ResumenAsignatura.pdf

JMRamG

Informática Gráfica

3º Grado en Ingeniería Informática

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
Universidad de Granada

Informática gráfica (2021-2022)
Grado en Ingeniería Informática
Universidad de Granada

Resumen del temario de la asignatura hasta el tema 3

Índice

Informática Gráfica

Banco de apuntes de la

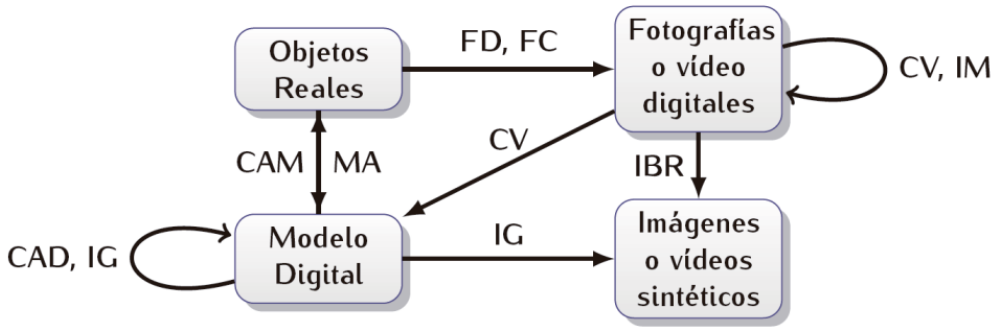
WUOLAH

Comparte estos flyers en tu clase y consigue más dinero y recompensas

1. Imprime esta hoja
2. Recorta por la mitad
3. Coloca en un lugar visible para que tus compis puedan escanar y acceder a apuntes
4. Llévate dinero por cada descarga de los documentos descargados a través de tu QR

1. Qué es la IG

La informática gráfica se encuentra dentro de un área de conocimiento que se denomina *área de computación visual*, donde se relaciona con otras disciplinas, señaladas en la figura 1.1.



FD	Fotografía Digital	FC	Fotografía Computacional
CV	Visión por Ordenador	IBR	*Rendering* Basado en Imág.
CAD	Diseño Asistido por Ord.	CAM	Fabric. Asistida por Ord.
MA	Adquisición de Modelos	IM	Tratamiento de Imágenes

Figura 1.1: computación visual

la informática gráfica actualmente está presente en disciplinas como:

- Videojuegos.
- Producción de animaciones y efectos especiales para cine y televisión.
- Diseño industrial.
- Modelado y Visualización en Ingeniería y Arquitectura.
- Simuladores y juegos serios para entrenamiento y aprendizaje.
- Visualización de datos.
- Visualización científica y médica.
- Arte digital.
- Patrimonio cultural.

Para poder visualizar imágenes correctamente, necesitamos:

- Modelos digitales 3D de lo que queremos representar:
  - Geometría (primitivas geométricas, normalmente triángulos).
  - Propiedades sobre la apariencia (textura, material, color).
- Iluminación (posición de luces, modelo de sombreado, etc).
- Una o varias cámaras, indicando para cada una:
  - Lente que usar(ángulo de visión).
  - Posición y orientación.
- Información sobre la resolución de la imagen de salida.

Estos elementos forman los dos componentes principales: la escena y los parámetros de visualización.
La escena es lo que está en el mundo, independientemente de que hagamos la foto o no. Los parámetros de visualización son los que hacen que la foto salga de una manera u otra (posición de la cámara, lente, formato de salida, filtros, etc.)
En la figura 1.2 podemos observar el flujo de transformaciones[1], que supone lo siguiente:

1. Transformaciones del modelo:
   - Situarlo en la escena.
   - Cambiarlo de forma.
   - Crear modelos compuestos de otros más simples.
2. Transformación de vista:
   - Poner al observador en la posición deseada.
3. Transformación de perspectiva:
   - Pasar de un mundo 3D a una imagen 2D.
4. Rasterización:
   - Calcular para cada píxel su color, teniendo en cuenta la primitiva que se muestra, su color, material, texturas, luces, etc...
5. Transformaciones del dispositivo:
   - Adaptar la imagen 2D a la zona de dibujado.

[1] A su vez en la imagen 1.3 tenemos una representación visual

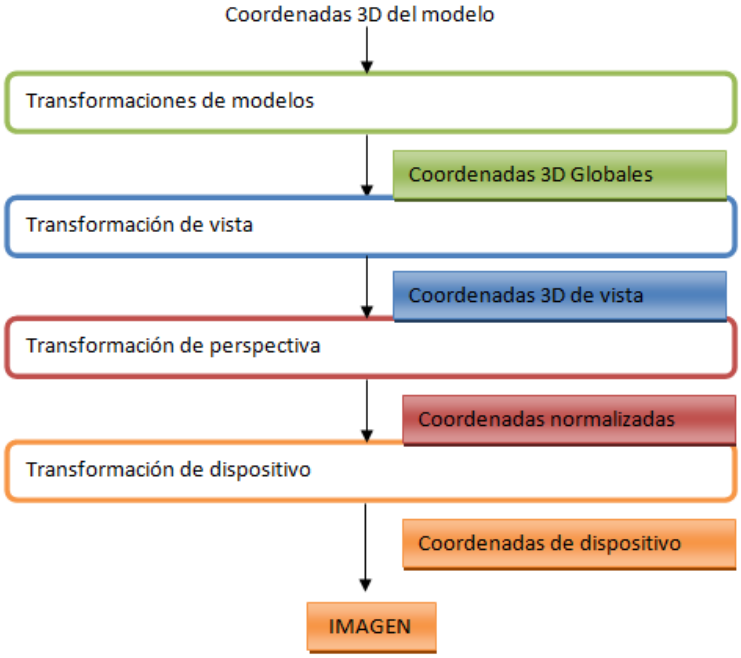


Figura 1.2: Paso de mundo 3D a imagen 2D

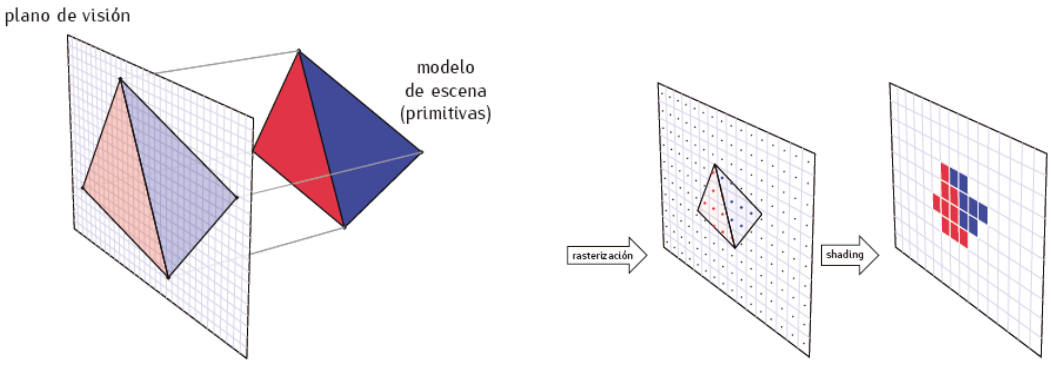


Figura 1.3: De primitivas a píxeles por rasterización

Para generar una imagen por rasterización, se siguen los siguientes pasos:

```
Inicializar el color de todos los pixeles

Para cada primitiva P de la escena a visualizar
    Encontrar el conjunto S de pixeles cubiertos por P

Para cada pixel s de S:
    Calcular el color de P en s
    Actualizar el color de s
```

2. Esquema básico del gestor de ventanas GLUT

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdio.h>
void dibuja() {
    ...
}
int main(&argc, argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
    glutInitWindowPosition(50,50);
    glutInitWindowSize(800,600);
    glutCreateWindow("Ejemplo");
    glutDisplayFunc(dibuja());
    glutReshapeFunc(dibuja());
    glutMainLoop();
    return(true);
}
```

Listing 1: Uso básico de GLUT

En el fragmento de codigo 1 se puede observar un ejemplo para inicializar una ventana con GLUT, donde:

- `glutInitDisplayMode` inicializa OpenGL.
- `glutInitWindowPosition`, `glutInitWindowSize` y `glutCreateWindow` inicializan la ventana.
- `glutDisplayFunc` establece la función gestora del evento de redibujado.
- `glutReshapeFunc` establece la función gestora del evento de redimensionado de ventan.
- `glutMainLoop` echa a andar la función de dibujado en un bucle infinito.

3. Primeras funciones de interés de OpenGL

En el fragmento de código 2 podemos observar qué significa cada parámetro de la función para el array de vértices de la primitiva, al igual que un uso simple del mismo junto a la función `glDrawElements`.

```
void glVertexPointer (GLint size, GLenum type, GLsizei stride, const
    GLvoid * pointer);

    size    Número de coordenadas por vértice: 2, 3 o 4.

    type   Tipo de cada coordenada del array (GL_SHORT, GL_INT, GL_FLOAT,
    o GL_DOUBLE)

    stride  Offset en bytes entre dos vertices consecutivos. Si es 0, se
    entiende que los vértices están consecutivos.

    pointer Puntero a la primera coordenada del primer vértice del array.

    Uso sencillo de glVertexPointer:

    float vertices[] = {
    -1.0f, 1.0f, 0.0f,
    1.0f, 1.0f, 0.0f,
    0.0f, 0.0f, 0.0f,
    };
    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(3, GL_FLOAT, 0, vertices);
    glDrawArrays(GL_TRIANGLES, 0, 3);
```

Listing 2: Array de vértices

La llamada a las instrucciones de definición de las primitivas puede hacerse única y exclusivamente desde la función de redibujado (o cualquiera de las invocadas desde ésta). Con OpenGL se puede utilizar, y de hecho se utiliza, un sistema de doble buffer para el dibujado, de forma que la rasterización se realiza en una imagen que no se ve, está oculta, hasta que no ejecutamos el intercambio de buffers. De esta forma no se ve cómo avanza el barrido del algoritmo raster (que va de arriba debajo de izquierda a derecha), y especialmente en las escenas animadas esto es muy importante. Se puede usar un único buffer, pero no es recomendable.

4. Representar modelos con OpenGL

Vamos a comenzar con una aclaración:

- **Geometría**. La geometría de un modelo viene dada por las posiciones o coordenadas de aquellos puntos que están sobre la superficie.
- **Topología**. La topología de un modelo es cómo se organiza la geometría para dar lugar a una superficie.

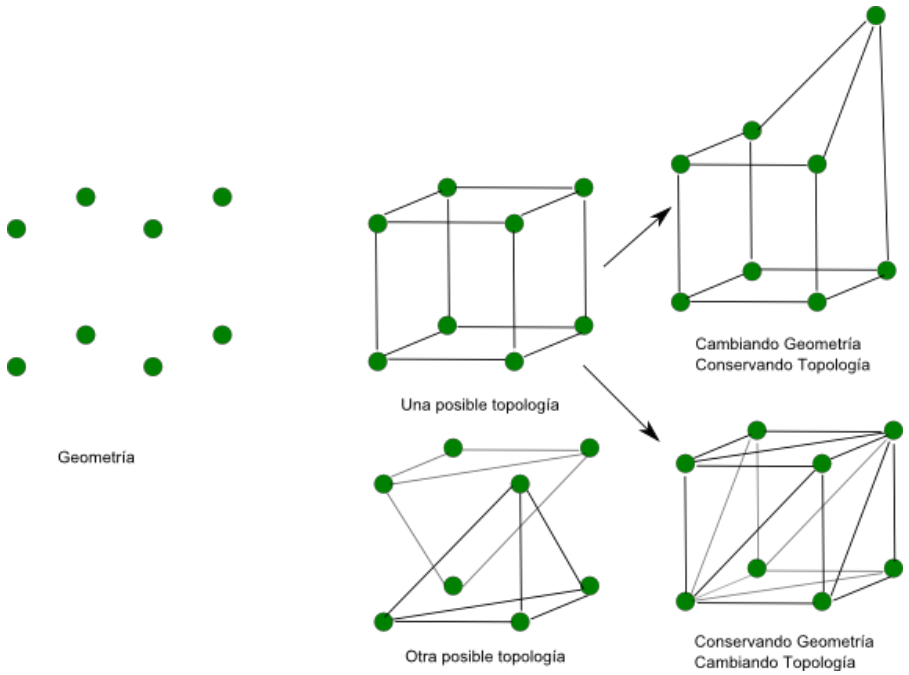


Figura 4.1: Relación geometría y topología

En la figura 4.1 podemos ver cómo geometría y topología no son cosas independientes a la hora de crear un modelo digital 3D, sino que están muy relacionadas. Se muestra cómo con ocho muestras espaciales, ocho vértices, se pueden generar bien un cubo o dos tetraedros, y a su vez, un cubo se puede representar cambiando la topología y conservando la geometria inicial. De igual manera, también se muestra cómo se puede conservar la topología y generar una figura distinta a un cubo.
A su vez, de la figura 4.1 también podemos obtener los siguientes términos:

- **Vértice**, posición en el espacio que corresponde con un punto de la superficie del objeto a representar.
- **Arista**, segmento de recta que une dos vértices. También se puede ver como el segmento que comparten dos caras adyacentes.

- **Cara**, polígono definido por una secuencia de vértices.

Esta estructura reticular de caras, aristas y vértices se denomina normalmente **malla de triángulos**, *triangle mesh* en inglés.
Lo que hay que hacer es intentar minimizar la redundancia en la información de los vértices y, además, proporcionar cierto tipo de información topológica.
Se puede entonces usar una estructura que indexe los vértices, y cuya principal información sean los triángulos. Se muestra gráficamente en la figura 4.2.

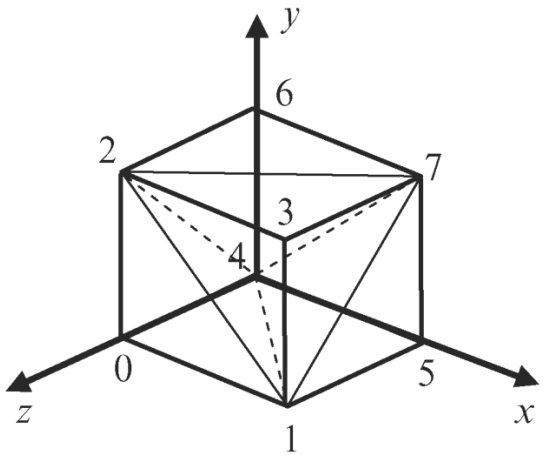


Tabla de vértices

x	y	z
0.0	0.0	1.0
1.0	0.0	1.0
0.0	1.0	1.0
1.0	1.0	1.0
0.0	0.0	0.0
1.0	0.0	0.0
0.0	1.0	0.0
1.0	1.0	0.0

Tabla de triángulos

v0	v1	v2
0	1	2
1	3	2
2	3	7
2	7	6
1	7	3
1	5	7
6	7	4
7	5	4
0	4	1
1	4	5
2	6	4
0	2	4

Figura 4.2: Representación visual de la tabla de vértices y triángulos

OpenGL proporciona una función para manejar los índices de vértices, de forma que, proporcionando un vector de vértices y uno de caras (con los índices de los vértices que las forman), se pinte todo.

```
void glDrawElements(GLenum mode, GLsizei count, GLenum type, const GLvoid
     * indices);

    mode     Especifica qué primitivas se van a dibujar: GL_POINTS,
    GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES, GL_LINE_STRIP_ADJACENCY,
    GL_LINES_ADJACENCY, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRIANGLES,
    GL_TRIANGLE_STRIP_ADJACENCY, GL_TRIANGLES_ADJACENCY o GL_PATCHES.

    count    Número de elementos que se van a dibujar.

    type     Tipo de los índices (GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT, o
    GL_UNSIGNED_INT).

    indices     Puntero al vector donde están los índices almacenados.
```

5. Aristas aladas

En las estructuras anteriores, las posibilidades de navegación por la malla son bastante limitadas, ya que no hay información de adyacencia entre caras ni una secuencia de aristas más o menos coherente.
Las estructuras de datos para mallas poligonales genéricas (no necesariamente de triángulos) están centradas en el concepto de arista, ya que podemos entender los triángulos como una ilusión óptica formada por la conectividad entre los vértices, es decir, por la red de aristas.
Cada arista almacena referencias a los vértices situados en los extremos de la arista, a las dos caras adyacentes a la arista, y a la siguiente y anterior arista en la cara izquierda y derecha. Además, cada vértice y cada cara almacenan una referencia a una de sus aristas.
Todo esto queda como se aprecia en la figura 5.1.

Vértice
Tupla3r P Natural arista
Arista
Natural V1, V2 Natural caraIzq, caraDrch Natural aristaSigIzq, aristaSigDrch Natural aristaAntIzq, aristaAntDrch
Cara
Natural arista

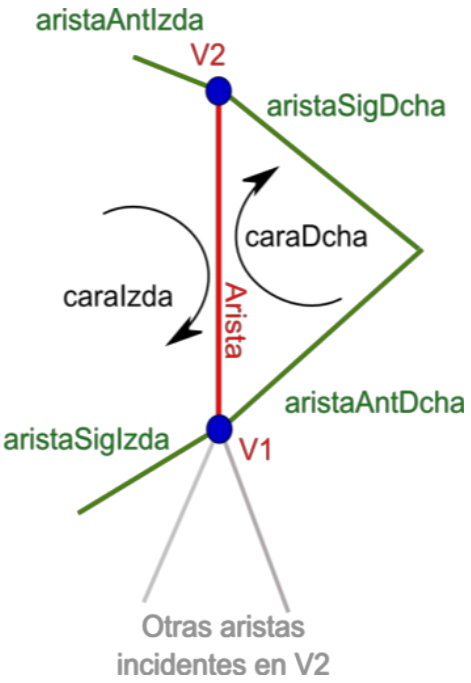


Figura 5.1: Representación de las aristas aladas

El hecho de almacenar la arista siguiente y anterior permite hacer recorridos por las entradas de la tabla de aristas:

- Dada una arista y una cara adyacente, se pueden saber las aristas de la cara o sus vértices.
- Dada una arista y uno de sus vértices, se pueden saber las aristas que inciden en el vértice o los triángulos que comparten ese vértice.

La estructura de aristas aladas es bastante elegante y potente, pero tiene una gran pega: hay que estar constantemente comprobando la orientación de la arista antes de decidir el paso a la siguiente arista.
La estructura de semiaristas aladas elimina la contrariedad anterior dividiendo cada arista en dos semiaristas orientadas en sentido antihorario de la cara a la que pertenece cada una, como se aprecia en la figura 5.2.

Vértice
Tupla3r P
Natural semiarista
Semiarista
Natural V
Natural cara
Natural semiaristaSig
Natural semiaristaAnt
Natural opuesta
Cara
Natural semiarista

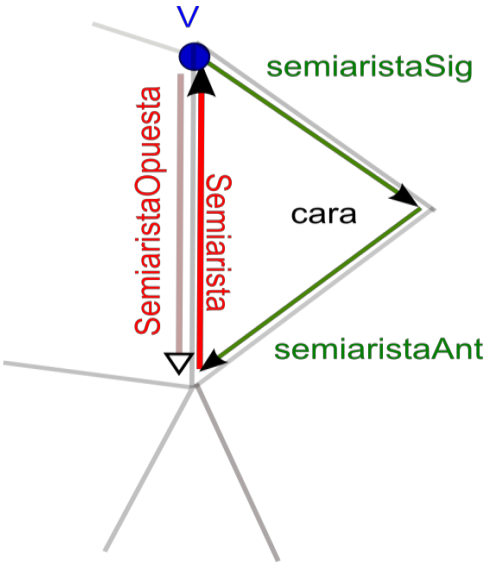


Figura 5.2: Representación de las semiaristas aladas

A partir de esta estructura de semiaristas aladas han aparecido evoluciones que optimizan aún más el espacio, pero son de menor implantación. Hoy en día, la mayoría de librerías que utilizan mallas de triángulos y conservan la información topológica, usan la estructura de datos de semiaristas aladas.

6. Atributos "primitivos" de las mallas

Por ahora nos vamos a centrar en dos importantes para la generación de escenas con cierta calidad estética:

- **Normales**: vectores de longitud unidad.
  - *normales de caras*: vector unitario perpendicular a cada cara, de longitud unidad, apuntando al exterior de la malla. Se precalcula a partir de la información de la cara.
  - *normales de vértices*: vector unitario perpendicular al plano tangente a la superficie en la posición del vértice.
- **Colores**: valores RGB o RGBA.
  - *colores de caras*: útil cuando cada cara representa un trozo de superficie de color homogéneo.
  - *colores de vértices*: color de la superficie en cada vértice. En este caso se supone que el color varía de forma continua entre los vértices.

Si se usan vertex arrays, se puede especificar un array de colores y otro de normales, donde habrá una entrada por cada vértice:

```
1 void glNormalPointer(GLenum type, GLsizei stride, const GLvoid * pointer)
2
3 void glColorPointer(GLint size, GLenum type, GLsizei stride, const GLvoid
      * pointer)
```

Si no se especifican estos array en el momento de dibujar los vértices, se usa el color actual y la normal actual a todos los vértices.

6.1. Normales

Las normales de caras son útiles cuando el objeto que se modela con la malla está realmente compuesto de caras planas. Si se aproxima un objeto curvo, las normales de los vértices se pueden aproximar a priori, bien porque conozcamos exactamente su valor (en el caso de una esfera) o bien promediando las normales de las caras que comparten dicho vértice.
Para un vértice v, con k caras adyacentes, su normal n se calcula como:

$$n = \frac{s}{\|s\|} \quad \text{donde} \quad s = \sum_{i=1}^{k} m_i$$

donde m_i son las normales de las caras que comparten dicho vértice.
Hay que tener claro que estas normales no son las reales del objeto, pues se desconoce la orientación exacta de la superficie en dicho punto, pero son una muy buena aproximación.

7. Transformaciones geométricas

en una escena con varios modelos, todos los vértices deben aparecer referidos a un único sistema de referencia común. Dicho sistema, denominado **sistema de referencia de la escena**, o del mundo (*world coordinate system*), hace que las coordenadas de los vértices no sean las que originalmente se dotaron al objeto, sino que se expresan en **coordenadas del mundo**.
Las transformaciones geométricas calculan coordenadas del mundo a partir de las coordenadas locales.
Una transformación geométrica T, se define matemáticamente como una aplicación que asocia a cualquier punto o vector p otro punto o vector q, y escribimos

$$q = Tp$$

que se lee "q es T aplicado a p".
Una transformación T cambia las coordenadas de los puntos sobre los que actúa.
Existen varias transformaciones geométricas simples que son muy útiles en Informática Gráfica para la definición de escenas y animaciones, y que constituyen la base de otras transformaciones:

- **Translación**. Consiste en desplazar todos los puntos del espacio de igual forma, es decir, en la misma dirección y la misma distancia.
- **Escalado**. Supone estrechar o alargar las figuras en una o varias direcciones.
- **Rotación**. Rotar los puntos un ángulo dado en torno a un eje de rotación.
- **Cizalla**. Se puede ver como un desplazamiento de todos los puntos en la misma dirección, pero con distintas distancias.

Estas transformaciones se aprecian en la figura 7.1

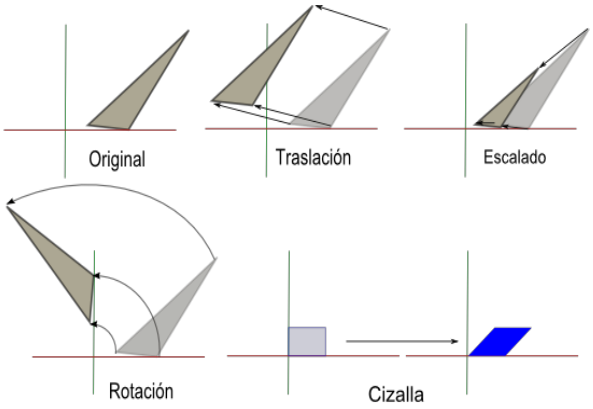


Figura 7.1: Transformaciones geométricas básicas

7.1. Translación

Si definimos la traslación como una función $f(x, y, z)$ (resumido sería $T[dx, dy, dz]$) que afecta a las coordenadas del punto p, podemos definirla en 3D como

$$p'_x = p_x + dx$$
$$p'_y = p_y + dy$$
$$p'_z = p_z + dz$$

7.2. Escalado

La transformación de escalado viene determinada por tres valores reales (e_x, e_y, e_z).
Equivale a un cambio de escala o tamaño con centro en el origen del sistema de coordenadas, es decir, que el origen queda inalterable.
Si definimos el escalado como una función $f(x, y, z)$ (resumido sería $E[e_x, e_y, e_z]$) que afecta a las coordenadas del punto p, podemos definirla en 3D cómo

$$f_x(p) = p'_x = e_x p_x$$
$$f_y(p) = p'_y = e_y p_y$$
$$f_z(p) = p'_z = e_z p_z$$

Cuando los tres valores de escala, e_x, e_y, e_z tienen el mismo valor, se dice que es un escalado uniforme.
Los escalados uniformes conservan las proporciones y los ángulos de los objetos, y se puede abreviar escribiendo $E[e]$.

$$[p'_x \quad p'_y \quad p'_z] = [p_x \quad p_y \quad p_z] \begin{bmatrix} e_x & 0 & 0 \\ 0 & e_y & 0 \\ 0 & 0 & e_z \end{bmatrix}$$

7.3. Rotacion

Podemos realizar rotaciones 2D.
Podemos construir la transformación de rotación $R[\alpha]$ en torno al origen O un ángulo α radianes.

$$[p'_x \quad p'_y] = [p_x \quad p_y] \begin{bmatrix} cos(\alpha) & sin(\alpha) \\ -sin(\alpha) & cos(\alpha) \end{bmatrix}$$

En 3D hay tres posibles rotaciones básicas, una en torno a cada eje del sistema de coordenadas. Las llamaremos $R_x[\alpha]$, $R_y[\alpha]$ y $R_z[\alpha]$. Visualmente lo podemos ver en la figura 7.2.
Como características tenemos:

- Cada rotación modifica dos coordenadas y deja la del eje que se usa para la rotación intacta.
- Son rotaciones siempre en sentido antihorario para α>0.

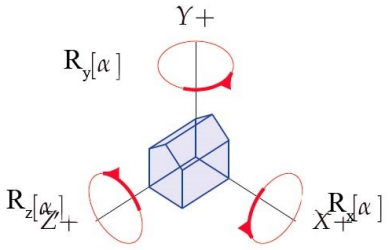


Figura 7.2: Rotaciones en los distintos ejes

Al ser tres rotaciones distintas, que afectan a ejes distintos, ya no hay una única fórmula de la rotación, sino que hemos de definir tres funciones distintas. Estas funciones se pueden expresar de la siguiente manera:

$R_x[\alpha]$	$R_y[\alpha]$	$R_z[\alpha]$
$p'_x = p_x$ $p'_y = cos(\alpha)p_y - sin(\alpha)p_z$ $p'_z = sin(\alpha)p_y + cos(\alpha)p_z$	$p'_x = cos(\alpha)p_x + sin(\alpha)p_z$ $p'_y = p_y$ $p'_z = -sin(\alpha)p_x + cos(\alpha)p_z$	$p'_x = cos(\alpha)p_x - sin(\alpha)p_y$ $p'_y = sin(\alpha)p_x + cos(\alpha)p_y$ $p'_z = p_z$

O bien en forma matricial:

$R_x[\alpha]$	$R_y[\alpha]$	$R_z[\alpha]$
$\begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\alpha) & sin(\alpha) \\ 0 & -sin(\alpha) & cos(\alpha) \end{bmatrix}$	$\begin{bmatrix} cos(\alpha) & 0 & -sin(\alpha) \\ 0 & 1 & 0 \\ sin(\alpha) & 0 & cos(\alpha) \end{bmatrix}$	$\begin{bmatrix} cos(\alpha) & sin(\alpha) & 0 \\ -sin(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}$

De forma que la rotación en torno al eje X del punto p sería:

$$[p'_x \quad p'_y \quad p'_z] = [p_x \quad p_y \quad p_z] \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\alpha) & sin(\alpha) \\ 0 & -sin(\alpha) & cos(\alpha) \end{bmatrix}$$

7.4. Composición de transformaciones

Una transformación C puede obtenerse como composición de otras dos transformaciones: A(primero) y B(después). En ese caso, se escribe $C = BA$.
La composición es, en general, **no conmutativa**, y se puede extender a 3 o más transformaciones:

$$T_4(T_3(T_2(T_1(p)))) = (T_4T_3T_2T_1)p$$

La composición es asociativa, y se interpreta de derecha a izquierda, es decir, es como si se aplicara primero T_1, luego T_2, a continuación T_3 y finalmente T_4.

7.5. Coordenadas homogéneas

Las coordenadas homogéneas nos permiten representar de la misma manera puntos y vectores en el espacio afín.
El espacio afín 2D (x,y) es un subespacio del espacio proyectivo 3D (X,Y,W), definido como los puntos con coordenada W = 1.
De la misma manera, el espacio afín 3D (x,y,z) es un subespacio del espacio proyectivo 4D (X,Y,Z,W), definido como los puntos con coordenada W=1.
Dicha esta introducción matemática, podemos expresar cualquier punto p(x,y,z) como

$$P = [x\ y\ z\ 1]$$

en coordenadas homogéneas 4D. De hecho, en realidad la conversión es

$$P^{3D}[x, y, z] = [P_x^{4D}/P_w^{4D}, \quad P_y^{4D}/P_w^{4D}, \quad P_z^{4D}/P_w^{4D}]$$

Por ejemplo, si $P^{4D} = [3, 4, 6, 2]$, el resultado al pasar al espacio afín 3D es $P^{3D} = [1{,}5, 2, 3]$
¿Y esto qué implicación tiene? Pues que todas las matrices anteriormente presentadas como 3x3 pueden expresarse, invariantemente como 4x4:

$$E[e_x, e_y, e_z, 1] = \begin{bmatrix} e_x & 0 & 0 & 0 \\ 0 & e_y & 0 & 0 \\ 0 & 0 & e_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$R_x[\alpha]$	$R_y[\alpha]$	$R_z[\alpha]$
$\begin{bmatrix} cos(\alpha) & 0 & -sin(\alpha) & 0 \\ 0 & 1 & 0 & 0 \\ sin(\alpha) & 0 & cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\alpha) & sin(\alpha) & 0 \\ 0 & -sin(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} cos(\alpha) & sin(\alpha) & 0 & 0 \\ -sin(\alpha) & cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Y como novedad, podemos expresar la matriz de translación:

$$T[dx, dy, dz, 1] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ dx & dy & dz & 1 \end{bmatrix}$$

7.6. Transformaciones en OpenGL

OpenGL almacena, como parte de su estado, una matriz 4x4 M que codifica una transformación geométrica, y que se llama *modelview matrix*(matriz de modelado y vista). Esta matriz se puede ver como la composición de dos matrices, V y N:

- N es la matriz de modelado que posiciona los puntos en su lugar en coordenadas del mundo.
- V es la matriz de vista, que posiciona los puntos en su lugar de coordenadas relativas a la cámara.

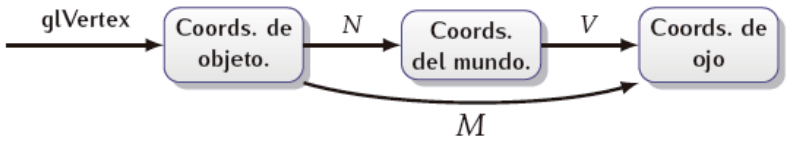


Figura 7.3: Transformaciones de la matriz *Modelview* realizadas por openGL

La matriz *modelview* se puede especificar en OpenGL mediante estos pasos:

1. Hacer la llamada `glMatrixMode(GL_MODELVIEW)`para indicar que las siguientes operaciones operan sobre la matriz modelview M.
2. Hacer `glLoadIdentity` para hacer M igual a la Identidad.
3. Usar `gluLookAt` u otras para hacer una matriz de vista V.
4. Usar una o varias llamadas para componer la matriz de modelado N:
   - `glRotatef( GLfloat a, GLfloat ex, GLfloat ey, GLfloat ez )`
   - `glTranslatef( GLfloat dx, GLfloat dy, GLfloat dz )`
   - `glScalef( GLfloat sx, GLfloat sy, GLfloat sz )`
   - `glMultMatrixf( GLfloat * A )`

Al final se obtiene que $M = VN$.
OpenGL construye M a partir de las matrices proporcionadas en los pasos 3 y 4.

8. Modelos jerárquicos

Pensemos en modelar un árbol. La forma más simple puede ser usar un cilindro para el tronco y una esfera para la copa. El cilindro y la esfera serían los objetos simples que forman el objeto compuesto *árbol*.
La estructura que representa este tipo de objetos, o incluso las escenas, se denomina grafo de escena. Un grafo de escena es un *grafo dirigido acíclico*, donde:

- Cada objeto compuesto es un subgrafo dentro del grafo.
- Cada objeto simple es un nodo terminal.
- Cada arco une dos nodos, y estos nodos pueden ser transformaciones geométricas, nodos terminales o nodos grupo que permiten unir elementos del grafo.

La visualización de modelos jerárquicos en OpenGL se basa en operaciones que permiten guardar y recuperar la matriz modelview. Se puede hacer de dos formas:

- Realizando operaciones de multiplicación de matrices, como realmente ocurre en la tarjeta gráfica, y usar la matriz identidad para cada inicio de recorrido del grafo.
- Usando el mecanismo de pila de OpenGL para recuperar versiones temporales de las matrices.

Internamente, OpenGL tiene un mecanismo de pilas para las transformaciones geométricas, que guarda en la pila el estado actual.
Cuando se realiza un `glPushMatrix` lo que se hace es duplicar el tope de la pila, de forma que se almacenan dos copias de la matriz actual.
La que queda en el tope es la única que se consulta y se ve afectada por las operaciones de transformación, mientras que la que hay debajo sólo será utilizada tras realizar un `glPopMatrix`.
Podemos verlo gráficamente en la figura 8.1.

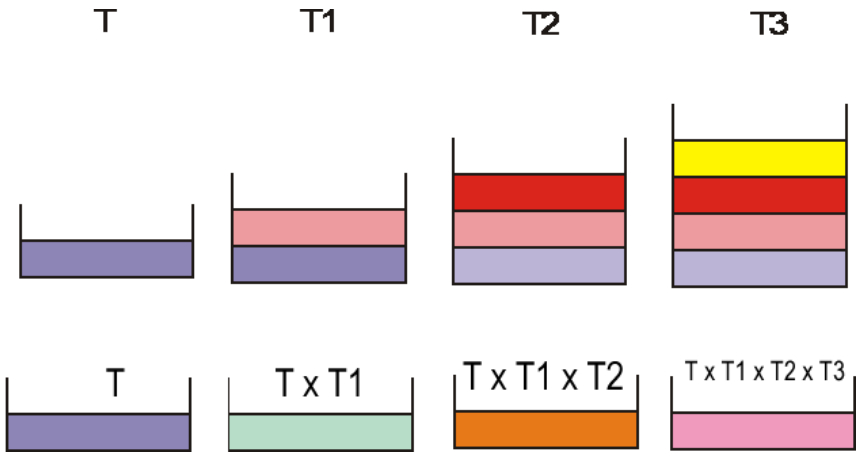


Figura 8.1: Gestión de la pila realizada por OpenGL usando `glPushMatrix` y `glPopMatrix` (arriba) y sin usarlos (abajo)

El código entre *push* y *pop* es neutro con respecto a M, lo que quiere decir que tras hacer un *pop*, la matriz M vale exactamente lo que valía cuando se hizo *push*. Lógicamente los *push* y *pop* han de estar balanceados, a lo que puede ayudar realizar una indentación en el código que permita visualizar rápidamente los distintos niveles de la pila.

8.1. Parametrización de las transformaciones

Un grafo de escena parametrizado se traduce en objetos con distinta geometría para distintos valores concretos de los parámetros. Este número de parámetros denota los **grados de libertad** del modelo jerárquico.
Estos grados de libertad, estos modelos jerárquicos parametrizables, permiten la realización de partes móviles, insertando un nodo transformación parametrizado para que modifique los elementos móviles.

9. Iluminación

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.
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**.
La luz en OpenGL tiene las siguientes características:

- Las fuentes de luz son puntuales o unidireccionales, y hay un número finito de ellas (hasta 8).
- 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.
- Los objetos o polígonos son totalmente opacos (no hay transparencias ni materiales translúcidos).
- No se consideran sombras arrojadas, es decir, un objeto no impide la trayectoria de la luz.
- El espacio entre los objetos no dispersa la luz (ésta tiene igual densidad independientemente de la distancia a la fuente).
- En lugar de considerar todas las longitudes de onda posibles, se usa el modelo RGB.

Para este modelo simplificado que usa OpenGL, podemos considerar dos tipos genéricos de reflexiones en la superficie: la reflexión **especular** y la **difusa**, como se aprecia en la figura 9.1.

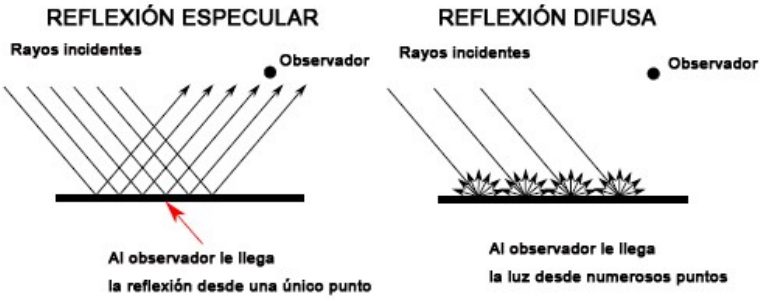


Figura 9.1: Reflexión especular y difusa

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.

9.1. Fuentes de luz

En OpenGL hay dos tipos de fuentes de luz, representadas gráficamente en la figura 9.2:

- 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.

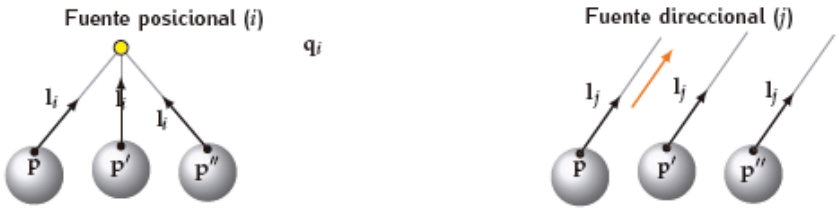


Figura 9.2: Fuentes de luz posicionales y direccionales

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.
n_p 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.

9.2. Materiales

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*.
Un material se puede 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.

9.3. 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`.

9.4. Configuración de una fuente de luz

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

```
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".
Las posiciones (en luces posicionales) o direcciones (en direccionales) se especifican 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 X**: si se especifica cuando $M = VN$ (donde N es la matriz de modelado del objeto X)

Esto permite crear luces que permanezcan fijas, estacionarias en un lugar del mundo, o moviéndose con un objeto. 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`.

9.5. Dirección de la luz 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() ;
```

9.6. Especificación de normales en vértices

Cada vez que se llama a `glDrawElements`, o `glDrawArrays`, si está la iluminación encendida, es necesario que para cada vértice haya un vector normal.
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 );
```

Los vectores normales especificados en nv 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)`.

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

  `glEnable(GL_NORMALIZE)`.

9.7. Propiedades del material

Las propiedades del material también forman parte del estado y se modifican con llamadas a la función `glMaterial`.
Para modificar la emisividad, el resto de colores y el componente de brillo e hacemos:

```
//Cambiamos la emisividad
GLfloat color[4] = { r, g, b, 1.0 } ;

glMaterialf( GL_FRONT_AND_BACK, GL_EMISSION, color );

//Cambiamos el resto de colores
GLfloat color[4] = { r, g, b, 1.0 };

glMaterialfv( GL_FRONT_AND_BACK, GL_AMBIENT, color );
glMaterialfv( GL_FRONT_AND_BACK, GL_DIFFUSE, color );
glMaterialfv( GL_FRONT_AND_BACK, GL_SPECULAR, color );

//Cambiamos el brillo
glMaterialf( GL_FRONT_AND_BACK, GL_SHININESS, v ); //Donde v ya tiene un
    valor asociado
```

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`)

```
glColorMaterial( GL_FRONT_AND_BACK, GL_EMISSION );

glColorMaterial( GL_FRONT_AND_BACK, GL_AMBIENT );

glColorMaterial( GL_FRONT_AND_BACK, GL_DIFFUSE );

glColorMaterial( GL_FRONT_AND_BACK, GL_SPECULAR );

glColorMaterial( GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE );
```

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` para la cara delantera de los polígonos.
- `GL_BACK` para la cara trasera de los polígonos.
- `GL_FRONT_AND_BACK` para ambas caras.

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.

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.

10. Texturas

Una textura 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 píxeles RGB (una imagen discretizada), a cuyos píxeles se les llama **texels** (**tex**ture **el**ements). A esta imagen se le llama **imagen de textura**.
En la figura 10.1 podemos ver como se asigna una textura a los vértices de un objeto.

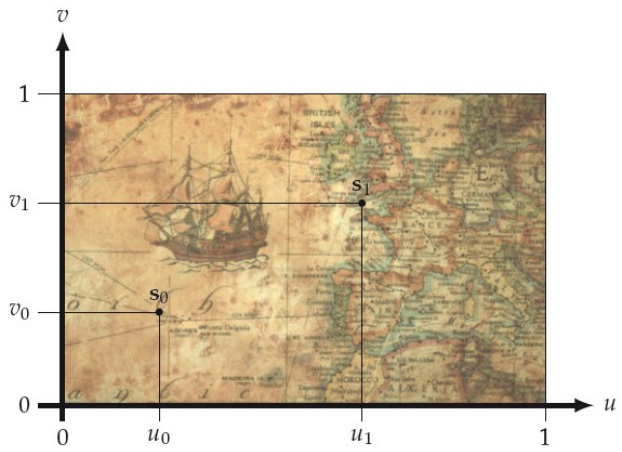


Figura 10.1: Visualización de la asignación de texturas y uso de la función T

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.
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 $(v_0, u_0), (v_1, u_1), \ldots (u_{n-1}, v_{n-1})$. Se puede hacer:
  - Manualmente en objetos sencillos.
  - De forma asistida usando software para CAD (p.ej. 3D Studio).

  Esta modalidad hace necesario realizar una interpolación de coordenadas de textura en el interior de los polígonos de la malla.

- **Asignación procedural**. La función f se implementa como un 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*.

10.1. Asignación procedural por coordenadas paramétricas

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.

10.2. 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:

$$\alpha = atan2^2(z, x) \qquad h = y$$

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

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

[2] atan2 calcula el ángulo que forman el punto dado y el eje de coordenadas

10.3. 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.
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:

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

De esta forma α está en el rango $[-\pi, \pi]$ y β está en el rango $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$, pudiendo calcular u y v como sigue:

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

10.4. Uso de texturas en OpenGL

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.
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`).
En el estado interno de OpenGL hay en cada momento un identificador de textura activa, para cambiar el identificador de textura activa podemos hacer:

```
glBindTexture( GL_TEXTURE_2D, idTex);
```

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 *)
);
```

Si es posible usar GLU, hay una alternativa preferible a `glTexImage2D`:

```
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.

10.5. Asignación explícita de texturas

En el estado interno hay un par de coordenadas de textura (s, t) que se asignan a cada vértice.
Si los vértices se envían con `glDrawElements`, es necesario indicar antes donde está la tabla de coordenadas de textura, esto se hace con `glTexCoordPointer`:

```
1  glVertexPointer( 3, GL_FLOAT, 0, ver );
2  glNormalPointer( GL_FLOAT, 0, nv);
3  glTexCoordPointer( 2, GL_FLOAT, 0, ctv);
4
5  glEnableClientState( GL_VERTEX_ARRAY );
6  glEnableClientState( GL_NORMAL_ARRAY );
7  glEnableClientState( GL_TEXTURE_COORD_ARRAY );
8
9  glDrawElements( GL_TRIANGLES, 3L*num_tri, GL_UNSIGNED_INT, tri );
10
11 glDisableClientState( GL_VERTEX_ARRAY );
12 glDisableClientState( GL_NORMAL_ARRAY );
13 glDisableClientState( GL_TEXTURE_COORD_ARRAY );
```

11. Cámara

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.

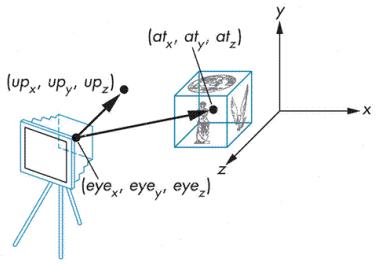


Figura 11.1: Esquematización de una cámara con los parámetros Eye, At y Up

En la figura 11.1 se muestra esquemáticamente los parámetros de la función `gluLookAt`.
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.
Otra forma de definir la posición y orientación de la cámara es con los siguientes parámetros, visualizados en la figura 11.2:

- **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.

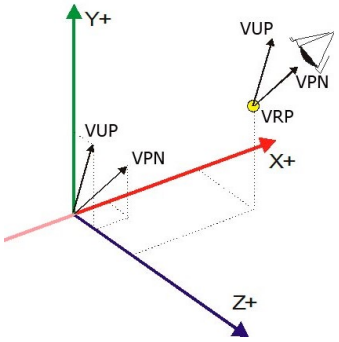


Figura 11.2: VRP, VPN y VUP

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 figura 11.3, donde vemos que los ejes de este sistema de coordenadas vienen dados por **n**, **v** y **u**:

- **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[3] entre estos dos vectores. ¡Ojo que el orden afecta!

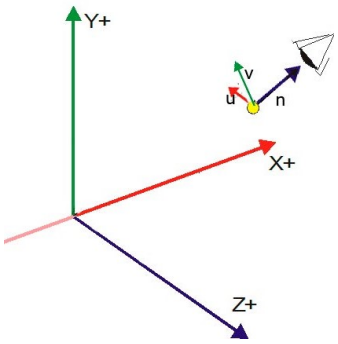


Figura 11.3: Cambio del sistema de referencia

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

- **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

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

La transformación de vista 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:

$$d_x = -VRP * u$$
$$d_y = -VRP * v$$
$$d_z = -VRP * n$$

[3]Siendo $\vec{u}$ y $\vec{v}$ vectores en $\Re^3$, el producto vectorial $\vec{u} \times \vec{v} = \vec{x} = \left| \begin{pmatrix} \vec{i} & \vec{j} & \vec{k} \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{pmatrix} \right|$

11.1. Frustrum

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. Los parámetros de este tipo de visualización se observan en la figura 11.4.

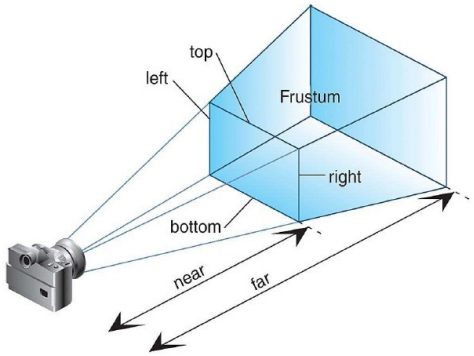


Figura 11.4: Visualización del frustrum

En una proyección ortográfica, el frustrum será un paralelepípedo, mientras que en una perspectiva, será una pirámide truncada.
Es importante tener claro que aunque se mueva la cámara, 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.

11.2. Proyecciones de perspectiva

¿Qué elementos conforman una operación de proyección? Se muestran gráficamente en la figura 11.5.

- El **proyector** es una recta que pasa por el punto a[4] 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.

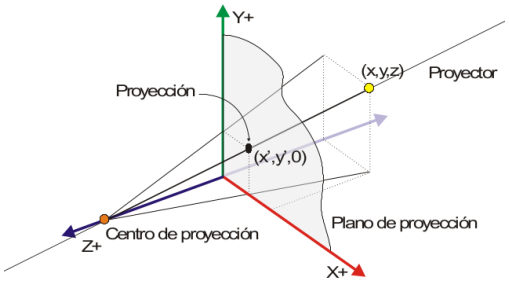


Figura 11.5: Elementos de una proyección

[4] (x, y, z)

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.

11.3. Proyección perspectiva

Para obtener el punto, realizamos una transformación de coordenadas, ya que como vemos en la figura 11.6, los valores de x e y varían en un función de z.

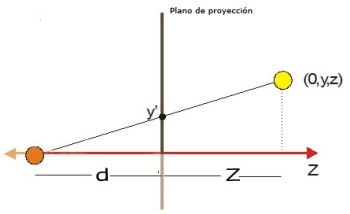


Figura 11.6: Coordenada 2D en una proyección perspectiva

$$\begin{bmatrix} x' & y' & z' & w' \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{d} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Lo cual da un punto $\begin{bmatrix} x & y & 0 & \frac{z}{d}+1 \end{bmatrix}$

Que al pasar a 3D dan los valores

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

11.4. Proyección ortogonal

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

Entonces

$$x' = x$$
$$y' = y$$

11.5. Matriz de proyección

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.
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**.

La librería GLU (GL Utilities), que viene "de serie" con OpenGL nos proporciona otra función que define la proyección de perspectiva con otros parámetros, mostrados en la figura 11.7:

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

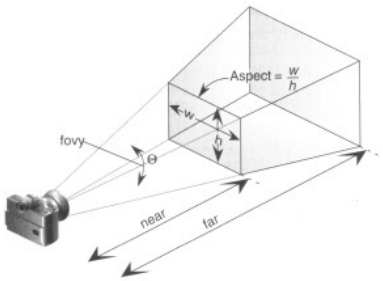


Figura 11.7: Perspectiva de GLU

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.

11.6. Recortado

Fijémonos en las matrices de las secciones anteriores. El valor de Z desaparece en esas matrices.
Realmente 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. 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.

11.7. Eliminación de partes ocultas (Z-Buffer)

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**. Para cada primitiva que se dibuje, se almacena en el píxel 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*).
En la figura 11.8 se puede ver como funciona.

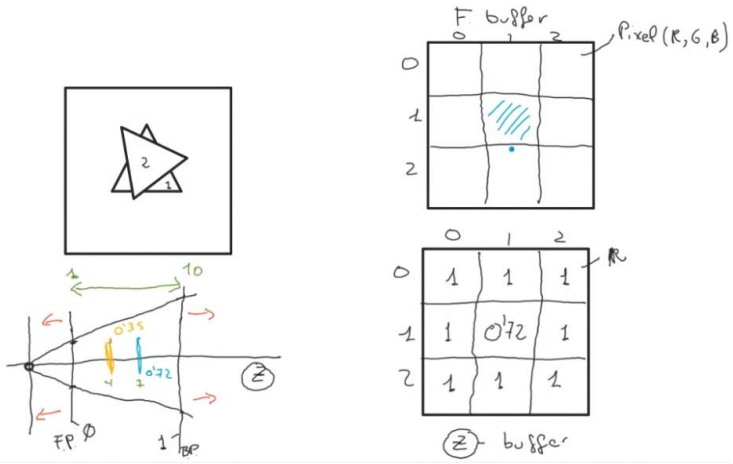


Figura 11.8: Funcionamiento del ZBuffer

Cuando se fuera a pintar el triángulo amarillo, como 0.35<0.72, se pintaría y se sustituiría el valor 0.72 del ZBuffer por el 0.35.
La principal ventaja de este algoritmo es que en el peor de los casos su complejidad es proporcional al número de primitivas.
En OpenGL, cada vez que se dibuja, igual que limpiamos el buffer de color, hay que limpiar el ZBuffer.