que no se pueden representar con materiales como los vistos en el apartado anterior, ya que estos suponen una homogeneidad a lo largo de la superficie.

Ilustración 102 Distintos ejemplos de texturas

Estas variaciones como las mostradas en la Ilustración 102 se pueden modelar como funciones que asignan a cada punto de la superficie de un objeto un valor diferente para algunos parámetros del modelo de iluminación local. Lo más usual es variar las propiedades de las componentes difusa y ambiente del material, pero se puede hacer tambien variando "artificialmente" los valores de la normal (*bump mapping*), o a veces otros parámetros.

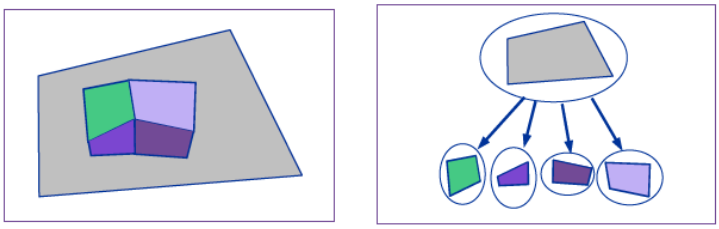


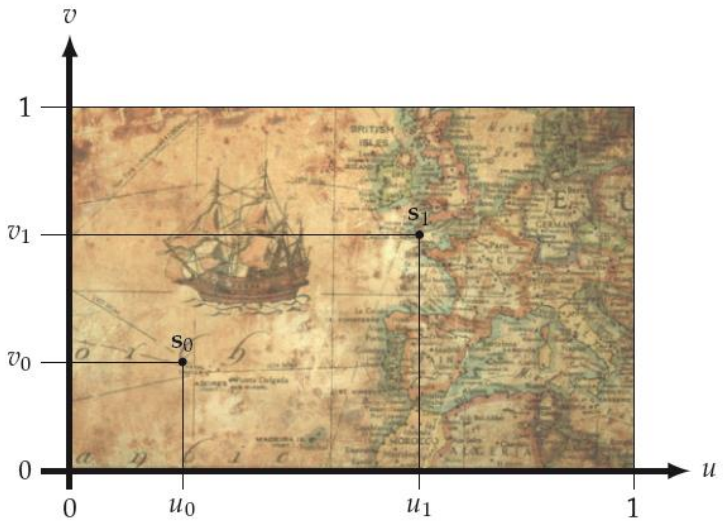
Ilustración 103 Geometría extra para ofrecer detalles de distinto color.

Una primera opción, intuitiva, podría ser crear pequeñas piezas de geometría que, aplicando materiales distintos, ofrezcan el nivel de detalle deseado, como se ve en la . Sin embargo, la complejidad en su generación y, especialmente, el gran espacio de almacenamiento necesario, lo hace casi impracticable si el único objetivo es ofrecer más nivel de detalle en el color.

Por ello surgieron las texturas. Una textura no es más que una imagen que representa las modificaciones de los parámetros anteriormente indicados . Se puede interpretar como una función T que asocia a cada punto s de un dominio D (usualmente [0, 1] x [0, 1]) un valor para un parámetro del modelo de iluminación global (típicamente M_D y M_A —propiedades del material difusa y ambiente). 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* (**tex**ture **el**ements). A esta imagen se le llama imagen de textura.

En la Ilustración 105 se puede ver como para dos coordenadas de textura distintas, expresadas como puntos s_i del dominio D=[0,1]x[0x1] el valor devuelto $T(s_i)$ es una terna RGB distinta, que corresponde con el valor del pixel de la imagen.



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

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

Ilustración 105 Textura 2D

A la hora de aplicar color a un objeto, hay tres formas posibles, mostradas en la Ilustración 104. En la parte inferior izquierdas vemos el cálculo de color como se ha visto en el epígrafe anterior, con un material blanco. En la parte superior izquierda se muestra la aplicación de una textura tal cual (se ve totalmente plana, sin afectar la iluminación). Si se evalúa el modelo de iluminación local con los valores de material obtenidos de la textura, el resultado es el que se muestra a la derecha .

Ilustración 104 Aplicación de textura

Coordenadas de textura

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

Debe existir una función f tal que $(u, v) = f(x, y, z)$. Entonces decimos que (u, v) son las coordenadas de textura del punto p.

La función f puede implementarse usando una tabla de coordenadas de textura de los vértices, o bien calcularse proceduralmente con un subprograma.

En la Ilustración 106 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.

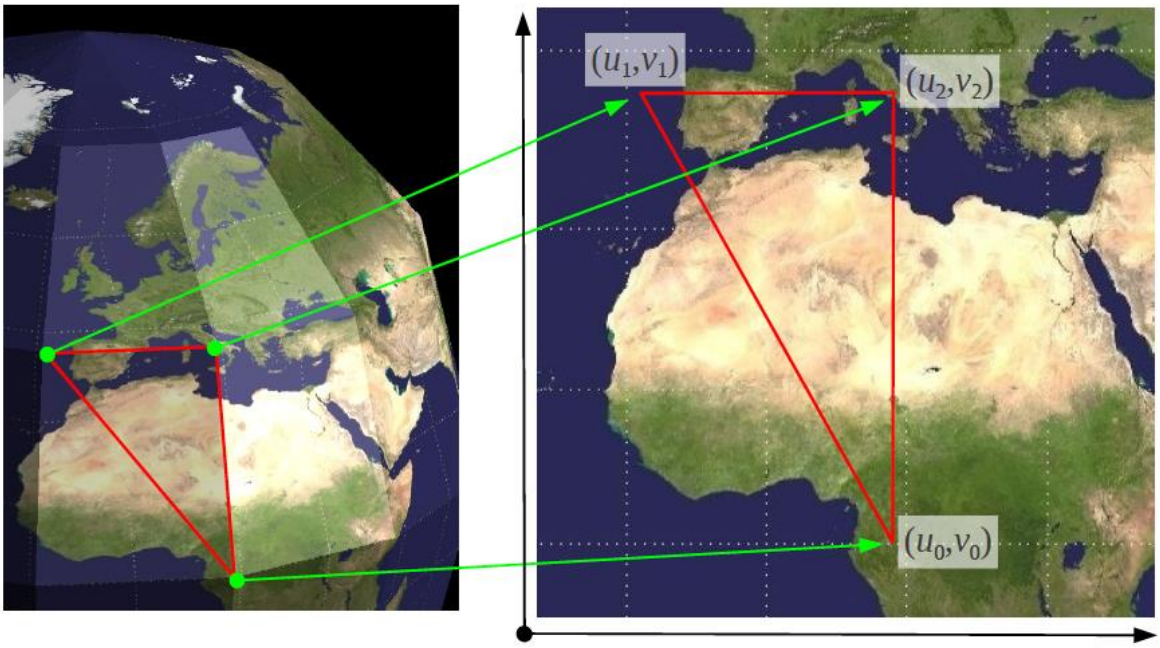


Ilustración 106 Coordenadas de textura de los vértices de un triángulo.

La asignación de coordenadas de textura se puede hacer de dos formas:

- **Asignación explícita a vértices**: La coordenadas forman parte de la definición del modelo de escena, y son un dato de entrada al cauce gráfico, en forma de un vector o tabla de coordenadas de textura de vértices (v_0, u_0), (v_1, u_1), . . . (u_{n-1}, v_{n-1}). Se puede hacer:
  - manualmente en objetos sencillos, o bien
  - de forma asistida usando software para CAD (p.ej. 3D Studio).

  Esta modalidad hace necesario realizar una interpolación de coordenadas de textura en el interior de los polígonos de la malla. Se puede ver un ejemplo en la Ilustración 107, donde cada cara del cubo es la imagen de la textura completa.

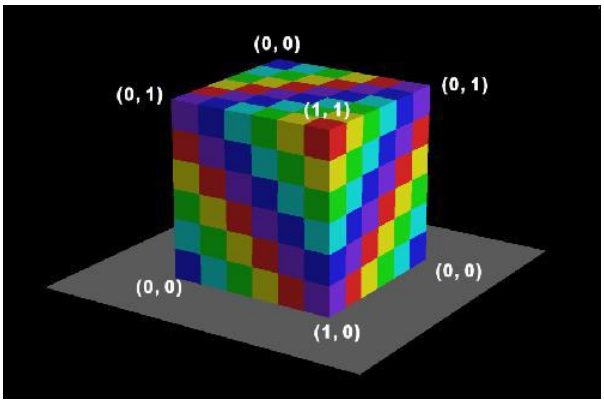
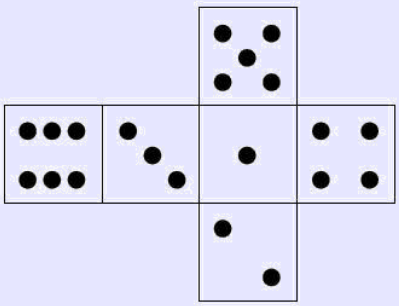


Ilustración 107 Asignación manual de texturas. Imagen de http://www.siggraph.org/education/materials/HyperGraph/mapping/r_wolfe/r_wolfe_mapping_1.htm

EJERCICIOS

62. *Supongamos que se desea crear una malla, formada por una tabla de vértices y triángulos, para un cubo de forma que deseamos aplicar un textura que incluya las caras de un dado. Para ello disponemos de una imagen de textura que tiene una relación de aspecto 4:3 (es decir el ancho en pixels, dividido por el alto, es igual a 4/3). La imagen aparece aquí.*



- *Describa razonadamente cuantos vértices (como mínimo) tendrá el modelo.*
- *Escriba la tabla de coordenadas de vértices, la tabla de coordenadas de textura, y la tabla de triángulos. Tenga en cuenta que el cubo tiene lado unidad y su centro está en (1/2, 1/2, 1/2). Dibuje un esquema de la textura en la cual cada vértice del modelo aparezca etiquetado con su número de vértice más sus coordenadas de textura.*

63. *Considere de nuevo el cubo y la textura del problema anterior. Ahora suponga que queremos visualizar con OpenGL el cubo usando sombreado de Gouroud, para lo cual debemos de asignar normales a los vértices.*

- *Describa razonadamente si sería posible usar la misma tabla de vértices y la misma tabla de coordenadas de textura que ha escrito en en el problema anterior, o es necesario usar otra tabla distinta.*
- *Escriba ahora la tabla de vértices, la de coordenadas de textura, y la tabla de normales.*

- **Asignación procedural**: La función *f* se implementa como un algoritmo `CoordText(p)` que se invoca para calcular las coordenadas de textura, de forma que acepta un punto p como parámetro y devuelve el par $(u, v)= f(p)$ con las coordenadas de textura de p. Esta asignación procedural se puede hacer de dos formas
  - ***Asignación procedural a vértices***, de forma que `coordText(vi)` es invocado para calcular las coordenadas de textura de cada vértice, y las coordenadas obtenidas se almacenan y después se interpolan linealmente en el interior de los polígonos de la malla. Es decir, usamos una función para realizar la asignación por vértices, pero una vez creado el vector de coordenadas de textura, se comporta igual que en el caso de la asignación explícita a vértices.
  - ***Asignación procedural a puntos***, en cuyo caso `coordText(p)` es invocado cada vez que hay que calcular el color de un punto p de la superficie durante los cálculos del modelo de iluminación local. Esto sólo se puede hacer programándolo en el *fragment shader.*

Los tipos de funciones f son de lo más diversos, por lo que nos centraremos sólo en tres aproximaciones, por ser las más usadas.

Asignación procedural por coordenadas paramétricas

La principal complejidad de la asignación de coordenadas de texturas a un modelo 3D es que hay que hacer corresponder una imagen bidimensional con unos elementos tridimensionales, con lo que ello supone.

Una superficie paramétrica es una variedad plana de dos dimensiones para la cual existe una función g (con dominio en [0, 1]x[0, 1]) tal que, si p es un punto de la superficie, entonces existen (s, t) tales que $p=g(s, t)$:

- En este caso, al par (s, t) se le llaman coordenadas paramétricas del punto p, y a la función g se le llama función de parametrización de la superficie.
- En estas condiciones, podemos hacer $(u, v) = f(p) = (s, t)$, es decir, podemos usar las coordenadas paramétricas como coordenadas de textura.

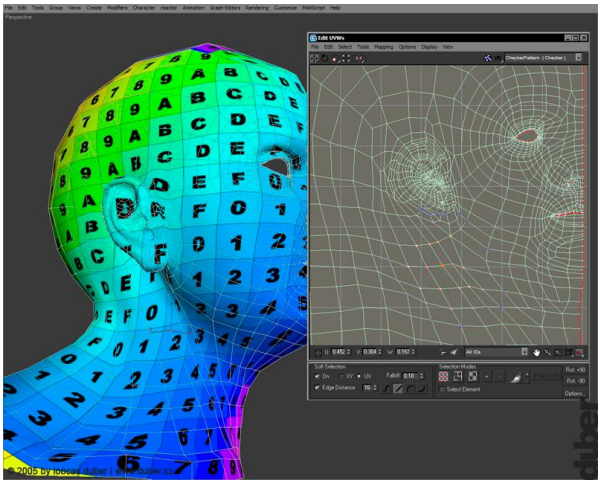


Ilustración 108 Parametrización de una cabeza

Una superficie paramétrica permite situar todos los vértices de la malla en un espacio (0,1)x(0x1), de forma que la correspondencia con las coordenadas de textura es inmediata. Si se aproxima la malla mediante una superficie paramétrica, como en la Ilustración 108 Parametrización de una cabeza, lo que suele ocurrir, salvo que la superficie sea plana, se producen distorsiones de área y ángulos en los polígonos.

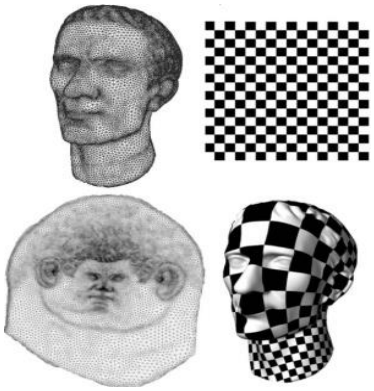


Ilustración 109 Parametrización y texturización de cabeza.

En la Ilustración 109 se puede ver cómo una malla homeomorfa a una esfera, como el caso de la cabeza, se puede texturizar con una textura de ajedrez, si bien, debido a la distorsión que se genera en la malla parametrizada, los cuadros de la parte frontal del rostro se ven mucho más grandes que los del cuello, que más o menos conservan el área.

En la Ilustración 110 se muestra una superficie parametrizada en 3D, con sus coordenadas de textura. Si se aplica la textura con dichas coordenadas, tenemos un parche parametrizado como se ve en la tercera imagen. Si "troceamos" la tetera en varios parches parametrizados de forma análoga, podemos texturizarla como se ve en la cuarta imagen con una única textura.

En numerosas ocasiones, para no incurrir en distorsiones tan graves como las de la Ilustración 109, se procede a trocear el modelo poligonal y realizar una texturización por partes.

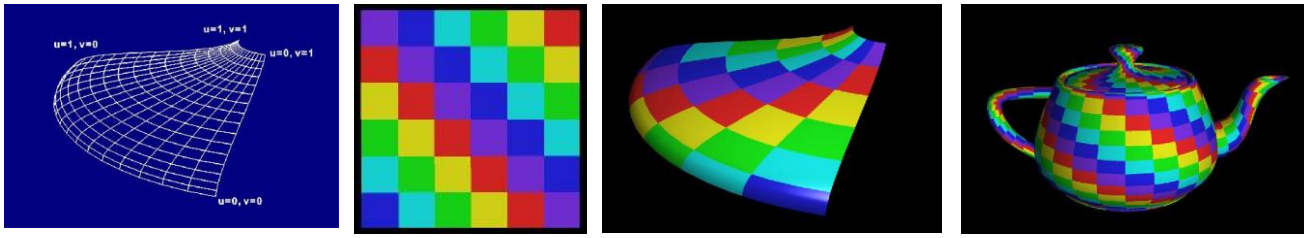


Ilustración 110 Parche paramétrico, al que aplicándose la textura (b), resulta un parche texturizado. A la derecha, tetera formada por varios parches parametrizados.

Asignación procedural por coordenadas cilíndricas

Se basa en usar 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 (α, h, r) se obtienen a partir de las coordenadas cartesianas (x, y, z) (también con origen en el centro del objeto). Hacemos:

$$\alpha = atan2(z, x) \qquad h = y$$

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

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

Los resultados de la asignación procedural de coordenadas de textura por coordenadas cilíndricas se pude ver en la Ilustración 111.

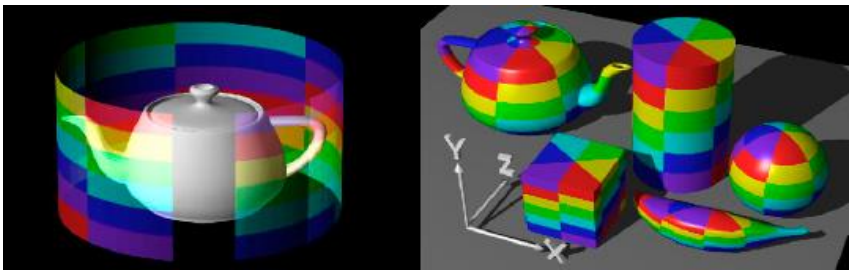


Ilustración 111 Aplicación de coordenadas cilíndricas sobre varios objetos.

Asignación procedural por coordenadas esféricas

Se basa en usar las coordenadas polares (longitud, latitud y radio) del punto p. Equivale a una proyección radial en una esfera, como se muestra en la Ilustración 112.

Las coordenadas (α, β, r) se obtienen a partir de las coordenadas cartesianas (x, y, z) (normalmente coordenadas de objeto, con el origen en un punto central de dicho objeto). Hacemos:

$$\alpha = atan2(z, x) \qquad \beta = atan2(y, \sqrt[2]{x^2 + z^2})$$

De esta forma α está en el rango [-π,π] y β está en el rango [-π/2,π/2], pudiendo calcular u y v como sigue:

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

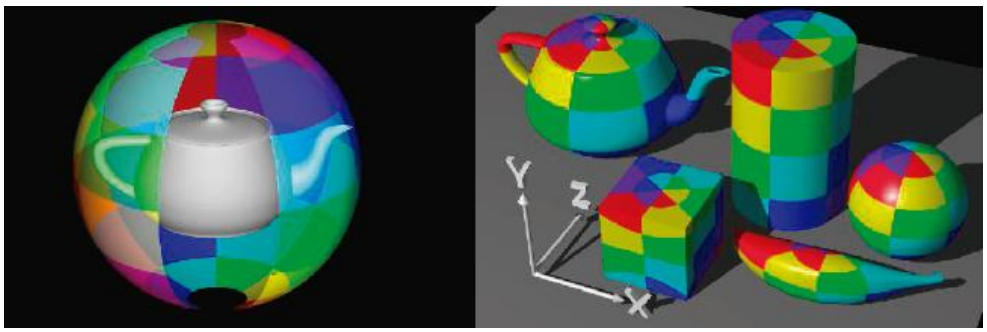


Ilustración 112 Aplicación de coordenadas esféricas.

Activación y desactivación

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

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

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

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

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

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 almacena el nuevo identificador
```

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

```
GLuint arrIdTex[n] ;
glGenTextures( n, arrIdTex);
```

Textura activa

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

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

Para cambiar el identificador de textura activa podemos hacer:

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

Alojamiento en RAM de imágenes de textura

Antes de usar una textura en OpenGL (de tamaño nx x ny), es necesario alojar en la memoria RAM una matriz con los colores de sus texels, ya sean cargados desde un archivo de imagen o generados proceduralmente.

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 3nx bytes de cada fila de texels se almacenan contiguos, de izquierda a derecha.

- Las ny 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, 3·nx·ny bytes consecutivos en memoria.

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

```
glTexImage2D( GL_TEXTURE_2D,
              0, // nivel de mipmap (para imágenes multiresolución)
              GL_RGB, // formato interno
              ancho, // núm. de columnas (potencia de dos: 2n)
              alto, // núm de filas (potencia de dos: 2m)
              0, // tamaño del borde, usualmente es 0
              GL_RGB, // formato y orden de los texels en RAM
              GL_UNSIGNED_BYTE, // tipo 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).

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 (mipmaps)

```
gluBuild2DMipmaps( GL_TEXTURE_2D,
                   GL_RGB, // formato interno
                   ancho, // núm. de columnas (arbitrario)
                   alto, // núm de filas (arbitrario)
                   GL_RGB, // formato y orden de los texels en RAM
                   GL_UNSIGNED_BYTE, // tipo de cada texel
                   Texels
);
```

Esta función hace varias versiones de la imagen, para usar una u otra a distintas resoluciones, en función de los píxeles que ocupa en pantalla el modelo.

Configuración de las 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:

- Cómo se usa el color de los texels con la iluminación activada (que reflectividades del material son obtenidas de la textura activa). La función `glLightModel` puede usarse para determinar cómo los colores de la textura afectan al modelo de iluminación local, 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);
    ```
  - 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);
    ```

- Cómo se selecciona el texel o texels a partir de una coordenada de textura (más cercano o interpolación), mostrado en la Ilustración 113. Si se decide tomar el valor más cercano usaremos

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

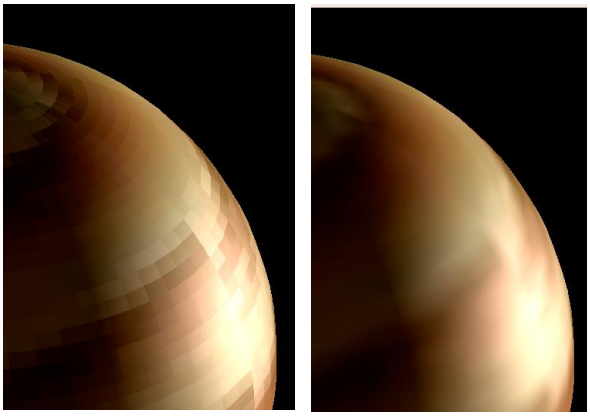


Ilustración 113 Interpolación "más cercano" (izda) o bilineal (dcha).

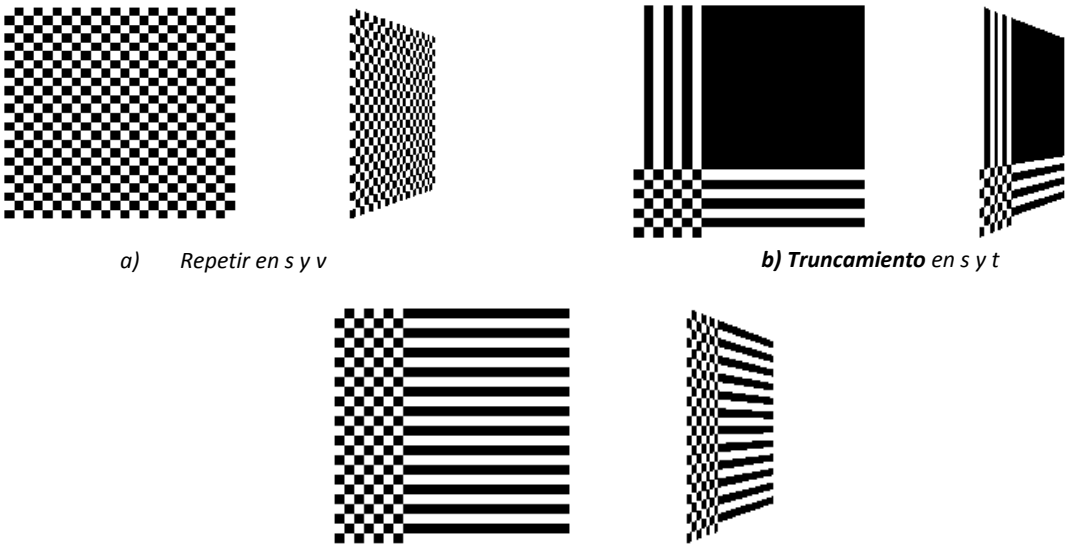
Y si se hace interpolación bilineal entre los cuatro texels con centros más cercanos al centro del pixel:

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

`GL_TEXTURE_MAG_FILTER` se usa cuando nos acercamos al objeto texturizado (magnificación), y `GL_TEXTURE_MIN_FILTER` para cuando nos alejamos.

- Cómo se selecciona el texel cuando las coordenadas de textura no están en el rango [0, 1] (replicado o truncamiento, *repeat* o *clamp*). El efecto se puede ver en la Ilustración 114, y se hace con las siguientes instrucciones para el caso 114.c:

```
glTexParameteri( GL_TEXTURE_2D,
                 GL_TEXTURE_WRAP_S,
                 GL_CLAMP );
glTexParameteri( GL_TEXTURE_2D,
                 GL_TEXTURE_WRAP_T,
                 GL_REPEAT );
```



a) Repetir en s y v ***b) Truncamiento*** *en s y t*

c) Repetir en t, truncar en s

Ilustración 114 Tratamiento de coordenadas de textura fuera del rango [0,1]

También se debe indicar a la máquina de estados si se asignan explícitamente coordenadas o bien OpenGL las genera proceduralmente, y en este caso como se hace. 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 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).

Con OpenGL podemos usar funciones lineales (proyección en un plano) para la generación automática de coordenadas de textura, como se muestra en la Ilustración 115.

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

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

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

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

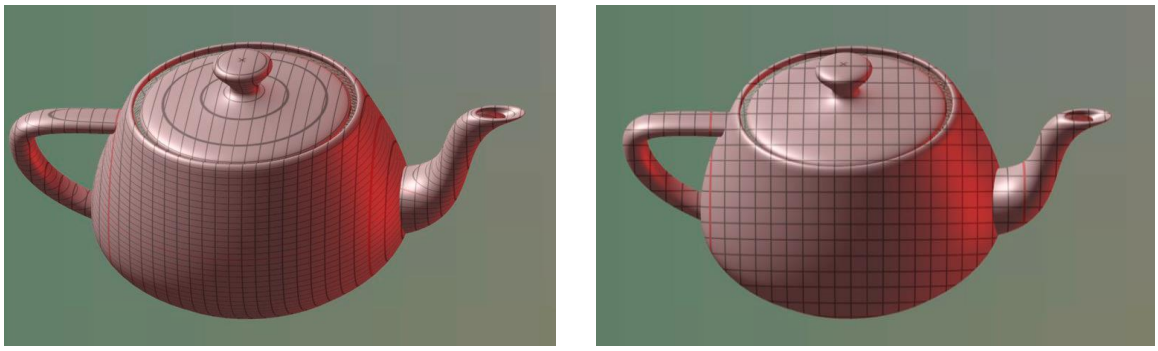


Ilustración 115 Asignación automática de coordenadas de textura en coordenadas de objeto (zida) o de vista (dcha)

También es possible generar coordenadas de textura esféricas con:

```
glTextGeni(GL_S,GL_TEXTURE_GEN_MODE, GL_SHPERE_MAP);
glTextGeni(GL_T,GL_TEXTURE_GEN_MODE, GL_SHPERE_MAP);
```

Asignación explícita de coordenadas de textura

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

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

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

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

```
glVertexPointer( 3, GL_FLOAT, 0, ver );
glNormalPointer( GL_FLOAT, 0, nv);
glTexCoordPointer( 2, GL_FLOAT, 0, ctv);

glEnableClientState( GL_VERTEX_ARRAY );
glEnableClientState( GL_NORMAL_ARRAY );
glEnableClientState( GL_TEXTURE_COORD_ARRAY );

glDrawElements( GL_TRIANGLES, 3L*num_tri, GL_UNSIGNED_INT, tri ) ;

glDisableClientState( GL_VERTEX_ARRAY );
glDisableClientState( GL_NORMAL_ARRAY );
glDisableClientState( GL_TEXTURE_COORD_ARRAY );
```

EJERCICIOS

64. *Visite la web* http://3dfd.ujaen.es/cgex/v2/#textures *y documente el efecto de los parámetros de WRAP y FILTERs sobre distintas texturas, modificando también las coordenadas de textura del polígono visualizado.*

BIBLIOGRAFÍA

[Foley97] **Computer Graphics: Principles and Practice in C**; Foley, Van Dam, Feiner y Hughes. de. Addison-Wesley , 1997.

[Shirley09] **Fundamentals of Computer Graphics;** P. Shirley. AK Peters, 2009

[Shreiner05] **OpenGL Programming Guide;** Shreiner et al., Addison Wesley, 2005

[Hill01] **Computer Graphics using OpenGL;** F.S. Hill Jr., Prentice Hall, 2001

[Angel08] **Interactive Computer Graphics: A Top Down Approach** (5ª Ed); E. Angel. Addison Wesley, 2008

[Hearn10] **Computer Graphics with Open GL** (4ª Ed) ; D. Hearn, P. Baker, W. Carithers; Prentice Hall, 2010

Recursos online (principalmente cursos en otras universidades):

- http://www.xmission.com/~nate/tutors.html
- http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-837-computer-graphics-fall-2003/
- http://www.student.cs.uwaterloo.ca/~cs488/
- http://3dfd.ujaen.es/cgex/v2/
- http://www.ntu.edu.sg/home/ehchua/programming/opengl/cg_basicstheory.html