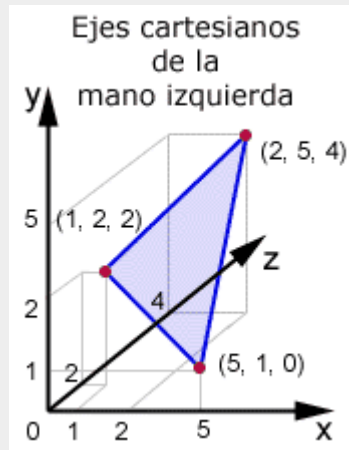


## Capítulo 0 - Introducción a Direct3D

### Sistema de coordenadas

Direct3D utiliza el sistema de coordenadas de la mano izquierda (left handed, OpenGL utiliza el opuesto, right handed) y por lo tanto sus ejes quedan definidos así:

- El eje X positivo va hacia la derecha.
- El eje Y positivo va hacia arriba.
- El eje Z positivo se aleja de nosotros (hacia dentro de la pantalla).



Para definir un vértice en el espacio 3D necesitamos especificar sus 3 componentes de coordenadas: x, y, z.

El origen de coordenadas es 0,0,0.

La posición de los vértices de un triángulo de ejemplo se describirían así:

```
v1( 1, 2, 2 )  
v2( 2, 5, 4 )  
v3( 5, 1, 0 )
```

Entonces, si nos fijamos en el dibujo anterior, podemos decir que cualquier vértice se puede especificar mediante un vector que vaya desde el origen de coordenadas al punto 3D en el espacio.

Direct3D tiene dos estructuras para definir vectores: **D3DVECTOR** y **D3DXVECTOR3**.

**D3DVECTOR** se encuentra definida en d3dtypes.h, y maneja las rutinas matemáticas que necesitas para usar vectores. **D3DXVECTOR3** está definida en d3dx8math.h y es una versión mejorada de la anterior estructura.

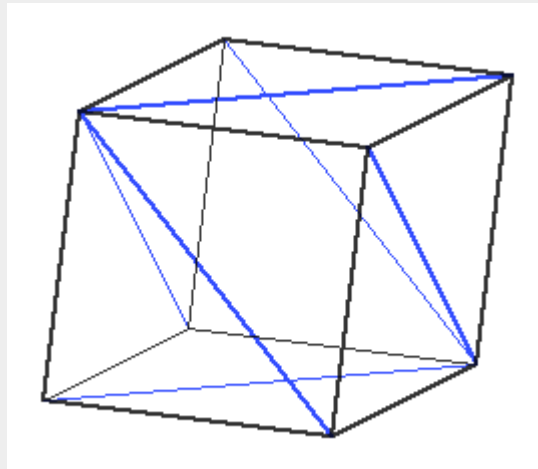
### Caras y normales

Cuando queremos representar cualquier objeto 3D necesitamos recurrir a puntos, líneas y sobretodo triángulos. ¿Por qué triángulos?

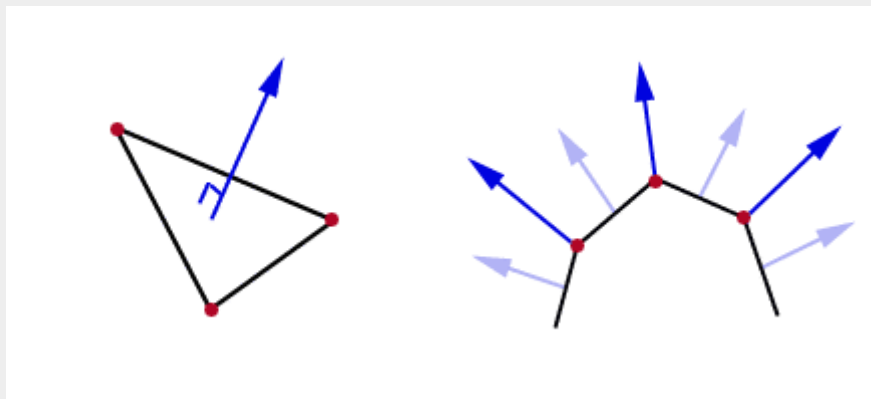
Muy sencillo, porque es la unidad mínima para modelar cualquier objeto complejo, y porque las tarjetas de video actuales trabajan muy rápidamente haciendo cálculos con triángulos.

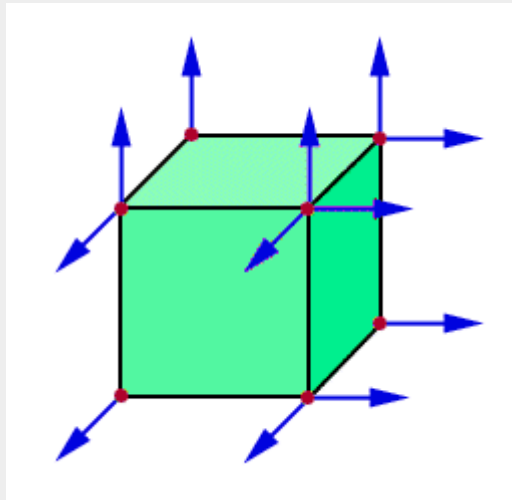
Así tenemos que cualquier cara de un objeto se va a dividir en triángulos.

Veamos el siguiente dibujo, un simple cubo de 6 caras esta compuesto por 12 triángulos:



Direct3D solo dibuja las caras frontales de los objetos, que por defecto son aquellas caras que formen los vértices agrupados de acuerdo al sentido de las agujas del reloj. Este proceso se denomina "backface culling" y se basa en eliminar aquellos triángulos que no miran a la cámara por medio de sus normales. En Direct3D aplicamos las normales para conocer la dirección a la que apunta una cara, o dicho de otra manera, la parte visible de una cara. Normal es el vector unitario perpendicular a la cara. Generalmente podemos definir normales en cada uno de los vértices de la cara. En los siguientes dibujos podemos ver los vectores que representan las normales de las caras, o de los vértices.

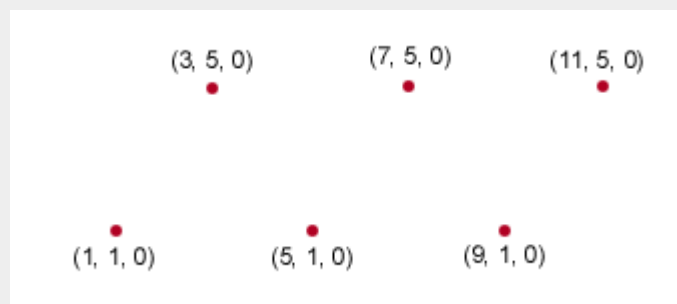




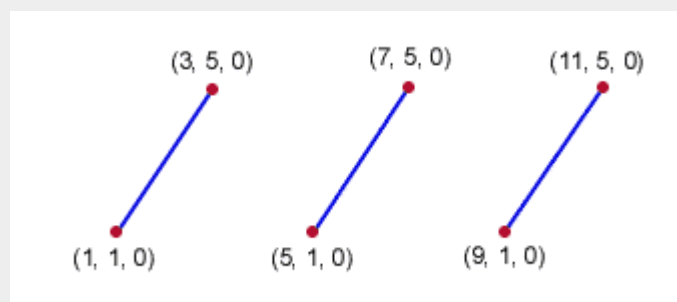
## Primitivas

Para indicar a Direct3D cómo tiene que interpretar una lista de vértices que le pasemos para que los pueda pintar como objetos 3D hay varias formas de hacerlo. Son las siguientes:

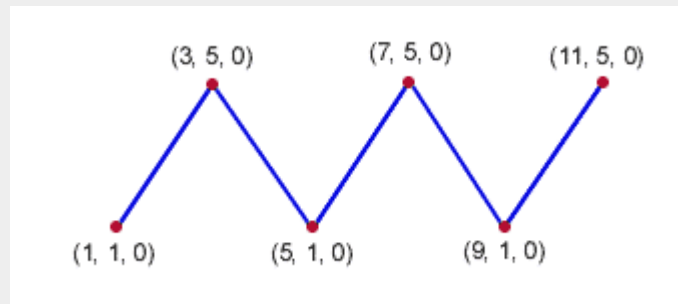
**Point List:** Una lista de puntos (corriente y moliente). Vale para apenas poco, en algunos juegos como Half-Life se puede ver en algún sistema de partículas pero poco más... Hay que puntualizar que estos puntos siempre se verán en pantalla con el tamaño de un pixel, independientemente de la distancia a la que se encuentren.



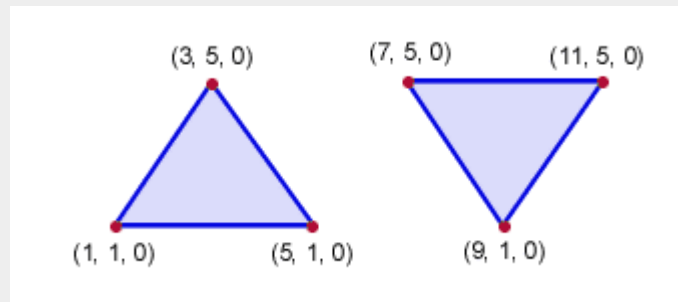
**Line List:** Una lista de líneas. Sin necesidad de seguir ningún orden, simplemente cada dos puntos es una línea. Por ejemplo para hacer lluvia...



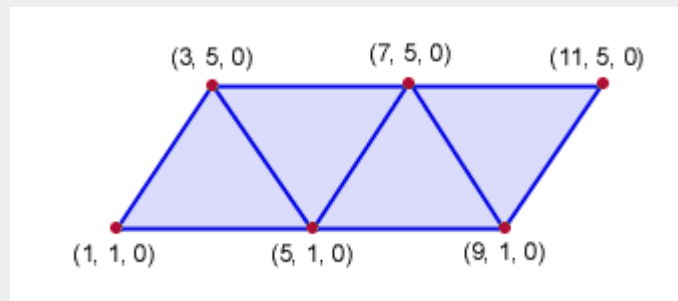
**Line Strip:** Literalmente una "tira de líneas". Son líneas unidas entre sí por un vértice.



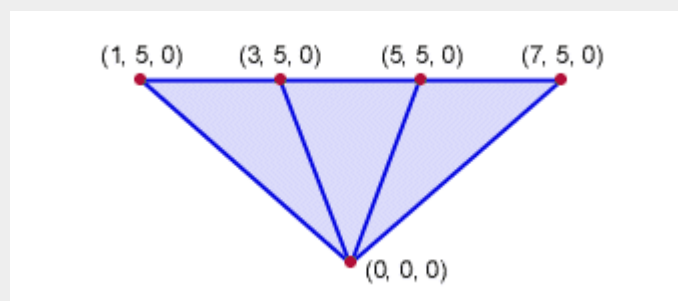
**Triangle List:** Una lista de triángulos sin orden alguno. Cada 3 vértices se forma un triángulo.



**Trinagle Strip:** Una "tira de triángulos" que comparten dos de sus vértices. De esta forma ahorramos muchos vértices en comparación con Triangle List.

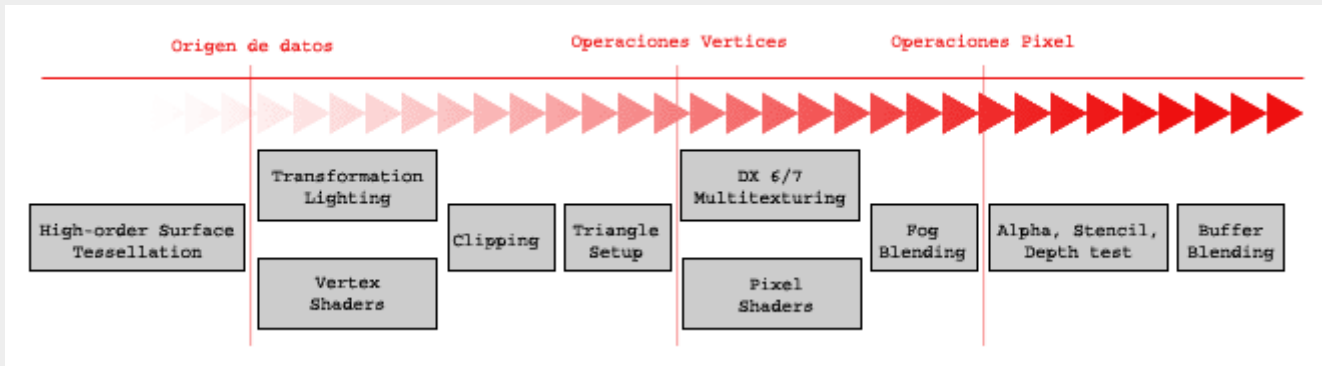


**Triangle Fan:** Un conjunto de triángulos que comparten un vértice entre todos. Una de las mejores formas de especificar nuestros objetos 3D.



La tubería de Direct3D

Esta es la traducción literal de "Direct3D pipeline" y viene a significar, más o menos, cómo funciona Direct3D si lo comparamos con una tubería (o una cadena de producción). Las cosas entran por un lado, se realizan una serie de operaciones, y salen por el otro ya hechas. Lo importante de esta cadena es entender en qué orden se efectúan cada una de las operaciones:



Una de las novedades respecto a modelos anteriores son los Vertex y Pixel Shaders. Algunos programadores prefieren implementar ellos mismos sus propios algoritmos de transformación y efectos de luz, es decir, que se pueden desactivar algunas partes de esta "tubería" y enviar la información a sus propias rutinas.

Gracias a los nuevos procesadores de las tarjetas gráficas actuales, las operaciones de transformación y de luz (de ahora en adelante T&L, de "Transformation and Lighting") se realizan en la propia tarjeta gráfica, en vez de en la CPU. Así que una vez liberada la CPU de la carga de realizar las operaciones de T&L, podemos usarla para otros fines en los juegos tales como Inteligencia Artificial y complejos cálculos físicos.

### La Tubería de transformación

Una de las partes importantes de esta "tubería" es la parte que se dedica a la Transformación (Transformation & Lighting o Vertex shaders).

Básicamente consiste en las operaciones que realizamos para transformar los valores de nuestro mundo 3D entre los diferentes espacios que lo componen. Es decir, descomponemos nuestro mundo 3D de acuerdo a diferentes momentos o diferentes sistemas de coordenadas más sencillos para trabajar con él, ya que si no sería demasiado complicado trabajar sobre él. Estos diferentes sistemas se llaman "espacios". Estos "espacios" son: espacio-modelo, espacio-mundo, espacio- vista y espacio-proyección.

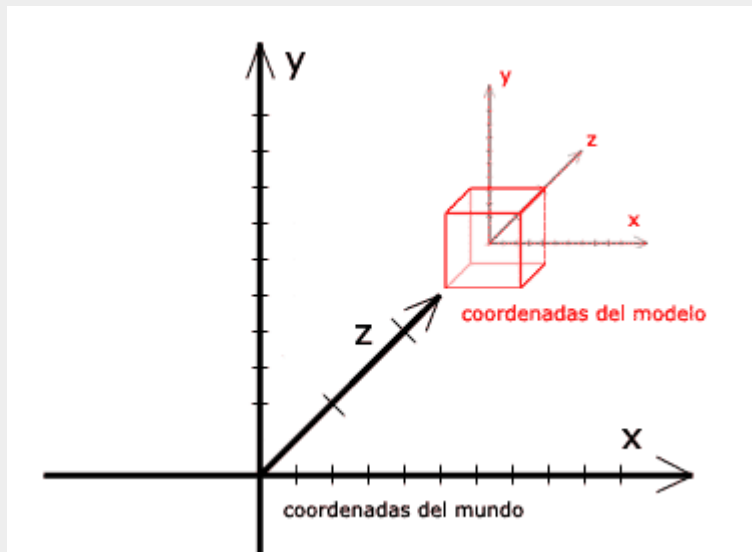


Esta tubería transforma cada vértice de un objeto, desde un punto abstracto en el sistema de coordenadas, a un píxel de la pantalla. Teniendo en cuenta las propiedades de la cámara virtual desde la que se ve la escena. Para realizar estas transformaciones necesitamos 3 matrices que definan nuestros

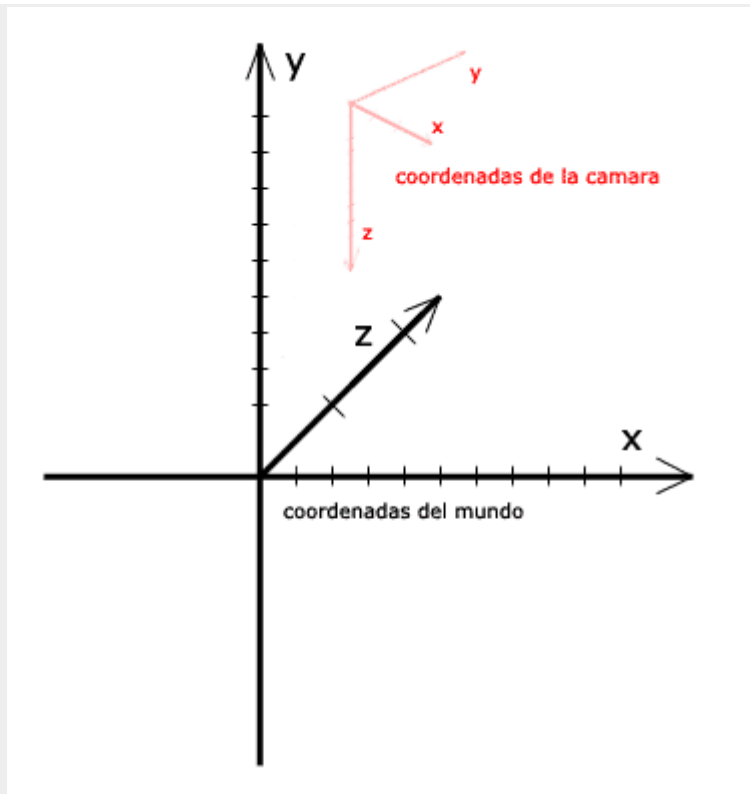
"espacios". La Matriz-Mundo, la Matriz-Vista y la Matriz-Proyección.

El primero de los pasos, "Transformar Mundo" lo que hace es transformar un objeto desde el espacio-modelo al espacio-mundo. "Espacio-modelo" es el sistema de coordenadas en el que definimos el objeto, sin tener en cuenta nada más. En este modelo es en el que podemos rotar el objeto, cambiarle el tamaño o trasladarlo de lugar para crear animaciones.

Por ejemplo, si tenemos un personaje que gira el torso, es más fácil calcular nosotros mentalmente (y entender) los puntos para rotar el torso cuando estamos en el espacio-modelo (esto es, el torso en el origen de coordenadas, y nada más), que cuando estamos en el modelo-mundo (esto es por ejemplo, mirando desde una esquina de la habitación y viendo el torso desde un ángulo).

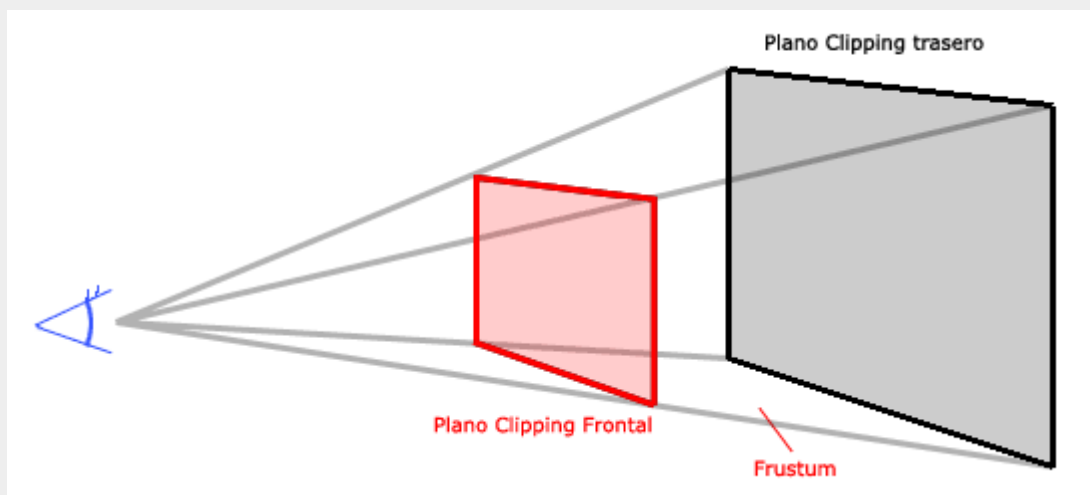


El segundo de los pasos, "Transformar Vista" transforma los objetos desde el espacio-mundo al espacio-vista (o cámara). En este punto podemos decir que tenemos ya una vista 2D mas o menos aproximada de lo que la cámara ve de nuestro mundo en 3D. En esta parte también se aplican cálculos de luz y de "backface culling" (es decir, que caras se muestran y cuales están ocultas).



Por último tenemos "Transformar Proyección", que según sean los campos de vista horizontal y vertical de nuestra cámara crea una adaptación de nuestro modelo-mundo que podamos representar en el monitor "pixel-a-pixel".

Los objetos que están mas cerca a la parte frontal del "frustum" se expanden, y los que están mas lejos se encogen. La adaptación se hace a partir de la geometría del frustum y de la cámara a una forma de 4x4. Esto se es posible a través de una matriz de proyección construida a partir de del campo de visión (FOV, Field of View) o también llamado "Viewing Frustum", un aspect ratio, un plano de corte frontal (o Frontal Clipping Plane), y un plano de corte trasero (o Back Clipping Plane).



Este último proceso se puede asemejar a elegir la lente apropiada a nuestra cámara: según la lente que pongamos veremos el espacio-mundo de una forma u otra.

## Repaso de matemáticas

Imaginemos un avión en un simulador de vuelo... la punta del avión esta en el eje Z positivo, su ala derecha en el eje X positivo, y la parte superior del ala trasera del avión (el timón) apunta al eje Y positivo. De momento tenemos que las coordenadas del objeto corresponden con las coordenadas de nuestro mundo.

Si rotamos el avión 90 grados sobre el eje Y, el morro del avión apuntaría al eje X positivo, y el ala derecha al eje Z negativo. El ala de atrás seguiría mirando hacia arriba, el eje Y positivo. En este nuevo punto rotamos el avión sobre el eje Z. Si nuestras transformaciones son correctas el avión estará rotando sobre su propio eje Z. Si no hemos calculado bien probablemente el avión esté girando sobre el eje Z del mundo.

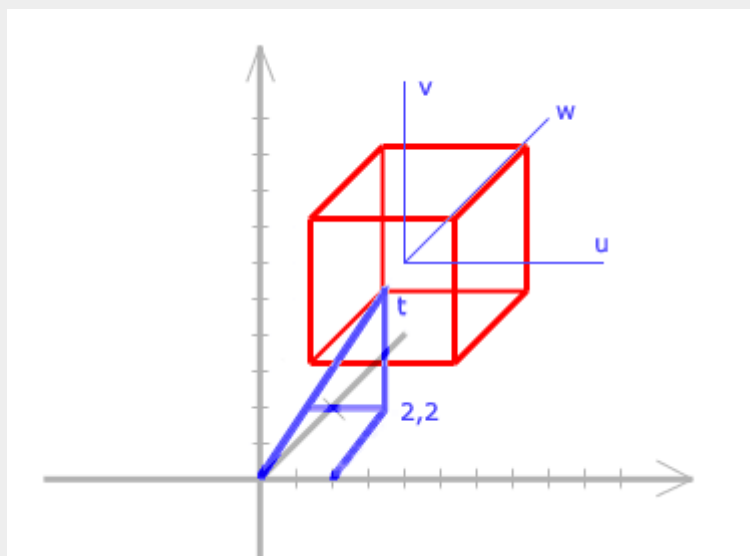
Para no machacarnos la cabeza con este tipo de transformaciones (que empiezan a ser complicadas cuando las coordenadas del modelo no corresponden con las del mundo) debemos usar lo que se llaman Matrices. Si le damos a Direct3D matrices para trabajar, el hará todas las transformaciones del modo correcto).

### Matrices:

Las matrices son arrays de 4x4. Una matriz-mundo contiene cuatro vectores, que representan las coordenadas del espacio-mundo:

```
ux uy uz 0
vx vy vz 0
wx wy wz 0
tx ty tz 1
```

Los vectores u,v y w representan lo que se llama el "cuerpo rígido". Esta es la matriz que define la transformación desde el espacio-modelo al espacio-mundo. Gráficamente quedaría así:



Para describir la posición del cubo, la matriz será esta:



```
1, 0, 0, 0
0, 1, 0, 0
0, 0, 1, 0
2, 2, 2, 1
```

Como el cubo esta orientado en el sistema de coordenadas, la primera columna contiene las coordenadas del espacio-mundo del eje X local, la segunda contiene el eje local Y, y la tercera el eje Z. Los vectores son vectores unitarios, es decir, que su magnitud es 1. La última fila contiene las coordenadas del espacio-mundo del origen del objeto.

En Direct3D se puede acceder a las matrices de este modo:

```
D3DXMATRIX mat;
mat._11 = 1.0f; mat._12 = 0.0f; mat._13 = 0.0f; mat._14 = 0.0f;
mat._21 = 0.0f; mat._22 = 1.0f; mat._23 = 0.0f; mat._24 = 0.0f;
mat._31 = 0.0f; mat._32 = 0.0f; mat._33 = 1.0f; mat._34 = 0.0f;
mat._41 = 0.0f; mat._42 = 0.0f; mat._43 = 0.0f; mat._44 = 1.0f;
```

Este ejemplo mostraba la composición de la matriz identidad.

### La matriz identidad:

```
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

La matriz identidad es una identidad que representa un sistema de coordenadas orientado con las coordenadas del mundo. La coordenada X de la X local es 1; la Z y la Y de la X local son 0, y el vector de origen es (0,0,0). Así que el eje x del modelo local encaja perfectamente con el eje x del mundo. Lo mismo ocurre con los ejes locales X y Z.

Si la posición de un objeto en el espacio-modelo corresponde con su posición en el espacio-mundo, entonces la matriz de transformación del mundo será la matriz identidad.

### La matriz Mundo:

Un ejemplo típico de matriz mundo seria una combinación de matrices de traslación, rotación y escalado de objetos.

### La matriz Vista:

Esta matriz describe la posición y la orientación de la cámara desde la que se ve la escena. Para especificar la vista necesitamos tres vectores: UP, RIGHT y LOOK.

El vector UP nos indica donde es arriba para la cámara. Suponiendo que en estos momentos tú eres la cámara, UP seria un vector que apuntase al techo. El vector RIGHT indicaría a tu derecha. El vector LOOK nos indicaría hacia donde estás mirando.

### Multiplicación de matrices:

Con estos dos dibujos y tus libros de COU (xDD) debería quedar claro:

$$\begin{array}{c} (x1, y1, z1, 1) \\ \text{Vértice} \end{array} \times \begin{array}{c} \left[ \begin{array}{cccc} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{array} \right] \\ \text{Matriz} \end{array} = \begin{array}{c} (x2, y2, z2) \\ \text{Nuevo Vértice} \end{array}$$

Cálculo de la multiplicación:

$$x2 = (x1 \times a) + (y1 \times e) + (z1 \times i) + (1 \times m)$$

$$y2 = (x1 \times b) + (y1 \times f) + (z1 \times j) + (1 \times n)$$

$$z2 = (x1 \times c) + (y1 \times g) + (z1 \times k) + (1 \times o)$$

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \times \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix}$$

Solución de la primera fila:

$$\text{Columna 1} = (1a) + (2e) + (3i) + (4m)$$

$$\text{Columna 2} = (1b) + (2f) + (3j) + (4n)$$

$$\text{Columna 3} = (1c) + (2g) + (3k) + (4o)$$

$$\text{Columna 4} = (1d) + (2h) + (3l) + (4p)$$

Rotar la cámara sobre un eje:

Para visualizarlo imaginémonos nuestro avión F22. Si nos sentáramos en la cabina y pulsamos el pedal de izquierda/derecha los vectores RIGHT y LOOK se deben rotar alrededor del vector UP (rotar sobre el eje Y se llama efecto YAW). Ahora, si estamos volando y nos inclinamos hacia la izquierda o hacia la derecha los vectores RIGHT y UP se deben rotar alrededor de LOOK (rotar sobre el eje Z se llama efecto ROLL). Por último, si inclinamos el avión hacia delante y atrás, estaríamos rotando el vector LOOK y UP sobre RIGHT (rotar sobre el eje X se llama PITCH).

Con nuestra matriz-Vista podemos situar la cámara donde queramos del espacio 3D y rotarla en cualquiera de los ejes para poder hacer nuestro juego.

De un modo gráfico la matriz-Vista quedaría así:

$$\begin{array}{cccc} vx & vy & vz & 0 \\ ux & uy & uz & 0 \\ nx & ny & nz & 0 \\ -(u*c) & -(v*c) & -(n*c) & 1 \end{array}$$

En esta matriz: u, n y v son UP, RIGHT y LOOK. Y c es la posición de la cámara en el mundo. Esta posición se calcula negando el producto entre la posición de la cámara y los vectores u, v y n. Se niegan porque la cámara trabaja es sentido contrario al mundo 3D.

### Otra alternativa: Quaternions

Hasta ahora hemos usado lo que se llaman "vectores libres" de magnitud 1 que tenían 3 componentes (x,y,z). Ahora vamos a usar quaternions (cuaternarios), que tienen 4 componentes (x,y,z,w).

Para representar una rotación de un ángulo G sobre un eje A (Xa Ya Za) el quaternion Q sería:

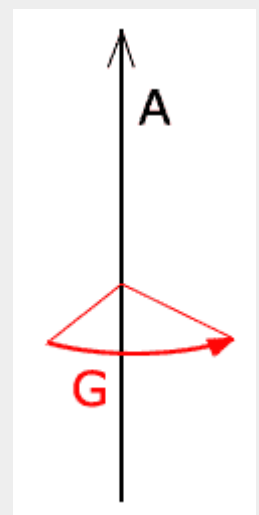
$$q = ( s \ Xa, s \ Ya, s \ Za, c )$$

donde:

$$\begin{aligned} s &= \text{sen} ( G / 2 ) \\ c &= \text{cos} ( G / 2 ) \end{aligned}$$

es decir,

$$q = ( \text{sen} ( G / 2 ) A, \text{cos} ( G / 2 ) )$$



Veámoslo de otro modo: Si nos dan el ángulo y el eje, estos serían los componentes del quaternion:

$$\begin{aligned} x &= Xa \text{ sen} ( G / 2 ) \\ y &= Ya \text{ sen} ( G / 2 ) \\ z &= Za \text{ sen} ( G / 2 ) \\ w &= \text{cos} ( G / 2 ) \end{aligned}$$

Para rotar un vector un ángulo de  $2G$  sobre un eje o usando un quaternion, deberíamos poner las coordenadas de un vector  $p = (px, py, pz, pw)$  en los componentes de un quaternion  $v$ , suponiendo además que tenemos un quaternion  $q$  unitario  $q = (\sin G u, \cos G)$ .

Es decir:

$$v' = q v q^{-1}$$

rota  $v$  sobre el eje  $y$ , un ángulo de  $2G$ . El vector  $v'$  que se obtiene tendrá siempre un valor escalar  $0$  para la componente  $w$ , así que lo puedes omitir para los cálculos. La rotación que hemos hecho aquí es parecida a si nos moviésemos alrededor de una gran mesa de billar.

Para programar toda esta matemática en nuestro programa esta un poco difícil, porque Direct3D (ni otras APIs como OpenGL) no soporta quaternions directamente. La respuesta es:

- Convertir el ángulo a quaterniones
- y convertir de quaterniones a matrices.

Es más sencillo de lo que parece, simplemente hay que seguir estos pasos:

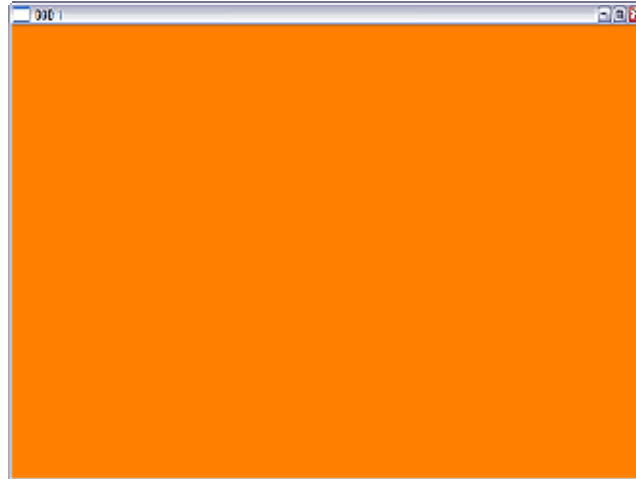
1. Trasladar el vector  $v_{Trans}$  (el que marca la posición de la cámara y el punto sobre el que rota).
2. Construir el quaternion usando los ángulos que nos devuelve  $D3DXVECTOR3$  al rotar la cámara.
3. Construir la matriz a partir del quaternion.
4. Concatenar la matriz con la matriz-posición.
5. Invertir la matriz-posición a la matriz-vista.

### La matriz Proyección:

Tan sólo decir que la matriz de proyección es la que convierte el "frustum" en un espacio cuadrado. Los objetos que están cerca de la cámara se hacen mas grandes, mientras que los otros se hacen mas pequeños.

## Capítulo 1 - Inicialización de D3D

En este ejemplo nos vamos a basar en el programa que construía una ventana y la dejaba lista para usar con D3D de la sección de Win32. Se trata de inicializar D3D, cargarlo en la ventana y hacer que pinte el fondo de ésta de un color sólido. El resultado será como esta ventana:



Por si no lo sabías DirectX 9 se compone de: DirectGraphics (o Direct3D), DirectInput, DirectMusic, DirectPlay, DirectShow y DirectSetup, que son un conjunto de interfaces para manejar toda la multimedia del PC.

Para poder usar estas interfaces deberemos instalar el SDK de DirectX 9 y cargar los ficheros .h y .lib que necesitamos. Para ello deberemos forzar al compilador a que reciba las librerías de los directorios que nos haya creado la instalación de DirectX. Lo haremos a través de "Barra de menú --> Tools --> Options --> Directories...". También podemos probar otro método más seguro que es copiar todos los archivos .h y .lib del SDK a los directorios "include" y "libs" de Visual C++ sobrescribiendo los anteriores que hubiera. Una vez hecho esto empezaremos cargando las librerías que nos proporcionan las funciones de Direct3D:

```
#include <d3d9.h>      //Para las interfaces de Direct3D
#include <d3dx9.h>     // Para las funciones D3DX
```

Realmente no hace falta incluir **d3d9.h** ya que **d3dx9.h** incluye a la anterior. Acto seguido debemos incluir en las opciones de "Link" (Barra de menú --> Project --> Settings...) estas librerías aparte de las que vienen por defecto: **dsound.lib strmiids.lib dxerr9.lib ddraw.lib d3dx9.lib d3dxof.lib d3d9.lib dinput9.lib winmm.lib dxguid.lib**. Una vez hecho esto tendremos el entorno preparado para poder enlazar a los archivos de cabecera que necesitamos utilizar de DirectX.

Lo primero que vamos a hacer es coger nuestro ejemplo de Win32 y añadirle unas cuantas cosas. Lo primero que cambiamos son las declaraciones de funciones. Como se puede observar hemos añadido tres funciones más. También hemos añadido nuevas variables en la declaración de variables globales:

```
//-----
// Declaración de funciones
//-----
LRESULT WINAPI MsgProc(HWND, UINT, WPARAM, LPARAM);
INT WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int);

HRESULT InicializarD3D(HWND hWnd);
void FinalizarD3D(void);
```

```
void Render(void);

//-----
// Declaración de variables globales
//-----
HWND g_hWnd; // Nuestra ventana
LPDIRECT3D9 g_pD3D = NULL; // Objeto D3D usado para crear el dispositivo D3D
LPDIRECT3DDEVICE9 g_pD3DDevice = NULL; // Dispositivo D3D
BOOL g_bWindowed = TRUE; // A pantalla completa o en ventana? (por defecto ventana)
```

La primera de ellas es nuestro objeto D3D que nos servirá para crear nuestro dispositivo 3D. g\_pD3DDevice es el objeto que representa a nuestra tarjeta gráfica (Device=Dispositivo). Este objeto es muy importante ya que a través de él accederemos a nuestra aceleradora y podremos crear transformaciones, dibujar triángulos, texturizar y cualquier otra cosa que sea manipular nuestro mundo 3D virtual. Por último tenemos una variable booleana que indica si nuestra aplicación se ejecutará a pantalla completa o en modo ventana. Por defecto la dejamos en modo ventana pero si cambiamos su valor a FALSE nuestra aplicación se ejecutará a pantalla completa ;)

Vamos a ver la primera de las funciones que hemos añadido. Esta se encarga de crear nuestro dispositivo D3D e inicializarlo:

```
//-----
// Función: InicializarD3D
// Propósito: Inicia el objeto D3D y el dispositivo D3D
//-----
HRESULT InicializarD3D(HWND hWnd)
{
    // En primer lugar creamos el objeto D3D
    if((g_pD3D = Direct3DCreate9(D3D_SDK_VERSION)) == NULL)
        return E_FAIL;

    // Tomamos las propiedades de nuestro escritorio (profundidad de color,
    resolución, etc)
    D3DDISPLAYMODE d3ddm;
    // Especificamos como dispositivo de visualización el dispositivo primario
    if(FAILED(g_pD3D->GetAdapterDisplayMode(D3DADAPTER_DEFAULT, &d3ddm)))
        return E_FAIL;

    // Ahora rellenamos la estructura usada para crear el dispositivo
    // Indicamos que queremos un backbuffer.
    // Ajustamos la anchura y la altura del backbuffer al tamaño de nuestra
    // ventana.
    // Especificamos una aplicación a pantalla completa.
    // Especificamos que D3D escoja el sistema más óptimo para realizar
    // el intercambio del backbuffer a frontbuffer.
    // Especificamos un formato para el backbuffer igual que el de nuestro
    // escritorio.
    D3DPRESENT_PARAMETERS d3dpp;
    ZeroMemory(&d3dpp, sizeof(d3dpp));
    d3dpp.BackBufferCount = 1;
    d3dpp.BackBufferWidth = WND_WIDTH;
    d3dpp.BackBufferHeight = WND_HEIGHT;
    d3dpp.Windowed = g_bWindowed;
    d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
    d3dpp.BackBufferFormat = d3ddm.Format;

    // Por último creamos el dispositivo D3D (la aceleradora 3D propiamente dicha)
```

```
// Escogemos el dispositivo por defecto (el primario)
// Especificamos que queremos un dispositivo HAL, es decir, que tenga
aceleración
// a traves de D3D.
// Especificamos la ventana que se asociará al dispositivo.
// Especificamos que queremos que D3D procese los vértices por software...esto
// se debería comprobar ya que si disponemos de un dispositivo TnL tal como
// GeForce, preferiremos que se realice por hardware.
if(FAILED(g_pD3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
D3DCREATE_SOFTWARE_VERTEXPROCESSING,
&d3dpp, &g_pD3DDev)))
{
    MessageBox(g_hWnd, "No se pudo crear el dispositivo 3D", "Error",
MB_OK|MB_ICONEXCLAMATION);
    return E_FAIL;
}

return S_OK;
}
```

Lo primero que hacemos es crear el objeto D3D con la función **Direct3DCreate9** a la cual se le pasa una macro que indica la versión de D3D. Seguidamente declaramos una variable que es una estructura del tipo **D3DDISPLAYMODE** y la rellenamos con la función **GetAdapterDisplayMode** que es miembro del objeto **g\_pD3D** y a la cual se le pasa como parámetros **D3DADAPTER\_DEFAULT** que quiere decir que recogemos las características de nuestro dispositivo por defecto, es decir, el primario. El segundo parámetro es la estructura la cual se rellenará con las características de dicho dispositivo.

Lo siguiente que hacemos es declarar una variable del tipo **D3DPRESENT\_PARAMETERS** que es una estructura que más adelante se le pasará a la función de creación de dispositivo. De ella hemos rellenado sólo los parámetros que nos interesan. Más adelante utilizaremos otros. Lo primero que hacemos es limpiar la estructura con la función **ZeroMemory**. Seguidamente indicamos que queremos un backbuffer, especificamos las dimensiones de éste, indicamos si queremos una aplicación en modo ventana o a pantalla completa, le decimos que escoja el intercambio del backbuffer al frontbuffer que sea más óptimo para nuestra aceleradora y por último indicamos que nuestro backbuffer tenga el mismo formato (profundidad de color) que nuestro escritorio (esto lo modificaremos mas adelante ya que en un juego real querremos poner el formato que mejor nos convenga y no el que se este usando en el escritorio de Windows).

Para terminar con esta función nos encontramos con la función que crea nuestro dispositivo D3D que es **CreateDevice**, miembro del objeto **g\_pD3D**. A esta función se le pasan 6 parámetros que son:

- Indicamos que D3D coja el dispositivo de visualización primario de Windows.
- Con el segundo parámetro le decimos a D3D que el dispositivo anterior debe ser un dispositivo que soporte una interfaz HAL (Hardware Abstraction Layer), es decir, que sea un dispositivo que soporte aceleración hardware a través de D3D.
- El siguiente parámetro es un handle a nuestra ventana principal.
- El cuarto parámetro le dice a Direct3D que las operaciones con los vértices se realizarán por software cosa que no interesa a los afortunados poseedores de una GeForce1/2.
- El penúltimo parámetro es la estructura que rellenamos anteriormente con las características de nuestro dispositivo de visualización.
- Por último, le pasamos el dispositivo D3D (**g\_pD3DDev**) el cual será inicializado.

Pues si todo lo hemos hecho correctamente ahora deberíamos estar en disposición de poder usar nuestro chip gráfico. Pero como todo lo que se crea debe ser destruido necesitamos una función que "limpie"

todo lo que hemos creado al salir de nuestra aplicación:

```
//-----
// Función: FinalizarD3D
// Propósito: Elimina el dispositivo D3D y el objeto D3D...¡por ese orden!
//-----
void FinalizarD3D(void)
{
    if(g_pD3DDev != NULL)
        g_pD3DDev->Release();

    if(g_pD3D != NULL)
        g_pD3D->Release();
}
```

Como podéis ver, se destruyen los objetos `g_pD3DDev` y `g_pD3D`. Decir que es conveniente que los destruyáis en ese orden ya que `g_pD3DDev` se creo a partir de `g_pD3D` por lo que si no lo hacemos por este orden puede que nos dé un error o no se eliminen bien.

A continuación vamos a describir nuestra función **Render**. Esta es una de las funciones más importantes ya que se encarga de pintar nuestra escena (renderizar). De momento sólo vamos a pintar el fondo de la escena de un color sólido para visualizarlo (naranja):

```
//-----
// Función: Render
// Propósito: Aquí se pondrán nuestras funciones de render
//-----
void Render(void)
{
    if(NULL == g_pD3DDev)
        return;

    // Limpia la pantalla de un color (naranja en este caso)
    g_pD3DDev->Clear(0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(255,127,0), 1.0f, 0);

    // Indica a D3D que en esta sección pondremos nuestras
    // funciones de render
    g_pD3DDev->BeginScene();

    //-----//
    // Funciones de render aquí
    //-----//

    // A partir de aquí ya no podemos poner más funciones
    // de render
    g_pD3DDev->EndScene();

    // Finalmente se pasa el contenido
    // del backbuffer al dispositivo.
    g_pD3DDev->Present(NULL, NULL, g_hWnd, NULL);
}
```

La función comienza comprobando que tenemos un dispositivo D3D. La siguiente función lo que hace es limpiar el buffer de pantalla. Los parámetros de la función **Clear** son:

- Número de áreas que hay en el array de áreas (2º parámetro). Si el array de áreas es `NULL`, este nº debe ser 0.



- Array de áreas a limpiar. En nuestro caso queremos limpiar todo el área de la pantalla por lo que lo ponemos a NULL.
- Flags que indican que tipo de superficie se limpiará. En nuestro caso se limpiará el buffer de pantalla (D3DCLEAR\_TARGET). Otros parámetros aceptados son D3DCLEAR\_ZBUFFER que limpia el ZBuffer (buffer de profundidades que veremos en otro tutorial) y D3DCLEAR\_STENCIL que limpia el buffer de estarcido (Stencil Buffer). Este último buffer se utiliza para crear efectos de reflexión de objetos sobre superficies y para la creación de sombras, etc.
- El siguiente parámetro es el color con el cual limpiaremos nuestra pantalla. En este caso lo haremos con el color naranja. Para especificar el color, podemos hacerlo de varias formas. Un es hacerlo directamente lo cual se haría poniendo el color en hexadecimal (0xFFFF7F00). La otra y más cómoda es utilizar la macro D3DCOLOR\_XRGB a la cual se le pasa como parámetros los valores R (red), G (green) y B (blue). lo de la X significa que se obvia la componente alpha y en ese caso se pone a 255 que significa opacidad total. Por lo tanto nuestro naranja quedaría D3DCOLOR\_XRGB(255,127,0).
- El penúltimo parámetro es el valor a poner en el ZBuffer al limpiarlo. Va de 0.0 (posición más cercana) a 1.0 (posición más alejada).
- Por último tenemos el valor con el cual se rellenará el buffer de estarcido al limpiarlo.

La función **BeginScene** miembro del objeto g\_pD3DDev indica a D3D que a partir de ese momento comenzarán las funciones de render. Después de esta función podemos poner todas nuestras funciones que dibujen triángulos, etc. Evidentemente, al igual que le decimos a D3D cuándo empezamos a renderizar, también le debemos decir cuándo dejamos de utilizar funciones de render. Para ello se utiliza la función **EndScene** que también es miembro de g\_pD3DDev.

Por último nos encontramos con la función **Present** que pasa el contenido de nuestro BackBuffer al FrontBuffer, es decir, a la pantalla. Sus dos primeros parámetros y el último deben ser NULL y el tercer parámetro es un handle a la ventana donde realizamos todas nuestras operaciones de render.

Sólo queda decir que estas tres nuevas funciones se llaman desde **WinMain**. Aquí tenemos el código del **WinMain** modificado:

```
//-----
// Función: WinMain
// Propósito; Es la función principal de toda aplicación de Windows
// Se encarga de crear la ventana, mostrarla y es la receptora de los
// mensajes que le son enviados a la aplicación
//-----
INT WINAPI WinMain(HINSTANCE hInstance, // Instancia a nuestra aplicación
HINSTANCE hPrevInstance, // Existen instancias previas de nuestra
aplicación?
LPSTR lpCmdLine, // Línea de comandos
int nShowCmd) // Cómo se despliega nuestra ventana?
{

    if(!SUCCEEDED(IniVentana(hInstance)))
        return 0;

    if(SUCCEEDED(InicializarD3D(g_hWnd))
    {
        // Mostrar nuestra ventana
        ShowWindow(g_hWnd, nShowCmd);
        UpdateWindow(g_hWnd);
    }
}
```

```
// Si está en modo ventana mostramos el cursor
if(g_bWindowed)
    ShowCursor(TRUE);
else // de lo contrario lo ocultamos
    ShowCursor(FALSE);

// Bucle de mensajes típico de Windows
MSG msg; // handle a un mensaje
ZeroMemory(&msg, sizeof(msg));

while(msg.message != WM_QUIT)
{
    // si se pulsa ESC salimos
    if(GET_KEYDOWN(VK_ESCAPE))
        PostQuitMessage(0);

    if(PeekMessage(&msg, NULL, 0U, 0U, PM_REMOVE))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    else
    {
        Render();
    }
}

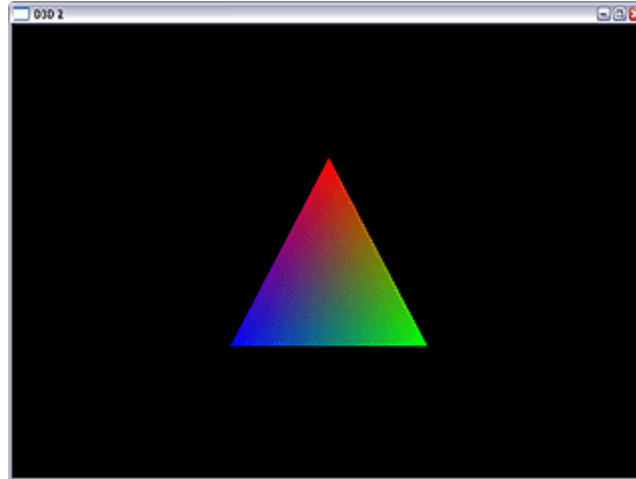
// Salir de la aplicación
FinalizarD3D();
UnregisterClass(APP_NAME,hInstance);

return (0);
}
```

Pues ya está, si has hecho todo bien te debería salir un ventana como la de arriba ;)

## Capítulo 2 - El polígono más simple

En este ejemplo vamos a continuar sobre la base del anterior para, una vez finalizada toda la inicialización, mostrar un triángulo con Direct3D y que esté gestionado por nuestro hardware. Vamos a aprender lo que es un búfer de vértices (Vertex Buffer) y vamos a ver como utilizarlo para definir grupos de vértices en nuestra aplicación. El resultado final será como el de esta ventana:



En primer lugar vemos que tenemos una nueva definición en el código. Este es nuestro tipo de vértices. Con ésta línea lo único que hacemos es definir un modelo para nuestros vértices que tenga coordenadas x, y, z y rhw (rhw indica que queremos coordenadas de pantalla y no tridimensionales, así en 800x600 el eje X ira de 0 a 799 y el eje Y de 0 599 cuando pongamos rhw. Esto se llama coordenadas transformadas) y un color para cada vértice.

```
// Coordenadas XYZ RHW y el color del vértice
#define D3DFVF_MIVERTICE (D3DFVF_XYZRHW | D3DFVF_DIFFUSE)
```

Ahora que hemos definido un tipo de vértice, debemos crear una estructura con esos datos que definan nuestro vértice:

```
// Nuestro tipo de vértice
struct MIVERTICE
{
    float x, y, z, rhw;    // posición de un vértice transformado
    DWORD color;           // color del vértice
};
```

Esta estructura contiene justo los elementos que necesitamos para definir nuestro vértice particular. Después de esto, hemos declarado una nueva función que se encargará de crear y rellenar nuestro buffer de vértices y nuestro nuevo objeto del tipo LPDIRECT3DVERTEXBUFFER9 definido en las declaraciones de variables globales.

Aquí esta dicha función:

```
//-----
// Función: IniBufferVert
// Propósito: Inicia el buffer de vértices
//-----
HRESULT IniBufferVert(void)
{
    // asignamos vértices que forman un triángulo con un color para cada vértice
    MIVERTICE vertices[] =
    {
```

```

// x, y, z, rhw, color del vértice
{ 320.0f, 140.0f, 0.5f, 1.0f, D3DCOLOR_XRGB(255, 0, 0), },
{ 420.0f, 340.0f, 0.5f, 1.0f, D3DCOLOR_XRGB(0, 255, 0), },
{ 220.0f, 340.0f, 0.5f, 1.0f, D3DCOLOR_XRGB(0, 0, 255), },
};

// Creamos el buffer de vértices.
// Primero especificamos el tamaño...3 vértices * el tamaño de la estructura
// que hemos definido.
// Seguidamente especificamos el modo de uso.
// El siguiente argumento es el tipo de vértice.
// Seguidamente especificamos que queremos que D3D elija la memoria donde
// se guardarán los vértices (memoria de video, agp, memoria de sistema...)
// Finalmente le pasamos el buffer de vértices al cual se le asignará una
// zona en memoria.
if(FAILED(g_pD3DDev->CreateVertexBuffer(3 * sizeof(MIVERTICE), 0,
D3DFVF_MIVERTICE, D3DPOOL_DEFAULT, &g_pVB, NULL)))
{
    return E_FAIL;
}

VOID* pVertices;

// Siempre que queramos añadir vértices a nuestro buffer, debemos bloquear el
// buffer.
if(FAILED(g_pVB->Lock(0, sizeof(vertices), &pVertices, 0)))
    return E_FAIL;

// copiamos los vértices en los que hemos especificado el triángulo
// a la zona de memoria a la que apunta nuestro buffer de vértices.
memcpy(pVertices, vertices, sizeof(vertices));

// ¡Nunca olvidar desbloquear el buffer cuando se halla terminado de
rellenarlo!
g_pVB->Unlock();

return S_OK;
}

```

Empezamos declarando una variable 'vertices' del tipo MIVERTICE que es la estructura que hemos creado anteriormente y que define nuestro vértice. Al mismo tiempo que la declaramos la rellenamos con 3 vértices, cada uno con su coordenada x, y, z y rhw y un color para cada vértice que en este caso es por este orden: rojo, verde y azul. Estos tres vértices van a definir nuestro triángulo y sus coordenadas han sido puestas así para que lo formen. Después de esto nos disponemos a crear nuestro vertex buffer. Para ello utilizamos la función **CreateVertexBuffer** miembro de g\_pD3DDev. Esta función tiene los siguientes argumentos:

- Tamaño de los datos que contendrá nuestro buffer. En este caso, al tener 3 vértices será 3 veces el tamaño de la estructura.
- Características de nuestro vertex buffer...lo dejamos por defecto (en todo caso consultar la ayuda del SDK).
- Tipo de vértice que vamos a utilizar. En nuestro caso usaremos el que hemos definido, es decir, D3DFVF\_MIVERTICE.
- Tipo de memoria en la que se almacenarán los vértices. Puede ser en la memoria de video, en el AGP, en memoria de sistema... Dejamos que D3D elija.
- Vertex buffer para que D3D le asigne una zona de memoria y lo inicialice.

- Por último le pasamos NULL.

Después de esto declaramos una variable puntero de tipo VOID, pVertices, la cual dentro de poco apuntará a la zona de memoria donde se encuentra nuestro buffer de vértices ubicado. Bien, pues ahora viene el momento de rellenar nuestro vertex buffer. Para ello usamos el par de funciones **Lock/Unlock**. Comenzamos bloqueando el buffer con **Lock** a la cual se le pasan los siguientes parámetros:

- Desplazamiento dentro del buffer que se bloqueará (en bytes). En nuestro caso se bloqueará desde el inicio.
- Cantidad de bytes del buffer a bloquear.
- Puntero a un array de tipo VOID que apuntará al inicio de la zona de memoria donde se almacenarán nuestros vértices.
- Tipo de acceso a nuestro buffer. Puede ser un acceso de solo lectura (lo cual hace el acceso más rápido), o de sólo escritura si al crear nuestro buffer así lo especificamos. Para más información ir a la guía de referencia del SDK.

Una vez bloqueado el buffer, podemos escribir en él. En nuestro caso, para mayor rapidez copiamos a través de memset el contenido de 'vertices' a 'pVertices', es decir, a la zona de memoria donde se ubicó nuestro vertex buffer.

Finalmente desbloqueamos el buffer con **Unlock** ¡y ya está!.

Por supuesto, lo mismo que hemos creado el vertex buffer, debemos destruirlo en **FinalizarD3D**):

```
if(g_pVB != NULL)
    g_pVB->Release();
```

Ya tenemos listo nuestro buffer de vértices por lo que ya podemos dibujar nuestro triángulo. Esta es la función Render modificada:

```
//-----
// Función: Render
// Propósito: Aquí se pondrán nuestras funciones de render
//-----
void Render(void)
{
    if(NULL == g_pD3DDev)
        return;

    // Limpia la pantalla de un color (negro en este caso)
    g_pD3DDev->Clear(0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0,0,0), 1.0f, 0);

    // Indica a D3D que en esta sección pondremos nuestras
    // funciones de render
    g_pD3DDev->BeginScene();

    //-----//

    // Especificamos cual es la fuente de la cual sacaremos la geometría
    g_pD3DDev->SetStreamSource(0, g_pVB, 0, sizeof(MIVERTICE));
    // Seleccionamos un tipo de vértice...en este caso el que hemos
    // creado nosotros
    g_pD3DDev->SetFVF(D3DFVF_MIVERTICE);
    // Dibujamos los vértices..en este caso los dibujamos de forma que se
    // interpreten como listas de triángulos, le decimos que comience en
```

```

// el vértice nº 0 y que tan sólo vamos a dibujar una primitiva del
// tipo triángulo.
g_pD3DDev->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 1);

//-----//

// A partir de aquí ya no podemos poner más funciones
// de render
g_pD3DDev->EndScene();

// Finalmente se pasa el contenido
// del backbuffer al dispositivo.
g_pD3DDev->Present(NULL, NULL, g_hWnd, NULL);
}

```

Aquí lo único que hemos hecho ha sido añadir el código que está entre las funciones **BeginScene** y **EndScene**. La función **SetStreamSource** sirve para especificar la fuente donde se encuentran nuestro vértices que definen la geometría a dibujar. De esta función lo que nos interesa es el segundo parámetro que es un puntero a nuestro vertex buffer y el cuarto que es el tamaño de la estructura de nuestro vértice. Seguido a esto (el orden no es prioritario) tenemos que especificamos el tipo de vértice que utilizaremos, es decir, D3DFVF\_MIVERTICE. Para ello utilizamos la función **SetFVF** que no creo que necesite mucha explicación. Finalmente ya estamos en disposición de dibujar nuestro triángulo con la función **DrawPrimitive** con los parámetros D3DPT\_TRIANGLELIST que quiere decir que D3D debe interpretar la cadena de vértices como una lista en la cual cada 3 vértices definen un triángulo, el vértice inicial y el número de primitivas a dibujar. En nuestro caso comenzamos en el vértice cero y dibujamos un único triángulo. Más adelante veremos otro tipo de primitivas como por ejemplo D3DPT\_TRIANGLESTRIP que define una tira (strip) de triángulos.

Ya sólo queda modificar el código de **WinMain** para que cargue nuestro búfer de vértices:

```

//-----
// Función: WinMain
// Propósito; Es la función principal de toda aplicación de Windows
// Se encarga de crear la ventana, mostrarla y es la receptora de los
// mensajes que le son enviados a la aplicación
//-----
INT WINAPI WinMain(HINSTANCE hInstance, // Instancia a nuestra aplicación
HINSTANCE hPrevInstance, // Existen instancias previas de nuestra
aplicación?
LPSTR lpCmdLine, // Línea de comandos
int nShowCmd) // Cómo se despliega nuestra ventana?
{

    if(!SUCCEEDED(IniVentana(hInstance)))
        return 0;

    if(SUCCEEDED(InicializarD3D(g_hWnd))
    {
        if(SUCCEEDED(IniBufferVert()))
        {
            // Mostrar nuestra ventana
            ShowWindow(g_hWnd, nShowCmd);
            UpdateWindow(g_hWnd);

            // Si está en modo ventana mostramos el cursor
            if(g_bWindowed)
                ShowCursor(TRUE);
        }
    }
}

```

```
else // de lo contrario lo ocultamos
    ShowCursor(FALSE);

// Bucle de mensajes típico de Windows
MSG msg; // handle a un mensaje
ZeroMemory(&msg, sizeof(msg));

while(msg.message != WM_QUIT)
{
    // si se pulsa ESC salimos
    if(GET_KEYDOWN(VK_ESCAPE))
        PostQuitMessage(0);

    if(PeekMessage(&msg, NULL, 0U, 0U, PM_REMOVE))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    else
    {
        Render();
    }
}

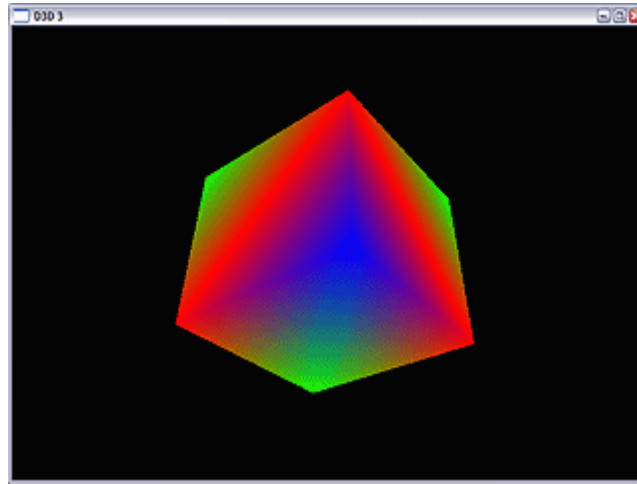
// Salir de la aplicación
FinalizarD3D();
UnregisterClass(APP_NAME,hInstance);

return (0);
}
```

Pues ya está, si has hecho todo bien te debería salir un triangulo coloreado como el de arriba ;)

## Capítulo 3 - Las matrices

En este ejemplo vamos a modificar el de la página anterior añadiéndole unos vértices más para formar un cubo y mediante transformaciones por matrices lograr que rote sobre sí mismo. El resultado final será como muestra esta ventana:

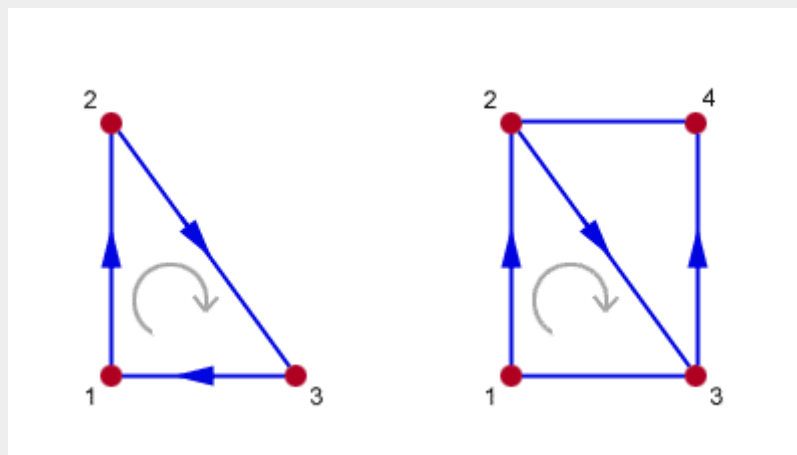


En primer lugar vemos que en la definición de nuestro tipo de vértice y en su estructura hemos quitado las coordenadas *r h y w* ya que ésta vez daremos las coordenadas en formato espacial (no en coordenadas de pantalla como hacíamos antes).

```
// Coordenadas XYZ RHW y el color del vértice
#define D3DFVF_MIVERTICE (D3DFVF_XYZ | D3DFVF_DIFFUSE)

// Nuestro tipo de vértice
struct MIVERTICE
{
    float x, y, z, rhw;    // posición de un vértice transformado
    DWORD color;           // color del vértice
};
```

Luego hemos retocado la función que inicializa el buffer de vértices para que ahora cree un cubo. La hemos cambiado el nombre y ahora se llama **IniGeometria**. Hay que fijarse en los nuevos vértices que hemos puesto y comprobar mediante papel que efectivamente forman un cubo. Es más, tienen que estar en un orden y nosotros hemos elegido el sentido de las agujas del reloj (Clock Wise - CW).



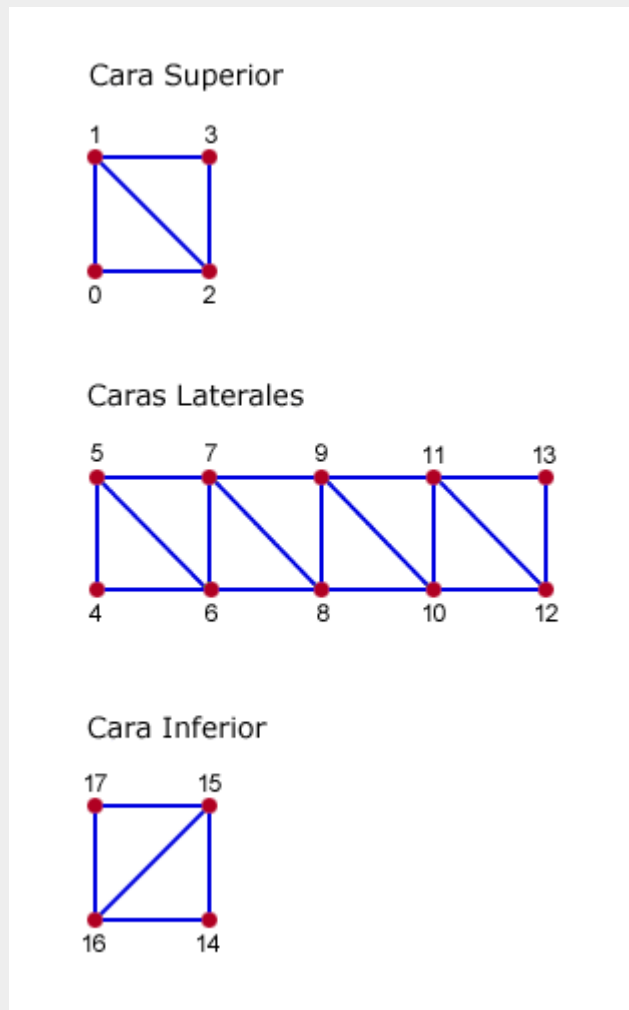
El cubo está compuesto por 3 Triangle Strips. Los cuatro primeros vértices componen el primer strip con



2 triángulos para la cara superior.

Las caras laterales es el segundo strip con 10 vértices y un total de 8 triángulos (2 por cada cara).

El tercer strip es la cara inferior que esta formada de la misma manera que la superior.



Aquí esta la nueva función **IniGeometria**:

```
//-----
// Función: IniGeometria
// Propósito: Inicia el buffer de vértices
//-----
HRESULT IniGeometria(void)
{
    // asignamos vértices que forman un cubo con un color para cada vértice
    // hay que asegurarse que los vertices se definen en el sentido contrario
    // a las agujas del reloj
    MIVERTICE vertices[] =
    {
        //Cara superior
        {-5.0f, 5.0f, -5.0f, D3DCOLOR_XRGB(0, 0, 255)}, //Vértice 0 - Azul
        {-5.0f, 5.0f, 5.0f, D3DCOLOR_XRGB(255, 0, 0)}, //Vértice 1 - Rojo
        {5.0f, 5.0f, -5.0f, D3DCOLOR_XRGB(255, 0, 0)}, //Vértice 2 - Rojo
        {5.0f, 5.0f, 5.0f, D3DCOLOR_XRGB(0, 255, 0)}, //Vértice 3 - Verde

        //Cara 1
        {-5.0f, -5.0f, -5.0f, D3DCOLOR_XRGB(255, 0, 0)}, //Vértice 4 - Rojo
```

```

        {-5.0f, 5.0f, -5.0f, D3DCOLOR_XRGB(0, 0, 255)}, //Vértice 5 - Azul
        {5.0f, -5.0f, -5.0f, D3DCOLOR_XRGB(0, 255, 0)}, //Vértice 6 - Verde
        {5.0f, 5.0f, -5.0f, D3DCOLOR_XRGB(255, 0, 0)}, //Vértice 7 - Rojo

        //Cara 2
        {5.0f, -5.0f, 5.0f, D3DCOLOR_XRGB(0, 0, 255)}, //Vértice 8 - Azul
        {5.0f, 5.0f, 5.0f, D3DCOLOR_XRGB(0, 255, 0)}, //Vértice 9 - Verde

        //Cara 3
        {-5.0f, -5.0f, 5.0f, D3DCOLOR_XRGB(0, 255, 0)}, //Vértice 10 - Verde
        {-5.0f, 5.0f, 5.0f, D3DCOLOR_XRGB(255, 0, 0)}, //Vértice 11 - Rojo

        //Cara 4
        {-5.0f, -5.0f, -5.0f, D3DCOLOR_XRGB(255, 0, 0)}, //Vértice 12 - Rojo
        {-5.0f, 5.0f, -5.0f, D3DCOLOR_XRGB(0, 0, 255)}, //Vértice 13 - Azul

        //Cara inferior
        {5.0f, -5.0f, -5.0f, D3DCOLOR_XRGB(0, 255, 0)}, //Vértice 14 - Verde
        {5.0f, -5.0f, 5.0f, D3DCOLOR_XRGB(0, 0, 255)}, //Vértice 15 - Azul
        {-5.0f, -5.0f, -5.0f, D3DCOLOR_XRGB(255, 0, 0)}, //Vértice 16 - Rojo
        {-5.0f, -5.0f, 5.0f, D3DCOLOR_XRGB(0, 255, 0)}, //Vértice 17 - Verde
    };

    // Creamos el buffer de vértices.
    // Primero especificamos el tamaño...18 vértices * el tamaño de la estructura
    // que hemos definido.
    // Seguidamente especificamos el modo de uso.
    // El siguiente argumento es el tipo de vértice.
    // Seguidamente especificamos que queremos que D3D elija la memoria donde
    // se guardarán los vértices (memoria de video, agp, memoria de sistema...)
    // Finalmente le pasamos el buffer de vértices al cual se le asignará una
    // zona en memoria.
    if(FAILED(g_pD3DDev->CreateVertexBuffer(18 * sizeof(MIVERTICE), 0,
D3DFVF_MIVERTICE, D3DPOOL_DEFAULT, &g_pVB, NULL)))
    {
        return E_FAIL;
    }

    VOID* pVertices;

    // Siempre que queramos añadir vértices a nuestro buffer, debemos bloquear el
    // buffer.
    if(FAILED(g_pVB->Lock(0, sizeof(vertices), &pVertices, 0)))
        return E_FAIL;

    // copiamos los vértices en los que hemos especificado el cubo
    // a la zona de memoria a la que apunta nuestro buffer de vértices.
    memcpy(pVertices, vertices, sizeof(vertices));

    // ¡Nunca olvidar desbloquear el buffer cuando se halla terminado de
    rellenarlo!
    g_pVB->Unlock();

    return S_OK;
}

```

Ya tenemos listo nuestro cubo, ahora sólo queda renderizarlo para que podamos verlo. Además vamos a hacer que rote sobre sus ejes. Para ello llamaremos a la función **ActulizaMatrices** justo antes de renderizar nuestro cubo:

```
//-----
```

```
// Función: Render
// Propósito: Aquí se pondrán nuestras funciones de render
//-----
void Render(void)
{
    if(NULL == g_pD3DDev)
        return;

    // Limpia la pantalla de un color (negro en este caso)
    g_pD3DDev->Clear(0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0,0,0), 1.0f, 0);

    // Indica a D3D que en esta sección pondremos nuestras
    // funciones de render
    g_pD3DDev->BeginScene();

    //-----//

    ActualizaMatrices();

    // Especificamos cual es la fuente de la cual sacaremos la geometría
    g_pD3DDev->SetStreamSource(0, g_pVB, 0, sizeof(MIVERTICE));
    // Seleccionamos un tipo de vértice...en este caso el que hemos
    // creado nosotros
    g_pD3DDev->SetFVF(D3DFVF_MIVERTICE);
    // Dibujamos los vértices..en este caso los dibujamos de forma que se
    // interpreten como listas de triángulos, le decimos que comience en
    // el vértice nº 0 y que tan sólo vamos a dibujar una primitiva del
    // tipo triángulo.
    g_pD3DDev->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 2);
    g_pD3DDev->DrawPrimitive(D3DPT_TRIANGLELIST, 4, 8);
    g_pD3DDev->DrawPrimitive(D3DPT_TRIANGLELIST, 12, 2);

    //-----//

    // A partir de aquí ya no podemos poner más funciones
    // de render
    g_pD3DDev->EndScene();

    // Finalmente se pasa el contenido
    // del backbuffer al dispositivo.
    g_pD3DDev->Present(NULL, NULL, g_hWnd, NULL);
}
```

Con esto podemos observar como dibujamos los 3 triangle strips en la función render.

A continuación vamos a ver la función **ActualizaMatrices**:

```
//-----
// Función: ActualizaMatrices()
// Propósito: Especifica las matrices de mundo, de vista y de proyección
//-----
void ActualizaMatrices(void)
{
    // especificamos matrices de rotación en el eje X, Y, y Z para nuestra
    // matriz de mundo. Esto hará que el cubo rote alrededor
    // de los ejes del mundo 3D.
    D3DXMATRIX matWorld, matWorldX, matWorldY, matWorldZ;

    //creamos las matrices de transformación
```

```

D3DXMatrixRotationX(&matWorldX, timeGetTime()/400.0f);
D3DXMatrixRotationY(&matWorldY, timeGetTime()/400.0f);
D3DXMatrixRotationZ(&matWorldZ, timeGetTime()/400.0f);

//Las combinamos multiplicandolas unas con otras
D3DXMatrixMultiply(&matWorld, &matWorldX, &matWorldY);
D3DXMatrixMultiply(&matWorld, &matWorld, &matWorldZ);

//Aplicamos la transformacion a la matriz de mundo
g_pD3DDev->SetTransform(D3DTS_WORLD, &matWorld);

//Aquí crearemos la cámara
//La cámara tiene tres paramateros: "Posicion", "Donde Mira" and "Direccion
Arriba"
//Hemos puesto lo siguiente:
//Posicion: (0, 0, -30)
//Donde Mira: (0, 0, 0)
//Direccion Arriba: Y-Axis.
D3DXMATRIX matView;
D3DXMatrixLookAtLH(&matView, &D3DXVECTOR3(0.0f, 0.0f, -30.0f), //Posicion
&D3DXVECTOR3(0.0f, 0.0f, 0.0f), //Donde Mira
&D3DXVECTOR3(0.0f, 1.0f, 0.0f)); //Direccion
Arriba
g_pD3DDev->SetTransform(D3DTS_VIEW, &matView);

// Ahora creamos una matriz de proyección la cual proyectará nuestros objetos
// 3D en un plano (2D). Una matriz de proyección se puede definir a través de
// un FOV (field of view - campo de visión) el cual es un ángulo que
// especificaremos en radianes. En este caso el FOV es de 45°, por un aspect
ratio
// o aspecto que indica la relación entre el ancho y el alto de la ventana (en
las
// pantallas normales tenemos una relacion 4:3 asique el ratio será 1.33333),
un plano
// cercano que especifica en qué punto cercano al observador se cortará la
escena, es
// decir, no se dibujará y finalmente un plano lejano que indica en qué punto
se dejará
// de dibujar la escena más allá del observador.
D3DXMATRIX matProj;
D3DXMatrixPerspectiveFovLH(&matProj,
D3DX_PI/4, WND_WIDTH/(float)WND_HEIGHT, 1.0f, 500.0f);
g_pD3DDev->SetTransform(D3DTS_PROJECTION, &matProj);
}

```

Antes de comenzar a explicar las líneas del código, vamos a repasar ciertos conceptos que son importantes para entender esta función bien.

Pensemos en un triángulo. Nosotros definimos el triángulo especificando sus vértices con respecto al origen de coordenadas que en nuestro caso era el centro de la pantalla (más bien habría que decir que es con respecto al centro del triángulo que para más comodidad se hace coincidir con el origen de coordenadas). A esto se le denominan coordenadas del objeto. Ahora imaginémonos que queremos mover ese triángulo a otra posición en un mundo 3D. Se podría pensar que esto se hace modificando los valores de las coordenadas de sus vértices y ciertamente esto se puede hacer así pero consumiría demasiado tiempo de proceso. En lugar de eso, debemos definir una matriz de transformación (en este caso de translación) la cual será multiplicada por todos los vértices de nuestro triángulo y hará que nuestro triángulo se mueva a la nueva posición. Pues bien, a estas coordenadas transformadas de nuestro triángulo se les denominan coordenadas de mundo. Ya tenemos nuestro triángulo situado en el lugar que

queremos.

Pero ahora nos encontramos con otro problema...resulta que nuestra pantalla es un plano 2D y no puede representar directamente nuestro triángulo ya que éste tiene una tercera dimensión. ¿Qué pasa entonces?... pues que tendremos que utilizar otro tipo de transformación que es la proyección. Hay varios tipos de proyección pero las más usadas son la proyección *ortográfica* o *isométrica* y la proyección *cónica*. En cualquiera de los dos casos de proyección, su misión es la de proyectar en un plano bidimensional nuestro objeto 3D de forma que esté preparado para ser plasmado en nuestra pantalla. En realidad este plano en el cual se proyecta nuestro objeto 3D es el plano posterior del frustum. El frustum es una figura geométrica la cual es una pirámide cuya cúspide ha sido recortada. Digamos que en el lugar donde se encontraba la cúspide es donde se sitúa la cámara, en definitiva, el punto de vista desde el que observamos la escena. Pues bien, según cómo definamos nuestro frustum, así se proyectará nuestro objeto 3D. Si nuestro frustum está definido para una proyección ortogonal tenemos que las dimensiones del objeto siempre son las mismas. Por ejemplo, si tenemos dos objetos con las mismas dimensiones, aunque uno esté muy cerca de la cámara y el otro muy lejos, veremos ambos objetos con el mismo tamaño. Sin embargo la proyección cónica respeta la perspectiva de los objetos.

Por último tenemos que para especificar un punto de vista desde el que ver nuestra escena debemos definir una nueva matriz que es la matriz de vista. Con ella especificaremos las coordenadas en las que se encuentra situada nuestra cámara y un vector director que es el que indica hacia dónde mira dicha cámara.

Bien, ahora vamos al código. La primera línea declara una variable del tipo `D3DXMATRIX` que representan matrices en D3D. Ahora debemos rellenar estas matrices de forma que contengan los valores necesarios para que se conviertan en matrices de rotación en el eje X, Y y Z respectivamente. Esto lo hacemos con la función **D3DXMatrixRotationY**. Esta función toma los siguientes parámetros:

- El primero es un puntero a la matriz que será rellenada. Como hemos dicho, ha de ser del tipo `D3DXMATRIX`.
- El segundo y último parámetro es el ángulo de rotación.

Una nota importante es que el ángulo en todas las funciones que lo requieran se especifica en **RADIANES** y **NO** en **GRADOS**.

En nuestro caso se especifica el ángulo de forma que nuestro triángulo dé un giro ( $360^\circ = 2 * \text{PI}$ ) sobre cada eje por cada 10 segundos. Ya que **timeGetTime** te devuelve el tiempo en milisegundos y multiplicado por 0.0001 lo pasamos a 10 segundos y `D3DX_PI` es una constante que vale 3.141592....

Después de crear las matrices tenemos que multiplicarlas para aunarlas todas en una (`matWorld`). Esto lo podemos hacer con la función **D3DXMatrixMultiply** que toma los siguientes parámetros:

- El primero es un puntero a la matriz que será rellenada. Como hemos dicho, ha de ser del tipo `D3DXMATRIX`.
- El segundo es un puntero a la primera matriz que será multiplicada.
- El tercero es un puntero a la segunda matriz que será multiplicada.

```
D3DXMATRIX matWorld, matWorldX, matWorldY, matWorldZ;
```

```
//creamos las matrices de transformación
```

```
D3DXMatrixRotationX(&matWorldX, 2*D3DX_PI*(timeGetTime()*0.0001f));
D3DXMatrixRotationY(&matWorldY, 2*D3DX_PI*(timeGetTime()*0.0001f));
D3DXMatrixRotationZ(&matWorldZ, 2*D3DX_PI*(timeGetTime()*0.0001f));

//Las combinamos multiplicandolas unas con otras
D3DXMatrixMultiply(&matWorld, &matWorldX, &matWorldY);
D3DXMatrixMultiply(&matWorld, &matWorld, &matWorldZ);
```

Ya tenemos la matriz final de rotación en matWorld Pero con crear la matriz de rotación no es suficiente. Si ahora mismo ejecutásemos el programa, nuestro triángulo no rotaría, ya que no hemos especificado nuestra matriz de rotación como la nueva matriz de mundo. Esto se hace con la función **SetTransform** miembro del objeto de dispositivo g\_pD3DDev. Esta función toma los siguientes valores:

- El primero de ellos especifica que matriz de las que mantiene D3D se va a modificar. Este parámetro puede ser:
  - D3DTS\_VIEW que la veremos en unos instantes y que lo que hace es mover nuestro mundo para que sea visto desde el punto que nosotros especifiquemos.
  - D3DTS\_WORLD que es el parámetro que hemos puesto en este caso y que lo que hace es que multiplica todos los vértices de los objetos que se rendericen a continuación de esta función por la matriz de mundo que acabamos de especificar.
  - D3DTS\_PROJECTION que especifica la matriz de proyección y que también veremos en poco tiempo.
  - D3DTS\_TEXTURE0....D3DTS\_TEXTURE7 que realiza transformaciones con las coordenadas de textura.
- El segundo parámetro es un puntero a la matriz que contiene la información adecuada para que surtan los efectos deseados.

```
//Aplicamos la transformacion a la matriz de mundo
g_pD3DDev->SetTransform(D3DTS_WORLD, &matWorld);
```

Lo siguiente que hemos hecho es especificar una matriz de vista. Con esta matriz de vista lo que conseguiremos es poner la cámara a través de la cual vemos nuestro mundo 3D en la posición y dirección adecuadas.

Creamos una matriz de vista con la función **D3DXMatrixLookAtLH**. Esta función crea una matriz de vista basado en el sistema de coordenadas cartesiano de la mano izquierda (LH = left handed). El otro modo es el de la mano derecha (RH = right handed). Direct3D se basa en el sistema de coordenadas cartesiano de la mano izquierda mientras que OpenGL se basa en el de la mano derecha. Esto significa que en D3D la Z positiva va hacia dentro del monitor y en OGL al revés. Los parámetros de la función que crea la matriz de vista son:

- Un puntero a la matriz la cual será rellenada con los datos pertinentes.
- Un puntero a un vector del tipo D3DXVECTOR3 que indicará la posición de nuestra cámara.
- Un puntero a un vector que indicará hacia donde mira la cámara, es decir, el vector director o dirección.
- Un puntero a un vector que indica hacia dónde está arriba.

El primer y segundo parámetro no ofrecen dudas. Tal vez sí el tercero y el cuarto. El vector "Up" (arriba) es el resultado del producto vectorial entre el vector "Right" (derecha) y el vector "LookAt" (mirar hacia). El vector "Right" es el resultado del producto vectorial entre los vectores "Up" y "LookAt". Pero todo esto se verá con detalle en el tema de la cámara en movimiento. Tan sólo aclarar

una cosa sobre el vector "LookAt" que es el cuarto parámetro de la función anterior. Este vector, como he dicho indica hacia donde mirará la cámara. En nuestro caso hemos especificado que mire hacia el origen de coordenadas que es donde se encuentra nuestro cubo. Si hubiésemos querido que nuestra cámara mirase hacia la izquierda (cosa que poco sentido tiene en estos momentos ya que lo único que veríamos sería una oscuridad total), deberíamos haber especificado un vector tal como (-1.0f, 0.0f, 0.0f). Para terminar con esto decir que tanto el vector "Up" como el vector "LookAt" deben estar normalizados, es decir, cualquiera de sus componentes debe ser 1.

Después de crear la matriz debemos llamar a la función **SetTransform** con los parámetros adecuados.

```
//Aquí crearemos la cámara
//La cámara tiene tres paramateros: "Posicion", "Donde Mira" y "Direccion
Arriba"
//Hemos puesto lo siguiente:
//Posicion:      (0, 0, -30) 30 unidades hacia atrás en el eje Z
//Donde Mira:    (0, 0, 0)   Origen
//Direccion Arriba: (0, 1, 0)   Eje-Y.
D3DXMATRIX matView;
D3DXMatrixLookAtLH(&matView, &D3DXVECTOR3(0.0f, 0.0f, -30.0f), //Posicion
                  &D3DXVECTOR3(0.0f, 0.0f, 0.0f), //Donde Mira
                  &D3DXVECTOR3(0.0f, 1.0f, 0.0f)); //Direccion
Arriba
g_pD3DDev->SetTransform(D3DTS_VIEW, &matView);
```

Lo último de esta función es crear una matriz de transformación con la función **D3DXMatrixPerspectiveFovLH**. Como en la función de creación de la matriz de vista, aquí también nos encontramos que la proyección se ve condicionada por el sistema de coordenadas (mano izquierda ó mano derecha). En nuestro caso creamos una matriz de proyección en perspectiva de mano izquierda. Los parámetros de esta función son:

- Un puntero a la matriz que se rellenará con los datos necesarios para convertirse en una matriz de proyección.
- FOV (field of view = campo de visión). El FOV es una ángulo que como tal se especifica en radianes.
- Aspect Ratio que es la relación entre el ancho y el alto de nuestra ventana.
- Near Plane (plano cercano). Es el plano que hay más pegado a la cámara y todo objeto que quede por detrás de él no será visible.
- Far Plane (plano lejano). Es el plano más alejado de la cámara y todo objeto que esté más allá de él no será visible

Hay que decir que tanto el "near plane" como el "far plane" forman parte del frustum, esa figura geométrica de la que hablamos al principio. El frustum está formado por 6 planos y todo aquello que esté totalmente fuera de él no será visible. El problema de todo esto es que no será visible pero seguirá siendo calculado cosa que debería evitarse mediante algún tipo de algoritmo, ya sean árboles BSP, Octrees, Portales... etc.

Finalizamos llamando a la función **SetTransform** con los parámetros adecuados.

```
// Ahora creamos una matriz de proyección la cual proyectará nuestros objetos
// 3D en un plano (2D). Una matriz de proyección se puede definir a través de
// un FOV (field of view - campo de visión) el cual es un ángulo que
// especificaremos en radianes. En este caso el FOV es de 45°, por un aspect
ratio
// o aspecto que indica la relación entre el ancho y el alto de la ventana (en
las
```



```
// pantallas normales tenemos una relacion 4:3 asi que el ratio será 1.33333),
un plano
// cercano que especifica en qué punto cercano al observador se cortará la
escena, es
// decir, no se dibujará y finalmente un plano lejano que indica en qué punto
se dejará
// de dibujar la escena más allá del observador.
D3DXMATRIX matProj;
D3DXMatrixPerspectiveFovLH(&matProj,D3DX_PI/4, WND_WIDTH/(float)WND_HEIGHT,
1.0f, 500.0f);
g_pD3DDev->SetTransform(D3DTS_PROJECTION, &matProj);
```

Por último nos queda añadir que al final de la función InicializarD3D hemos añadido las siguientes líneas

```
//Activa el back face culling. Lo hacemos porque no queremos que se pinte las
//caras de los poligonos que no se ven
g_pD3DDev->SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);

//Desactivamos la luz porque la estemos especificando en el color de vértice
g_pD3DDev->SetRenderState(D3DRS_LIGHTING, FALSE);
```

La primera activa el *Back Face Culling*. El culling es la técnica de ocultación de caras posteriores. Hay 3 opciones que son: **D3DCULL\_CW**, **D3DCULL\_CCW** y **D3DCULL\_NONE**. Con la primera opción le decimos a D3D que las caras de nuestros objetos que queremos ocultar están formadas por triángulos cuyos vértices han sido especificados en el sentido de las agujas del reloj (CW = clock wise). Con el segundo método es todo lo contrario, los vértices que queremos ocultar se han especificado en el sentido contrario al de las agujas del reloj (CCW = counter-clockwise) y por último, **D3DCULL\_NONE** le dice a D3D que haga caso omiso de ese orden y renderice ambas caras del triángulo. Todo esto es muy importante porque si por ejemplo creamos un objeto con triángulos creados en sentido CW y especificamos a D3D un culling tipo CW, resulta que no vamos a ver nuestro objeto, ya que las caras que están siendo renderizadas son las caras posteriores a las que esperábamos ver. En un objeto convexo serían las que dan al interior del objeto (si entráramos en el interior del objeto, sí que veríamos las caras). Sólo cuando le dijésemos a D3D que debe tomar los triángulos a ocultar como CCW se renderizarían bien todos los triángulos que forman nuestro objeto. Ni que decir tiene que es muy conveniente que nunca desactivemos el culling por razones de optimización... aunque habrá momentos en los que sí tengamos que hacerlo, cuando por ejemplo tengamos una rejilla o una valla ya que debe poder verse por los dos lados.

La segunda desactiva la iluminación. Esto es porque no queremos que D3D calcule por nosotros el color difuso (diffuse color). Si hubiésemos activado la iluminación, la componente difusa la hubiese proporcionado la fuente de luz y con esta información D3D hubiese, a través de cálculos de interpolación del color entre vértices y degradación, aplicado el color adecuado acorde con la iluminación que nosotros especificásemos en nuestra escena. De todas maneras, aunque ahora activemos la iluminación, nuestro cubo no se vería por dos razones. La primera y más obvia es que no hemos especificado ninguna fuente de luz y la segunda y no tan obvia para los no iniciados es que no hemos especificado ninguna normal a nuestros vértices. Una normal es un vector perpendicular a una superficie. ¿Y cómo hallamos este vector normal?...pues necesitamos dos vectores y calcular el producto vectorial entre ambos. Precisamente esta operación devuelve un vector que es perpendicular a estos dos vectores (que al fin y al cabo forman un plano). Pero esto se verá en el tutorial sobre iluminación. Por último queda decir que hemos desactivado la iluminación para que nuestro triángulo se renderice con los colores que nosotros hemos especificado en la componente difusa de los vértices que lo forman.

La función que utilizan ambas líneas es **SetRenderState()** que sirve para especificar ciertas



características que D3D tendrá que tener en cuenta para realizar la rasterización. Sus parámetros son:

- La propiedad que será configurada. (Consultar la ayuda del SDK para ver todos los valores que puede tomar este argumento)
- El valor que se le dará a la propiedad anterior. (Consultar la ayuda del SDK para ver todos los valores que puede tomar este argumento)

Pues ya está, si has hecho todo bien te debería salir un cubo rotando sobre los ejes y coloreado como el de arriba ;)

## Capítulo 4 - Las Texturas

A partir de aquí vamos a cambiar bastante el estilo de codificación ya que vamos a utilizar clases para todo, es decir C++. En este ejemplo en concreto vamos a crear un clase **CJuego** que va a contener las funciones que antes teníamos globales que hagan referencia al juego en cuestión, como son **InicilizarD3D** y **ActulizarMatrices...** Dejaremos otras globales como las teníamos: **IniVentana**, **WinMain** y **MsgProc**. Para nuestro cubo crearemos un clase **CCubo** que contendrá funciones como **Inicilizar**, **Renderizar...** Y por último crearemos una pequeña clase **CUtil** con algunas funciones de utilidad.

El resultado es este:



Ya que hemos introducido un gran cambio adoptando la metodología C++ es deber tuyo repasar las nuevas clases del código fuente que acabo de comentar y ver que realmente queda todo más estructurado. No voy a explicar qué hace cada clase porque eso deberías saberlo con tan sólo echarlas un vistazo ya que lo único que hemos hecho ha sido tomar el ejemplo anterior y reorganizarlo en clases.

Lo que se va a tratar en este tutorial es el tema de las texturas, porque como verás ahora nuestro cubo las soporta, y alguna que otra modificación que hemos hecho en el código de inicialización de D3D y otras puntualizaciones...

Pues bien, vamos allá:

Lo primero que vamos a reseñar es que en la clase **CUtil** hemos creado unas funciones que nos permiten manejar un logbook, esto es, un archivo en el que almacenamos lo que ocurre mientras se ejecuta nuestro programa. Es muy útil para, en caso de error, saber dónde se ha cometido éste y qué función lo ha provocado. Nuestro archivo lo hemos llamado **log.txt**

El código de estas funciones es simple, creamos un archivo nuevo y en caso de que existiera borraríamos su contenido, con **ActivarLog**. Y con la función **EscribirLog** podemos escribir ya sobre el archivo. Si en algún momento del programa queremos desactivar dicho log utilizamos **DesactivarLog**.

```
//-----
// Función: DesactivarLog
// Propósito: Desactiva el Logbook
//-----
void CUtil::DesactivarLog(void)
{
    LogActivado = FALSE;
}
```

```
//-----
// Función: ActivarLog
// Propósito: Activa el logbook creando un archivo nuevo
//-----
void CUtil::ActivarLog(void)
{
    LogActivado = TRUE;

    FILE* pFile;

    //Elimina el contenido del archivo
    pFile = fopen("log.txt", "wb");

    //Cerrarlo
    fclose(pFile);
}

//-----
// Función: EscribirLog
// Propósito: Escribe en el logbook
//-----
void CUtil::EscribirLog(char *lpszText, ...)
{
    if(LogActivado)
    {
        va_list argList;
        FILE *pFile;

        //Inicializa la lista variable de argumentos
        va_start(argList, lpszText);

        //Abre el archivo para añadir
        pFile = fopen("log.txt", "a+");

        //Escribe el texto y mete una nueva línea
        vfprintf(pFile, lpszText, argList);
        putc('\n', pFile);

        //Cerramos el archivo
        fclose(pFile);
        va_end(argList);

        EscribirConsola(" Escribiendo en log.txt . . .\n");
    }
}
```

También hay que reseñar que hemos añadido una consola de depuración donde podemos mostrar mensajes en tiempo de ejecución. Sus funciones son: **ActivarConsola**, **DesactivarConsola** y **EscribirConsola**. Estos tipos de ventana tipo consola se explican en el tutorial 3 de Win32.

Vamos a ver ahora nuestro nuevo **WinMain**:

```
//-----
// Función: WinMain
// Propósito: Es la función principal de toda aplicación de Windows
// Se encarga de crear la ventana, mostrarla y es la receptora de los
// mensajes que le son enviados a la aplicación
//-----
INT WINAPI WinMain(HINSTANCE hInstance, // Instancia a nuestra aplicación
```

```

HINSTANCE hPrevInstance,           // Existen instancias previas de nuestra
aplicación?
LPSTR lpCmdLine,                   // Línea de comandos
int nShowCmd)                       // Cómo se despliega nuestra ventana?
{

    char hora[128];
    char fecha[128];
    struct tm *today;
    time_t ltime;

    if(!SUCCEEDED(IniVentana(hInstance)))
        return 0;

    if(g_bDebug)
        ActivarConsola();

    EscribirConsola("!!!IMPORTANTE NO CERRAR ESTA VENTANA!!!\n\n");

    Util= new CUtil(g_hWnd,debugInputHandle,debugOutputHandle);
    Juego= new CJuego(g_hWnd,debugInputHandle,debugOutputHandle);

    _strtime(hora);
    _strdate(fecha);
    today=localtime(&ltime);

    Util->ActivarLog();
    Util->EscribirLog("***** %s Logbook ***** %s - %s *****\n",
APP_NAME, hora, fecha);

    Util->EscribirLog("-> Objeto g_hWnd registrado. Ventana Creada.\n");

    if(g_bConsola)
        Util->EscribirLog("-> Consola activada.\n");

    if(g_bWindowed)
        Util->EscribirLog("-> Procediendo a incializar D3D en modo
ventana...\n");
    else
        Util->EscribirLog("-> Procediendo a incializar D3D en pantalla
completa...\n");

    if(SUCCEEDED(Juego->InicializarD3D(g_bWindowed)))
    {
        Util->EscribirLog("-> Direct3D Inicializado.\n");

        // Si está en modo ventana mostramos el cursor
        if(g_bWindowed)
        {
            ShowCursor(TRUE);
            Util->EscribirLog("-> Cursor activado.\n");
        }
        else // de lo contrario lo ocultamos
        {
            ShowCursor(FALSE);
            Util->EscribirLog("-> Cursor desactivado.\n");
        }

        // Mostrar nuestra ventana
        ShowWindow(g_hWnd, nShowCmd);
        UpdateWindow(g_hWnd);
    }
}

```

```

Util->EscribirLog("-> Ventana mostrada.\n");

if (SUCCEEDED(Juego->InicializarJuego()))
{
    // Bucle de mensajes típico de Windows
    MSG msg; // handle a un mensaje
    ZeroMemory(&msg, sizeof(msg));

    Util->EscribirLog("
*****\n");
    Util->EscribirLog("\t-> Ejecutando la aplicación . . .");

    while(msg.message != WM_QUIT)
    {
        // si se pulsa ESC salimos
        if(GET_KEYDOWN(VK_ESCAPE))
            PostQuitMessage(0);

        if(PeekMessage(&msg, NULL, 0U, 0U, PM_REMOVE))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        else
        {
            Render();
        }
    }
}

// Salir de la aplicación
if(g_bConsola)
{
    DesactivarConsola();
    Util->EscribirLog("-> Consola desactivada.\n");
}

BorrarObj(Juego);

UnregisterClass(APP_NAME,hInstance);

Util->EscribirLog("-> Objeto g_hWnd eliminado. Ventana Finalizada.\n");

Util->EscribirLog("-> Aplicación Terminada.\n");

_strtime(hora);
_strdate(fecha);

Util->EscribirLog("***** %s Fin Logbook ***** %s - %s ****\n",
APP_NAME, hora, fecha);

BorrarObj(Util);

return (0);
}

```

Vemos mucho código, pero la mayor parte es para escribir en el log. Lo único que realmente ha cambiado son las llamadas a las funciones **InicializarD3D** e **InicializarJuego** miembros de la clase

**CJuego.** También podemos ver unas funciones para manejar el tiempo (`_strtime`, `_strdate`) que utilizamos para guardar la fecha y hora en el log, al principio y al final.

En la inicilización de D3D también han cambiado unas cuantas cosas:

```
//-----
// Función: InicializarD3D
// Propósito: Inicia el objeto D3D y el dispositivo D3D
//-----
HRESULT CJuego::InicializarD3D(BOOL bWindowed)
{
    // En primer lugar creamos el objeto D3D
    if((g_pD3D = Direct3DCreate8(D3D_SDK_VERSION)) == NULL)
    {
        Util->EscribirLog("\t*- Error al crear el objeto D3D.");
        return E_FAIL;
    }

    Util->EscribirLog("\t-> Objeto D3D inicializado (m_pD3D).");

    D3DDISPLAYMODE d3ddm;
    // Especificamos como dispositivo de visualización el dispositivo primario

    d3ddm.Format=SelModoVideo(WND_WIDTH,WND_HEIGHT,32);
    if(d3ddm.Format != D3DFMT_UNKNOWN)
    {
        //Ancho x Alto x 32bit seleccionado
        d3ddm.Width = WND_WIDTH;
        d3ddm.Height = WND_HEIGHT;

        Util->EscribirLog("\t-> Modo %dx%d x 32bit seleccionado. Formato =
%d.", WND_WIDTH, WND_HEIGHT, d3ddm.Format);
    }
    else
    {
        d3ddm.Format=SelModoVideo(WND_WIDTH,WND_HEIGHT,16);
        if(d3ddm.Format != D3DFMT_UNKNOWN)
        {
            //Ancho x Alto x 16bit seleccionado
            d3ddm.Width = WND_WIDTH;
            d3ddm.Height = WND_HEIGHT;

            Util->EscribirLog("\t-> Modo %dx%d x 16bit seleccionado. Formato =
%d.", WND_WIDTH, WND_HEIGHT, d3ddm.Format);
        }
        else
        {
            Util->EscribirLog("\t*- Imposible seleccionar el modo de video para
%d x %d", WND_WIDTH, WND_HEIGHT);
            return E_FAIL;
        }
    }
}

// Ahora rellenamos la estructura usada para crear el dispositivo
// Indicamos que queremos un backbuffer.
// Ajustamos la anchura y la altura del backbuffer al tamaño de nuestra
// ventana.
// Especificamos una aplicación a pantalla completa.
// Especificamos que D3D escoja el sistema más óptimo para realizar
// el intercambio del backbuffer a frontbuffer.
```

```
// Especificamos un formato para el backbuffer igual que el de nuestro
// escritorio.
D3DPRESENT_PARAMETERS d3dpp;
ZeroMemory(&d3dpp, sizeof(d3dpp));
d3dpp.BackBufferCount = 1;
d3dpp.BackBufferWidth = WND_WIDTH;
d3dpp.BackBufferHeight = WND_HEIGHT;
d3dpp.hDeviceWindow = hWnd;
d3dpp.Windowed = g_bWindowed;
d3dpp.BackBufferFormat = d3ddm.Format;
if (!bWindowed)
{
    d3dpp.FullScreen_RefreshRateInHz = D3DPRESENT_RATE_DEFAULT;
    d3dpp.PresentationInterval = D3DPRESENT_INTERVAL_IMMEDIATE;
}
else
{
    d3dpp.PresentationInterval = 0;
    d3dpp.FullScreen_RefreshRateInHz = 0;
}

//Seleccionamos el mejor buffer de profundidad, 32, 24 o 16 bit

if(m_pd3D->CheckDeviceFormat(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL,
d3ddm.Format, D3DUSAGE_DEPTHSTENCIL, D3DRTYPE_SURFACE, D3DFMT_D32)==D3D_OK)
{
    d3dpp.AutoDepthStencilFormat = D3DFMT_D32;
    d3dpp.EnableAutoDepthStencil = TRUE;

    Util->EscribirLog("\t-> Seleccionado 32bit depth buffer.");
}
else
    if(m_pd3D->CheckDeviceFormat(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL,
d3ddm.Format, D3DUSAGE_DEPTHSTENCIL, D3DRTYPE_SURFACE, D3DFMT_D24X8)==D3D_OK)
    {
        d3dpp.AutoDepthStencilFormat = D3DFMT_D24X8;
        d3dpp.EnableAutoDepthStencil = TRUE;

        Util->EscribirLog("\t-> Seleccionado 24bit depth buffer.");
    }
    else
        if(m_pd3D->CheckDeviceFormat(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL,
d3ddm.Format, D3DUSAGE_DEPTHSTENCIL, D3DRTYPE_SURFACE, D3DFMT_D16)==D3D_OK)
        {
            d3dpp.AutoDepthStencilFormat = D3DFMT_D16;
            d3dpp.EnableAutoDepthStencil = TRUE;

            Util->EscribirLog("\t-> Seleccionado 16bit depth buffer.");
        }
        else
        {
            d3dpp.EnableAutoDepthStencil = FALSE;
            Util->EscribirLog("\t*- Imposible seleccionar un depth
buffer.");
        }

// Por último creamos el dispositivo D3D (la aceleradora 3D propiamente dicha)
// Escogemos el dispositivo por defecto (el primario)
// Especificamos que queremos un dispositivo HAL, es decir, que tenga
aceleración
// a traves de D3D.
```

```

// Especificamos la ventana que se asociará al dispositivo.
// Especificamos que queremos que D3D procese los vértices por software...esto
// se debería comprobar ya que si disponemos de un dispositivo TnL tal como
// GeForce, preferiremos que se realice por hardware.
if(FAILED(g_pD3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
D3DCREATE_SOFTWARE_VERTEXPROCESSING,
&d3dpp, &g_pD3DDev)))
{
    Util->EscribirLog("\t*- No se pudo crear el dispositivo D3D.\n");
    return E_FAIL;
}

Util->EscribirLog("\t-> Dispositivo D3D creado (m_pD3DDev).\n");

//Activa el back face culling. Lo hacemos porque no queremos que se pinte las
//caras de los poligonos que no se ven
g_pD3DDev->SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);

//Desactivamos la luz porque la estemos especificando en el color de vértice
g_pD3DDev->SetRenderState(D3DRS_LIGHTING, FALSE);

return S_OK;
}

```

Aquí han cambiado varias cosas. En primer lugar mediante la función **SelModoVideo** seleccionamos el mejor formato posible para el modo de video dado y con todo ese código condicional seleccionamos en definitiva el mejor modo de video posible a la resolución dada.

El siguiente cambio es en la estructura de los parámetros. Hemos añadido 3 parámetros más para el modo de pantalla completa que lo único que consiguen es suavizar el movimiento. Para más información sobre estos parámetros consulta el SDK.

El último cambio significativo es que hemos añadido un buffer de profundidad (Depth Buffer) de tipo Stencil (o estarcido) y hemos seleccionado el que tenga más profundidad (16, 24 o 32 bits) si lo puede soportar nuestro adaptador de video.

Con la línea **d3dpp.EnableAutoDepthStencil = TRUE** lo que hacemos es que D3D administre el buffer de profundidad (depth buffer) y el buffer de estarcido (stencil buffer) por nosotros. Si este parámetro está activado, entonces también debemos darle un valor correcto al parámetro **AutoDepthStencilFormat**. Por eso vamos comprobando de mayor a menor la resolución de buffer de profundidad hasta que encontremos el mayor posible y si no desactivarlo.

Sólo nos queda un pequeño apunte antes de pasar con las texturas, que es la librería **principal.h**

A esta librería la llamaremos desde todas las clases que utilicemos. En ella hacemos dos definiciones muy importantes que nos van a servir para borrar punteros a objetos y liberar interfaces de una forma segura. Aquí las tenemos:

```

//-----
// principal.h
//-----

#ifndef AFX_PRINCIPAL_H
#define AFX_PRINCIPAL_H

#include <stdlib.h>
#include <stdio.h>
#include <windows.h>
#include <time.h>

```



```
//-----
// Definiciones
//-----

#define WIN32_LEAN_AND_MEAN
#define APP_NAME "D3D 4"

// Elimina objetos e interfaces de memoria
#define BorrarInter(pInterface) if(pInterface != NULL) {pInterface->Release();
pInterface=NULL;}
#define BorrarObj(pObject) if(pObject != NULL) {delete pObject; pObject=NULL;}

// Nos dice si una tecla determinada ha sido pulsada
#define GET_KEYDOWN(key) ((int)((GetAsyncKeyState(key) & (1 << 15)) >> 15))

#endif //AFX_PRINCIPAL_H
```

Dicho esto ya podemos ver el código para añadir la textura a nuestro cubo. Lo primero es ver que hemos añadido una nueva variable a la clase **CCubo** del tipo **LPDIRECT3DTEXTURE9**, que va a ser nuestra textura en sí:

```
LPDIRECT3DTEXTURE9 m_pTextura;
```

Ahora observemos que hemos cambiado la estructura del tipo de vértice y la definición del tipo de vértice para nuestro cubo. Ahora en la estructura hemos añadido dos nuevas variables (tu y tv) de tipo float que sirven para mapear la textura y en la definición le hemos puesto un nuevo parámetro que le indica que el vértice va a usar una textura:

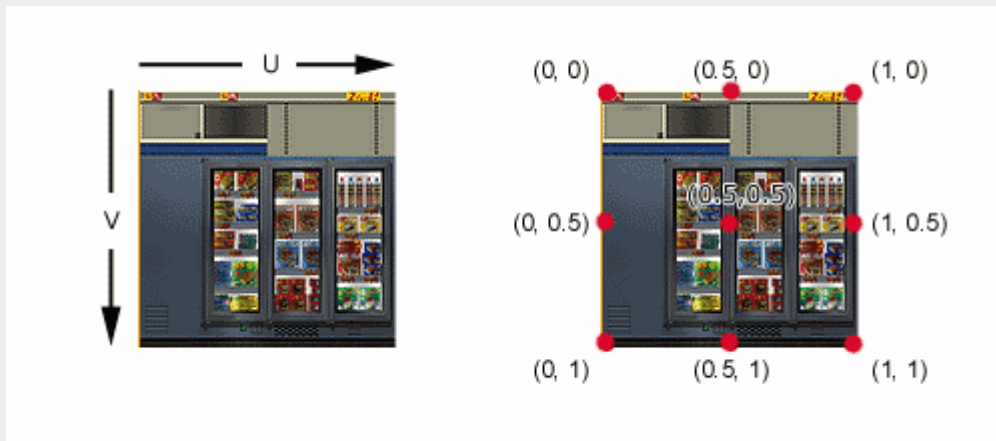
```
#define D3DFVF_VERTCUBO (D3DFVF_XYZ|D3DFVF_DIFFUSE|D3DFVF_TEX1)

struct VERTCUBO
{
    FLOAT x, y, z;
    DWORD color;
    FLOAT tu, tv;
};
```

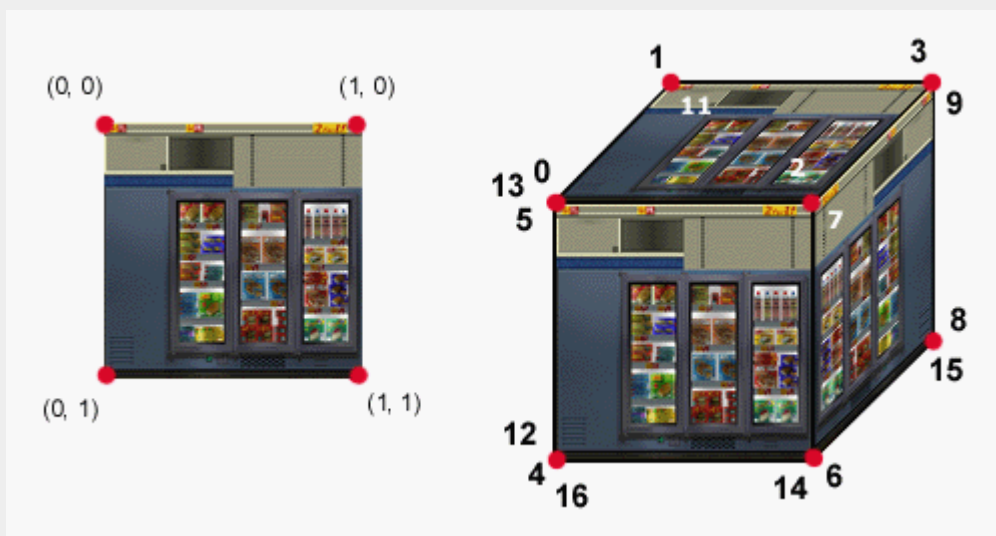
Como podemos intuir, ahora en la inicialización de los vértices (en la función **ActualizarVertices** miembro de **CCubo**) deberemos incluir los nuevos datos que añadimos a la estructura del vértice, osea, las coordenadas de textura. Estas coordenadas se asignan directamente a los vértices. De esta manera podemos controlar que porción de textura es mapeada en la primitiva. Por ejemplo si ponemos en el vértice de la esquina superior izquierda de un cuadrado la posición tu, tv (0.0f, 0.0f) y en el vértice inferior derecho ponemos (1.0f, 1.0f) la textura se mapeará entera sobre el cuadrado. También tendremos que indicar en el vértice superior derecho (1.0f, 0.0f) y en el vértice inferior izquierdo (0.0f, 1.0f). En cambio si en vez de poner 1.0f pusieramos 0.5f se mapearía la mitad. 0.0 indica cominezo de la textura, ya sea por arriba de ésta o por su izquierda y 1.0 indica su fin. Si pusieramos un valor superior a 1 la textura se repetiría al mapearse en nuestra primitiva.

Supongamos que tenemos una textura (textura.bmp) que esta dividida en su interior en 4 cuadrados del mismo tamaño que contienen un dibujo distinto cada uno, y que nosotros queremos coger para texturizar nuestra primitiva justo el cuadrado que se encuentra en la parte inferior derecha de la textura. Pues tendríamos que poner en el vértice de la esquina superior izquierda de nuestra primitiva (0.5f, 0.5f) porque empezamos en mitad de la textura, tanto X como Y (en texturas se llaman U y V de ahí tu y tv). En la esquina inferior derecha (1.0f, 1.0f) ya que terminamos al final de la textura. En la superior

derecha (1.0f, 0.5f) porque en U terminamos con la textura pero en V se encuentra en la mitad. Y en la inferior izquierda (0.5f, 1.0f) justo al revés que el anterior.



Una vez mapeada la textura la función **Textura** miembro de **CCubo** se encarga de cargar nuestra textura de un archivo y asociarla a la variable que creamos antes (m\_pTextura).



```
//-----
// Función: Textura
// Propósito: Carga la textura de un archivo
//-----
HRESULT CCubo::Textura(const char *szArchivo)
{
    if(FAILED(D3DXCreateTextureFromFile(D3DDisp, szArchivo, &m_pTextura)))
    {
        return E_FAIL;
    }

    return S_OK;
}
```

Esta función miembro utiliza la función **D3DXCreateTextureFromFile** cuyos parámetros son:

- Un objeto de dispositivo LPDIRECT3DDEVICE8.
- La ruta de la textura a cargar (esta función soporta los formatos BMP, TGA y JPEG).

- Un puntero a nuestra textura.

Tampoco hay que olvidar que al finalizar el objeto de tipo **CCubo** (en el destructor) hay que liberar el espacio de memoria utilizado por la textura y el buffer de vértices:

```
//-----
// Función: Destructor
//-----
CCubo::~CCubo()
{
    BorrarInter(m_pTextura);
    BorrarInter(BufferVert);
}
```

Sólo nos queda ver el código que se encarga de mostrar la textura al renderizar el cubo y que se encuentra en la propia función **Renderizar** de **CCubo**:

```
//-----
// Función: Renderizar
// Propósito: Renderiza el cubo
//-----
BOOL CCubo::Renderizar()
{
    D3DDisp->SetStreamSource(0, BufferVert, sizeof(VERTCUBO));
    D3DDisp->SetVertexShader(D3DFVF_VERTCUBO);

    if(m_pTextura)
    {
        //Hay textura. No queremeos que la textura se sombree segun los colores
        //de los vertices y para ello ponemos D3DTOP_SELECTARG1
        D3DDisp->SetTexture(0, m_pTextura);
        D3DDisp->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_SELECTARG1);
    }
    else
    {
        //No hay textura. Desactivamos el renderizado con textura.
        D3DDisp->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_DISABLE);
    }

    D3DDisp->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2); //Arriba
    D3DDisp->DrawPrimitive(D3DPT_TRIANGLESTRIP, 4, 8); //Lados
    D3DDisp->DrawPrimitive(D3DPT_TRIANGLESTRIP, 14, 2); //Abajo

    return TRUE;
}
```

Como podemos ver, si la textura ha sido inicializada se va a cargar para renderizar con **SetTexture** que toma como argumentos:

- El número de stage al cual se le asignará dicha textura.
- La textura.

En Direct3D hay un máximo de 8 stages. Los stages son zonas en las que se almacenan unas determinadas texturas (con sus propiedades) para posteriormente realizar algún tipo de operación entre ellas. Normalmente siempre utilizaremos el stage 0 pero más adelante, cuando tratemos la técnica single-pass multitexturing, utilizaremos también el resto de stages.

En la siguiente línea se utiliza la función **SetTextureStageState** para especificar que queremos como

color en la primitiva el primer argumento (nuestra textura en este caso).

Por último, se especifican a través de la función **DrawPrimitive** los triangle strip que componen nuestro cubo para que se rendericen.

Sólo nos queda puntualizar que al final de la función **InicializarJuego** miembro de **CJuego** activamos el filtrado bilinear mediante la función **SetSamplerState** para que nuestra textura se vea difuminada y no se pixelicen mucho los texels:

```
//Ponemos el filtrado bilinear de texturas  
m_pD3DDev->SetSamplerState(0,D3DSAMP_MINFILTER, D3DTEXF_LINEAR);  
m_pD3DDev->SetSamplerState(0,D3DSAMP_MAGFILTER, D3DTEXF_LINEAR);
```

Pues ya está, ya tenemos un cubo rotando y con textura como el de arriba ;)

## Capítulo 5 - La Cámara

En este tutorial hemos añadido muchas cosas al código. A partir de aquí tendremos un clase que controla la entrada (teclado y ratón) llamada **CInput**, otra clase a modo de cronómetro llamada **CCrono**, otra que maneja nuestra fuente para escribir en pantalla llamada **CFuente** y la más importante de este tutorial, **CCamara** que controla nuestra cámara.



Pues bien, el código de la clase **CInput** se explica en el tutorial sobre DirectInput, el código de la clase **CCrono** no tiene más misterio que unas cuantas funciones para calcular el tiempo que lleva en ejecución y los FPS (mejor si le echas un vistazo al código antes de empezar) y el código de la clase **CFuente** a estas alturas sería de fácil comprensión, tan sólo decir que no utiliza buffer de vértices y emplea 2 triángulos mediante **TRIANGLE STRIP** para cada letra, que toma de una textura, y que los efectos de *alpha blending* que usa se explicarán en el siguiente tutorial. Una vez dicho esto y ojeado previamente el código nos podemos meter con nuestra cámara.

Una persona cuando piensa en una cámara piensa en un objeto a través del cual podemos ver el mundo que nos rodea desde un punto de vista determinado y con un campo de visión específico. Pues bien, básicamente en D3D (y cualquier API 3D) la cámara es eso, es decir, una "herramienta" que nos permite posicionarnos en la situación más ventajosa de nuestro mundo 3D. Pero la similitud entre una cámara "real" y una cámara "virtual" se queda ahí... Por ejemplo, podemos pensar que para que una cámara virtual gire a la derecha 90° tan sólo hay que rotar la misma 90° a la derecha al igual que una cámara real... Pues nada más lejos de la realidad. Para que nuestra cámara en D3D rote 90° a la derecha lo que tendremos que hacer es rotar todo nuestro mundo 3D 90° a la izquierda... Bien pues lo mismo se puede decir del resto de rotaciones y también para las traslaciones. Por ejemplo imaginemos que queremos trasladar nuestra cámara hacia adelante en el eje Z 10 unidades... pues bien lo que realmente se hace es trasladar todo el mundo 3D -10 unidades en el eje Z.

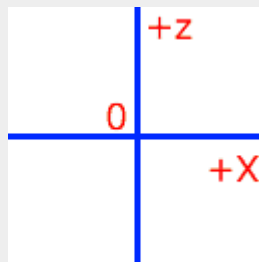
Entonces, ¿cómo hacer todo esto de rotar/trasladar nuestro mundo al contrario de la cámara?. Bien podríamos decir que nuestra cámara en realidad siempre se encuentra en el origen de coordenadas y que todo debe rotarse y trasladarse en relación a ella. Evidentemente las rotaciones y traslaciones se consiguen como ya sabemos, es decir, con matrices. Pero hay una cosa que es importante tener en cuenta. La concatenación (multiplicación) de matrices no goza de la propiedad conmutativa, es decir, que no es lo mismo multiplicar la matriz A por la matriz B que multiplicar la matriz B por la matriz A. Es por esto que para crear nuestra matriz de vista (la matriz que posiciona el punto de vista adecuado o mejor la matriz que mueve todo nuestro mundo para posicionarlo de forma que se visiona desde el punto de vista especificado) debemos seguir el siguiente orden:

$$\text{ViewMatrix} = \text{TranslationMatrix} * \text{RotationYMatrix} * \text{RotationXMatrix} * \text{RotationZMatrix}$$

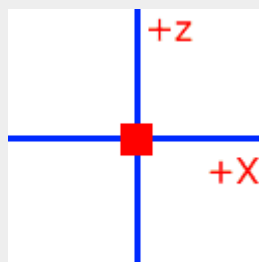
Hemos dicho que para rotar nuestra cámara tenemos que hacer lo mismo con el mundo pero en sentido inverso y lo mismo con la traslación. También hemos dicho que debemos rotar nuestro mundo en relación a nuestra cámara, es decir, que el eje de rotación de nuestro mundo es la posición de la cámara que como hemos dicho es el origen de coordenadas. Pues entonces queda claro que para rotar el mundo alrededor de la cámara primero debemos trasladarlo a la "posición de la cámara" pero invertido y entonces rotarlo la cantidad necesaria también de forma inversa. Es por eso que la concatenación de matrices siga el orden que sigue. Este orden de concatenación es para la cámara pero imaginémonos que queremos trasladar un objeto y rotarlo un determinado número de grados sobre sí mismo. Pues entonces el orden de las concatenaciones sería:

$$\text{WorldMatrix} = \text{RotationYMatrix} * \text{RotationXMatrix} * \text{RotationZMatrix} * \text{TranslationMatrix}$$

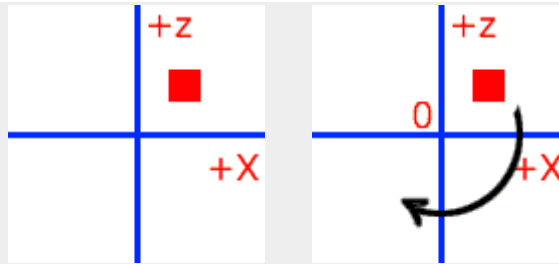
O sea que primero se rota el objeto y después se traslada... Hay que pensar que cada vez que vayamos a realizar una transformación sobre cualquier objeto éste siempre se encuentra en el origen de coordenadas y que el origen de coordenadas siempre es el eje de referencia de todas nuestras transformaciones. Pues bien teniendo en cuenta esto imaginemos que queremos trasladar un objeto a la posición  $X = 10$ ,  $Y = 0$  y  $Z = 10$  y que queremos que ese objeto rote sobre sí mismo en el eje  $Y$   $180^\circ$  por segundo. Bien, pues ahora imaginemos los ejes de coordenadas vistos desde arriba, es decir que veríamos lo siguiente:



Ahora imaginemos nuestro objeto antes de realizar cualquier transformación sobre el origen de coordenadas:



Ahora imaginemos por un momento que realizamos las transformaciones de forma incorrecta, es decir, primero trasladamos y después rotamos:



Como se puede observar, la rotación no se realiza alrededor del objeto en sí sino que lo hace alrededor del origen de coordenadas, porque como hemos dicho todas las transformaciones se realizan tomando como eje de referencia el origen de coordenadas. Ahora es fácil de entender por qué hay que realizar primero la rotación y después la traslación. También queda claro el por qué de realizar primero la traslación y después la rotación cuando nos referimos a la cámara (pensando siempre que la cámara está en el origen de coordenadas).

Antes de pasar al código puntualicemos que las rotaciones reciben un nombre especial. A la rotación en el eje X se le conoce como **pitch**, a la del eje Y se denomina **yaw** y por último a la rotación en el eje Z se le llama **roll**.

Empecemos con la definición de la clase:

```
//-----
// Clase: CCamara
//-----

class CCamara
{
    private:

        float m_fFov;

        int m_iAncho;
        int m_iAlto;

        float m_fPitch;
        float m_fYaw;
        float m_fRoll;

        float m_fPlanoCercano;
        float m_fPlanoLejano;

        D3DXVECTOR3 m_Posicion;

        D3DXVECTOR3 m_Dcha;
        D3DXVECTOR3 m_Arriba;
        D3DXVECTOR3 m_Look;

    public:

        CCamara();
        ~CCamara();

        void ActualizarVista(LPDIRECT3DDEVICE8 pD3DDev);
        void ActualizarProyeccion(LPDIRECT3DDEVICE8 pD3DDev);

        void PonerFov(float fFov);
        float DaFov(void);
}
```

```

void PonerDimensiones(int iAncho, int iAlto);

void PonerPlanos(float fCerca, float fLejos);

void Avanzar(float fVel, float fDTiempo);
void Desplazar(float fVel, float fDTiempo);
void Subir(float fVel, float fDTiempo);

void PonerPosAbs(D3DXVECTOR3 &pos);
D3DXVECTOR3 DaPosicion(void);

void PonerOrientacion(float fPitch, float fYaw, float fRoll);
D3DXVECTOR3 DaOrientacion(void);

D3DXVECTOR3 DaVectorDcha(void);
D3DXVECTOR3 DaVectorArriba(void);
D3DXVECTOR3 DaVectorLookAt(void);

};

```

De ella explicaremos las partes de código más importantes y complejas. Para empezar tenemos los miembros privados de la clase. Podéis ver la variable **m\_fFov** que recibirá el valor del *FOV* que le pongamos a nuestra cámara, **m\_iAncho** y **m\_iAlto** que especifican el ancho y el alto de la pantalla, **m\_fPitch**, **m\_fYaw** y **m\_fRoll** que especifican cada uno de los ángulos de rotación de nuestra cámara, **f\_PlanoCercano** y **m\_fPlanoLejano** que especifican el plano cercano y lejano del *viewing frustum* (para la matriz de proyección), **m\_Posicion** que lleva la posición de la cámara y finalmente tres vectores; **m\_Dcha**, **m\_Arriba** y **m\_Look** los cuales son los vectores que definen la orientación de nuestra cámara. En realidad estos tres vectores también son aplicables a cualquier objeto que no sea la cámara. De estos tres vectores, el que más se utilizará será el vector **m\_Look** (*LookAt*) que es el vector que indica hacia donde apunta la cámara. Este vector se utiliza por ejemplo para el lanzamiento de misiles especificando la dirección de salida del misil.

Pasando a la parte pública tenemos los métodos de la clase. Aquí hay muchas funciones que son obvias y no vamos a perder tiempo en ellas. En lugar de ello os aconsejo que os dirijáis al archivo de implementación de la clase (CCamara.cpp) y miréis los comentarios de cada una de ellas. En la función que sí nos vamos a detener va a ser en **ActualizarVista()** ya que es en esta función donde ponemos en práctica toda la "teoría" que os hemos explicado anteriormente. Veamos su contenido:

```

//-----
// Función: ActualizaVista
// Propósito: Crea una matriz de vista adecuada a los valores especificados
// y la activa.
//-----
void CCamara::ActualizarVista(LPDIRECT3DDEVICE8 pD3DDev)
{
    D3DXMATRIX T, Rx, Ry, Rz;
    D3DXMATRIX view;

    // Cogemos el vector dcha, arriba y lookat actualizados
    pD3DDev->GetTransform(D3DTS_VIEW, &view);
    m_Dcha = D3DXVECTOR3(view._11, view._21, view._31);
    m_Arriba = D3DXVECTOR3(view._12, view._22, view._32);
    m_Look = D3DXVECTOR3(view._13, view._23, view._33);

    // Calculamos la nueva vista
    D3DXMatrixTranslation(&T, -m_Posicion.x, -m_Posicion.y, -m_Posicion.z);
    D3DXMatrixRotationY(&Ry, -m_fYaw);
}

```



```

D3DXMatrixRotationX(&Rx, -m_fPitch);
D3DXMatrixRotationZ(&Rz, -m_fRoll);

view = T * Ry * Rx * Rz;

// La activamos
pD3DDev->SetTransform(D3DTS_VIEW, &view);
}

```

Lo primero que hacemos es actualizar los vectores que definen la orientación de nuestra cámara. Como he dicho hay situaciones en las que son útiles sobre todo el vector *LookAt*. Como podéis ver estos tres vectores los sacamos de la matriz de vista. Esta es la estructura de una matriz de vista:

<b>Right.x</b>	<b>Up.x</b>	<b>Look.x</b>	<b>0</b>
<b>Right.y</b>	<b>Up.y</b>	<b>Look.y</b>	<b>0</b>
<b>Right.z</b>	<b>Up.z</b>	<b>Look.z</b>	<b>0</b>
<b>-</b>	<b>-</b>	<b>-</b>	
<b>DotProduct(Position, Right)</b>	<b>DotProduct(Position, Up)</b>	<b>DotProduct(Position, Look)</b>	<b>1</b>

En ella *DotProduct* es la operación *Producto Escalar* la cual es una operación muy útil y que a partir de ahora utilizaremos más a menudo.

Una vez que hemos cogido los vectores *Right*, *Up* y *Look*, pasamos a crear las matrices de rotación y de traslación como ya sabemos. La única cosa a reseñar es que, como ya hemos dicho al comienzo del tutorial, los valores de traslación y rotación se invierten. Después se crea la matriz de vista concatenando las matrices en el orden que acordamos. Finalmente se activa la matriz resultante como nueva matriz de vista.

Bien, pasemos ahora a otro tema de interés. Centrémonos en la función **Avanzar()**:

```

//-----
// Función: Avanzar
// Propósito: Avanza (fSpeed > 0) / Retrocede (fSpeed < 0) la cámara.
//-----
void CCamara::Avanzar(float fSpeed, float fDTime)
{
    m_Posicion += fSpeed * fDTime * m_Look;
}

```

Como podemos ver, a la posición actual se le suma la velocidad por el tiempo (como todos deberíamos saber la posición es igual a la posición inicial más la velocidad por el incremento del tiempo) pero además se multiplica por el vector *m\_Look*. ¿Por qué?. Bien, ¿qué pasa cuando multiplicamos un escalar (un número normal y corriente) por un vector?... pues que escalamos ese vector una cierta cantidad pero en el sentido y en la dirección que especifica dicho vector. Es por esta razón por la que entra en juego el vector *m\_Look*. Si no especificásemos ningún vector la cámara no sabría hacia dónde moverse. Para entenderlo mejor pondremos un ejemplo. Imaginemos que nuestra cámara está en la posición (0, 0, 0). Ahora imaginemos que la queremos mover a una velocidad de 10 unidades/segundo a lo largo del eje Z positivo (IMPORTANTE: cuando decimos "a lo largo del eje Z positivo" nos estamos refiriendo al eje Z local, es decir, el eje Z de la propia cámara no del mundo. En este caso en concreto da la casualidad; para claridad del ejemplo, de que coinciden) y que nuestro vector *Look* apunta hacia el eje Z positivo (aquí si nos referimos al eje Z positivo del mundo), es decir, (0, 0, 1). La cosa quedaría de la siguiente forma:

$$\mathbf{m\_Posicion} = (0, 0, 0) + 10 * \mathbf{fDTime} * (0, 0, 1)$$

Ahora sustituimos **fDTime** por un valor distinto de cero, por ejemplo 1 segundo:

$$\mathbf{m\_Posicion} = (0, 0, 0) + 10 * 1 * (0, 0, 1)$$

Realizando las operaciones tenemos que:

$$\mathbf{m\_Posicion} = (0, 0, 10)$$

Que es justamente la nueva posición de la cámara que podíamos esperar. Si no hubiésemos multiplicado por el vector *Look*, el resultado hubiese sido:

$$\mathbf{m\_Posicion} = (10, 10, 10)$$

Lo cual es del todo incorrecto. Lo mismo pasa en las funciones **Desplazar()** y en **Subir()**. La diferencia está que en **Desplazar()** utilizamos el vector *Right* y en **Subir()** utilizamos el vector *Up*.

Pues después de hacer todo el trabajo duro sólo nos queda ver cómo lo hemos implementado en nuestro ejemplo, como podemos ver al principio del bucle principal y justo después de actualizar los datos de los dispositivos de entrada hacemos una llamada a la función que nos actualizará los datos de la cámara, **ActualizaCamara()**:

```
//-----
// Función: BuclePpal
// Propósito: Este es el bucle principal del juego
//-----
void CJuego::BuclePpal(void)
{
    if(NULL == m_pD3DDev)
        return;

    Crono->Frame();

    Input->Actualizar();

    // Limpia la pantalla de un color (negro en este caso)
    m_pD3DDev->Clear(0, NULL, D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER,
D3DCOLOR_XRGB(0,0,0), 1.0f, 0);

    // Indica a D3D que en esta sección pondremos nuestras
    // funciones de render
    m_pD3DDev->BeginScene();

    //-----//

    //Actualizamos los movimientos de camara
    ActualizaCamara();

    //Matrices que usaremos
    D3DXMATRIX matWorld, matWorldX, matWorldY, matWorldZ;

    //Matrices de rotacion para los distintos ejes
    D3DXMatrixRotationX(&matWorldX, timeGetTime()/1000.0f);
```

```
D3DXMatrixRotationY(&matWorldY, 0);
D3DXMatrixRotationZ(&matWorldZ, timeGetTime()/1000.0f);
.
.
.
```

La función **ActualizaCamara()** coge los datos de los dispositivos de entrada (que queramos...) y hace los respectivos cambios. En este caso hemos creado sólo un avance y retroceso con el ratón:

```
//-----
// Función: ActualizaCamara()
// Propósito: Especifica las matrices de mundo, de vista y de proyección
//-----
void CJuego::ActualizaCamara(void)
{
    //Avanzamos o retrocedemos la camara 50 unidades * la Y relativa del raton
    Camara->Avanzar(-50*((float)Input->YRelativaRaton()),Crono->DaTiempoDelta());

    //Actualizamos tanto vista como proyeccion
    Camara->ActualizarVista(m_pD3DDev);
    Camara->ActualizarProyeccion(m_pD3DDev);
}
```

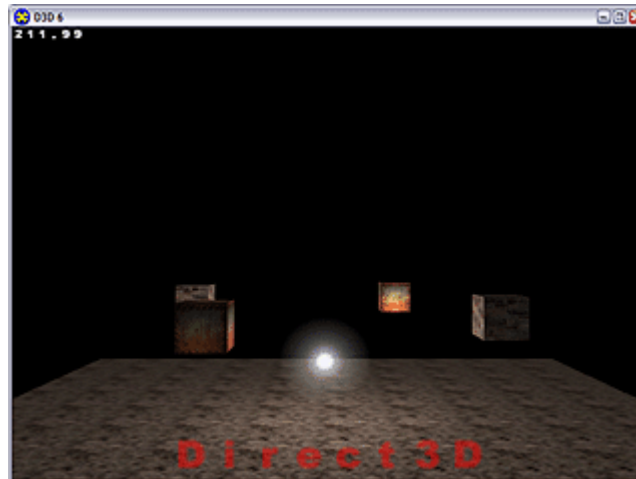
Por supuesto antes de hacer todo esto podíamos haber inicializado nuestra clase para la cámara pero en este ejemplo hemos dejado como suficientes los datos por defecto que asigna el constructor de la clase.

Pues ya está, ya podemos manejar nuestra cámara a través del ratón y teclado. Para poner el punto al tutorial sólo te queda añadir un par de líneas más al ejemplo para que cubra todas las funcionalidades de la cámara que no hemos puesto, como hacer que mire hacia arriba, abajo y a los lados con el ratón, que cambie el *FOV* para efectos de zoom, etc.

## Capítulo 6 - Luces y Alpha blending

Este tutorial va a añadir luces y un efecto transparente al tutorial visto anteriormente. Los únicos cambios considerables en el código a partir del tutorial 5 son: Rediseño de la función

**ActualizaVertices()** de la clase **CCubo**, implementación de la clase **CPanel**, inclusión de un luz oscilante sobre el eje Y, y un efecto de transparencia de acuerdo a la posición de la luz. El resto es mera disposición de cubos y rotaciones para moverlos y la inclusión de fuentes de texto vistas en tutoriales anteriores.



Empecemos con la teoría de luces. Pues bien, en DirectX podemos crear diferentes tipos de luces para hacer nuestras escenas mucho más realistas. Pero el modelo de luces utilizado por este API es tan sólo una aproximación a la iluminación en el mundo real. En la realidad la luz es emitida desde una fuente, por ejemplo una bombilla, y viaja en línea recta hasta que se difumina o llega a nuestros ojos. Según traza su viaje puede interceptar objetos y ser reflejada por estos en diferentes direcciones. Cuando se refleja, el objeto puede absorber parte de la luz. De hecho la luz se refleja millones de veces antes que se difumine o llegue a nuestros ojos. La luz es reflejada de diferentes maneras según qué objetos y la naturaleza del material del que estén hechos. Materiales brillantes reflejan más luces que otros mate. La cantidad de cálculos que se necesitan para simular un modelo así queda muy lejos de lo que pueden hacer nuestros ordenadores en tiempo real, por lo que la iluminación en DirectX es tan sólo una aproximación.

Vista la introducción podemos seguir con los atributos de las luces en DirectX. Vamos a verlos todos, pero esto no quiere decir que todos los tipos utilicen cada uno de estos atributos, por lo que para cada tipo de luz se utilizarán algunos.

### Posición (Source)

Es la posición en coordenadas 3D de la fuente de luz.

### Dirección (Direction)

Es el vector que marca la dirección en la cual la luz es emitida desde el punto de origen (source).

### Rango (Range)

Es la máxima distancia que viajará un haz de luz desde el punto de origen. Todo objeto fuera del rango no recibirá luz desde esa fuente.

### Atenuación (Attenuation)

Indica cómo la luz cambia sobre la distancia. Se puede especificar que la luz no varíe a lo largo del espacio recorrido o podemos hacer que decaiga de la manera que mejor nos convenga.

### **Luz Difusa (Diffuse Light)**

Es el color de la componente difusa de la luz. La luz difusa es aquella que se ha dispersado pero que conserva su dirección.

### **Luz especular (Specular Light)**

Es el color de la componente especular de la luz. La luz especular, al contrario que la anterior, es aquella que no se ha dispersado apenas. Se usa para crear brillos metálicos en los objetos.

### **Luz Ambiental (Ambient Light)**

Es el color de la componente ambiental de la luz. La luz ambiental suele ser luz de fondo. Esta luz se ha dispersado tanto que ha perdido su dirección, y su fuente de origen es igual para cualquier punto de la escena.

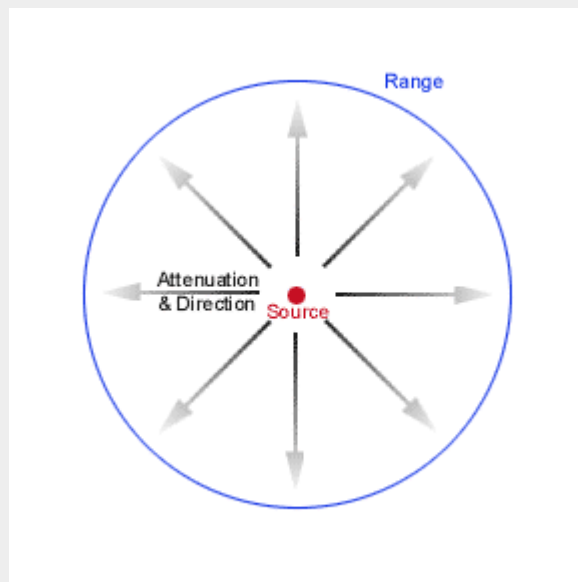
A continuación vamos a ver los distintos tipos de luces en DirectX:

### **Ambiente (Ambient)**

Es la posición en coordenadas 3D de la fuente de luz.

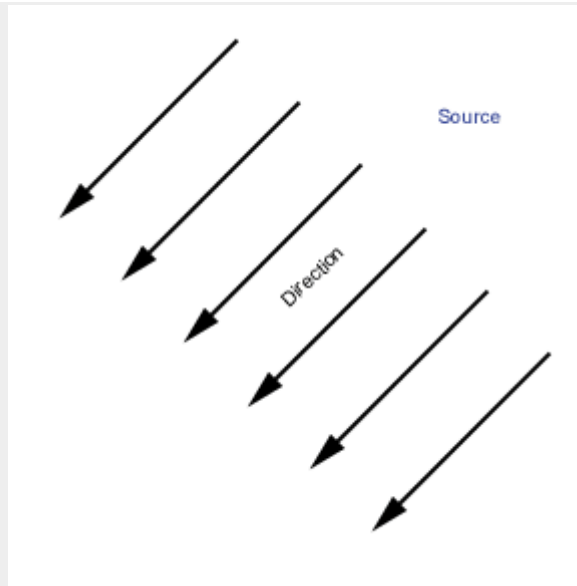
### **Puntual (Point)**

Un ejemplo de luz puntual es una bombilla. Ésta tiene posición pero no dirección ya que emite luz en todas las direcciones igualmente. También tiene color, rango y atenuación.



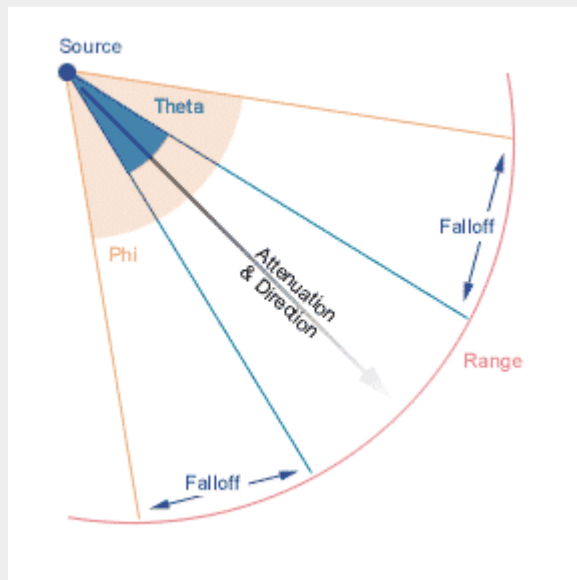
### **Direccional (Directional)**

Las luces direccionales tienen dirección y color pero no posición por lo que tampoco tendrán atributos de atenuación. La posición en estas luces se simula que está en el infinito. Un ejemplo de luz direccional puede ser el sol, que considerando su distancia a la tierra puede considerarse infinito, ya que casi nos llegan sus rayos de forma paralela. Todos los objetos de la escena recibirán la misma luz desde la misma dirección.



### Foco (Spotlight)

Un ejemplo de foco puede ser una linterna. Los focos tienen posición, dirección, color, atenuación y rango. Para un foco se pueden definir dos conos, uno exterior y otro interior, con los que la luz se mezcla entre los dos. Para poder definir los conos se especifican mediante su ángulo. El ángulo del cono interior se conoce como Theta y el del exterior como Phi. Para definir como cambia la luz entre los dos conos usaremos el valor de caída (fall of).



Todos estos tipos de luz añaden tiempo de computación a la aplicación, algunos más que otros. La luz que menos carga es la ambiental, le sigue la direccional, luego las puntuales y finalmente los focos. Hay que tener muy en cuenta esto a la hora de decidir que tipo de luz queremos usar.

Prosigamos con los materiales. ¿Qué es un material? Un material describe cómo la luz es reflejada por un objeto (polígono). Podemos especificar cuánto se refleja para hacer el objeto más brillante o mate. También podemos poner un color para el objeto. Aquí están las propiedades que se le pueden atribuir a un material:

### Reflexión difusa (Diffuse Reflection)

Es la cantidad de luz difusa que el objeto reflejará. Se define mediante un color así que podemos especificar que el objeto sólo reflejará sólo luz roja, o la que queramos...

### Reflexión ambiental (Ambient Reflection)

Es la cantidad de luz ambiente que reflejará. De la misma manera que la anterior se define mediante un color por lo que podemos hacer que refleje sólo tonalidades de un color.

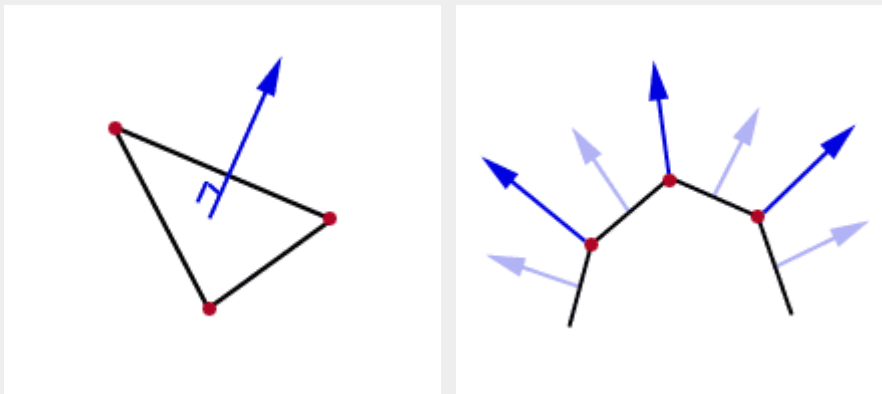
### Reflexión especular y Potencia (Specular Reflection y Power)

Es la cantidad de luz especular reflejada. Usaremos esta propiedad junto a la de Potencia (Power) para crear brillos metálicos en los objetos.

### Emisión (Emission)

También podemos hacer que un objeto simule que emite luz cambiando la propiedad de Emisión. El objeto realmente no emite luz, por lo que el resto de objetos no se verán afectados por esta emisión.

Por último sólo nos queda hablar de las normales. La Normal de un polígono es el vector perpendicular a la cara de ese polígono. La dirección de la normal viene determinada por el valor que le hayamos dado a la hora de crear los vértices. La Normal de un vértice es, normalmente, la media de las normales de las caras que comparten ese vértice. Las normales se usan para muchas cosas, pero en este tutorial las necesitaremos para que Direct3D haga los cálculos necesarios sobre la incisión de las luces en nuestros polígonos.



Vista la teoría pasemos al código. Lo primero que hemos hecho, como ya se ha dicho, es modificar la clase **CCubo** y principalmente la función **ActualizarVértices()**, pero antes de ver ésta función vamos a ver cómo ha quedado definido nuestro nuevo tipo de vértices con normales:

```
#define D3DFVF_VERTCUBO (D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1)
.
.
.
struct VERTCUBO
{
    FLOAT x, y, z;
    FLOAT nx, ny, nz;
    FLOAT tu, tv;
};
```

Hemos quitado el componente de color (ya que no nos va a servir de mucho si iluminamos con

Direct3D) y hemos añadido la normal como otro vector más. Ahora veamos como ha quedado **ActualizarVertices()**:

```
//-----
// Función: ActualizarVertices
// Propósito: Actualiza los cambios en los vértices
//-----
BOOL CCubo::ActualizarVertices()
{
    VOID* pVertices;
    D3DVECTOR vNormal;

    // asignamos vértices que forman un cubo con un color para cada vértice
    // hay que asegurarse que los vertices se definen en el sentido de
    // las agujas del reloj ya que las caras contrarias las eliminamos con el
    culling
    VERTCUBO cvVertices[] =
    {
        //Cara Superior
        {m_rX - (Ancho / 2), m_rY + (Alto / 2), m_rZ - (Largo / 2), 0.0f, 0.0f,
0.0f, 0.0f, 1.0f,}, //Vertex 0
        {m_rX - (Ancho / 2), m_rY + (Alto / 2), m_rZ + (Largo / 2), 0.0f, 0.0f,
0.0f, 0.0f, 0.0f,}, //Vertex 1
        {m_rX + (Ancho / 2), m_rY + (Alto / 2), m_rZ - (Largo / 2), 0.0f, 0.0f,
0.0f, 1.0f, 1.0f,}, //Vertex 2
        {m_rX + (Ancho / 2), m_rY + (Alto / 2), m_rZ + (Largo / 2), 0.0f, 0.0f,
0.0f, 1.0f, 0.0f,}, //Vertex 3
        {m_rX + (Ancho / 2), m_rY + (Alto / 2), m_rZ - (Largo / 2), 0.0f, 0.0f,
0.0f, 1.0f, 1.0f,}, //Vertex 4
        {m_rX - (Ancho / 2), m_rY + (Alto / 2), m_rZ + (Largo / 2), 0.0f, 0.0f,
0.0f, 0.0f, 0.0f,}, //Vertex 5

        //Cara 1
        .
        .
        .

        .
        .
        .

        //Cara Inferior
        {m_rX + (Ancho / 2), m_rY - (Alto / 2), m_rZ - (Largo / 2), 0.0f, 0.0f,
0.0f, 0.0f, 1.0f,}, //Vertex 30
        {m_rX + (Ancho / 2), m_rY - (Alto / 2), m_rZ + (Largo / 2), 0.0f, 0.0f,
0.0f, 0.0f, 0.0f,}, //Vertex 31
        {m_rX - (Ancho / 2), m_rY - (Alto / 2), m_rZ - (Largo / 2), 0.0f, 0.0f,
0.0f, 1.0f, 1.0f,}, //Vertex 32
        {m_rX - (Ancho / 2), m_rY - (Alto / 2), m_rZ + (Largo / 2), 0.0f, 0.0f,
0.0f, 1.0f, 0.0f,}, //Vertex 33
        {m_rX - (Ancho / 2), m_rY - (Alto / 2), m_rZ - (Largo / 2), 0.0f, 0.0f,
0.0f, 1.0f, 1.0f,}, //Vertex 34
        {m_rX + (Ancho / 2), m_rY - (Alto / 2), m_rZ + (Largo / 2), 0.0f, 0.0f,
0.0f, 0.0f, 0.0f,}, //Vertex 35
    };

    //Poner las normales
    int i;

    for(i = 0; i < 36; i += 3)
    {
        vNormal = DaNormalTriangulo(&D3DXVECTOR3(cvVertices[i].x, cvVertices[i].y,
```



```

cvVertices[i].z), &D3DXVECTOR3(cvVertices[i + 1].x, cvVertices[i + 1].y,
cvVertices[i + 1].z), &D3DXVECTOR3(cvVertices[i + 2].x, cvVertices[i + 2].y,
cvVertices[i + 2].z));

    cvVertices[i].nx = vNormal.x;
    cvVertices[i].ny = vNormal.y;
    cvVertices[i].nz = vNormal.z;

    cvVertices[i + 1].nx = vNormal.x;
    cvVertices[i + 1].ny = vNormal.y;
    cvVertices[i + 1].nz = vNormal.z;

    cvVertices[i + 2].nx = vNormal.x;
    cvVertices[i + 2].ny = vNormal.y;
    cvVertices[i + 2].nz = vNormal.z;

}

if(FAILED(BufferVert->Lock(0, sizeof(cvVertices), &pVertices, 0)))
{
    return FALSE;
}

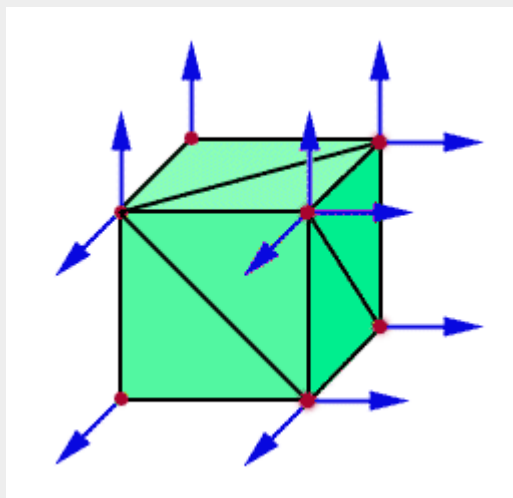
memcpy(pVertices, cvVertices, sizeof(cvVertices));

BufferVert->Unlock();

return TRUE;
}

```

Hemos cambiado los triangle strips por triangle list, ya que con strips las caras comparten vértices y puesto que vamos a tener normales para cada vértice necesitamos que los triángulos no compartan sus normales, ya que de esta forma la iluminación no sería correcta. Por lo demás la primera parte del código se corresponde con la creación de un cubo a base de triangle list. Después viene la parte de las normales en la que para cada triángulo del cubo se llama a la función **DaNormalTriangulo()** para que nos devuelva la normal (perpendicular se entiende) a ese triángulo. De esta manera asignamos a cada triángulo del cubo su normal. Una vez calculada esta normal para el triángulo se le asigna a sus tres vértices quedando un cubo de ésta manera:



Pasemos a ver cómo se extrae la normal mediante **DaNormalTriangulo()**:

```
//-----
// Función: DaNormalTriangulo
// Propósito: Devuelve la normal de un triángulo
//-----
D3DXVECTOR CCubo::DaNormalTriangulo(D3DXVECTOR3* vVertice1, D3DXVECTOR3* vVertice2,
D3DXVECTOR3* vVertice3)
{
    D3DXVECTOR3 vNormal;
    D3DXVECTOR3 v1;
    D3DXVECTOR3 v2;

    D3DXVec3Subtract(&v1, vVertice2, vVertice1);
    D3DXVec3Subtract(&v2, vVertice3, vVertice1);

    D3DXVec3Cross(&vNormal, &v1, &v2);

    D3DXVec3Normalize(&vNormal, &vNormal);

    return vNormal;
}
```

Como podemos ver, la función recibe los 3 vértices del triángulo y los procesa de tal forma que halla la resta entre dos vértices para encontrar dos vectores coplanarios con el triángulo y que compartan un vértice de éste para luego hallar el producto escalar entre estos vectores para conseguir el vector perpendicular a ambos. Finalmente lo normaliza ya que las normales deben ser vectores unitarios para una correcta iluminación. Con esto tenemos la normal de ese vértice. Pondremos todos los vértices de la misma cara con esa normal ya que queremos aristas rectas. Si quisiéramos una iluminación más difuminada entre cara y cara utilizaríamos la media de las normales de las caras para cada vértice, como en esferas.

Ya tenemos nuestra geometría casi lista para recibir iluminación, sólo nos queda un pequeño detalle: el material. Necesitamos añadir un material a cada objeto ya que si no no se vería. Hemos añadido una nueva variable de tipo **D3DMATERIAL8** a la clase para este efecto. Aquí podemos ver su declaración y su inicialización por medio de la función **Material()**:

```
D3DMATERIAL8 m_matMaterial;
.
.
.
//ponemos un material estandar (R, G, B, A)
D3DCOLORVALUE rgbaDifusa = {1.0, 1.0, 1.0, 0.0,};
D3DCOLORVALUE rgbaAmbiente = {1.0, 1.0, 1.0, 0.0,};
D3DCOLORVALUE rgbaEspecular = {0.0, 0.0, 0.0, 0.0,};
D3DCOLORVALUE rgbaEmisiva = {0.0, 0.0, 0.0, 0.0,};

Material(rgbaDifusa, rgbaAmbiente, rgbaEspecular, rgbaEmisiva, 0);
.
.
.
//-----
// Función: Material
// Propósito: Inicializa el material para el cubo
//-----
BOOL CCubo::Material(D3DCOLORVALUE rgbaDifuso, D3DCOLORVALUE rgbaAmbiente,
D3DCOLORVALUE rgbaEspecular, D3DCOLORVALUE rgbaEmisivo, float rPotencia)
{
    m_matMaterial.Diffuse = rgbaDifuso;

    m_matMaterial.Ambient = rgbaAmbiente;
```

```

m_matMaterial.Specular = rgbaEspecular;

m_matMaterial.Power = rPotencia;

m_matMaterial.Emissive = rgbaEmisivo;

return TRUE;
}

```

Lo único que hemos hecho es asignar unos valores por defecto de tipo en el constructor de la clase por medio de ésta función. Con esto hecho la geometría queda lista para recibir luces.

Pasemos ahora a la inicialización de las luces en la clase **CJuego**. Lo primero es que hemos puesto una nueva variable privada **Luz** a esta clase que va a representar a nuestra luz. Esta variable es del tipo **D3DLIGHT8**. Luego hemos añadido una línea con la función **InicializarLuces()** a la inicialización del juego. Veamos el código de esta función:

```

//-----
// Función: InicializarLuces
// Propósito: Inicializa las luces del juego
//-----
BOOL CJuego::InicializarLuces(void)
{
    //Inicializar la estructura a cero
    ZeroMemory(&Luz, sizeof(D3DLIGHT8));

    //Ponemos un punto de luz blanca en (0, 0, 0).
    Luz.Type = D3DLIGHT_POINT;

    Luz.Diffuse.r = 1.0f;
    Luz.Diffuse.g = 1.0f;
    Luz.Diffuse.b = 1.0f;

    Luz.Ambient.r = 0.0f;
    Luz.Ambient.g = 0.0f;
    Luz.Ambient.b = 0.0f;

    Luz.Specular.r = 0.0f;
    Luz.Specular.g = 1.0f;
    Luz.Specular.b = 0.0f;

    Luz.Position.x = 0.0f;
    Luz.Position.y = 0.0f;
    Luz.Position.z = 0.0f;

    Luz.Attenuation0 = 1.0f;
    Luz.Attenuation1 = 0.0f;
    Luz.Attenuation2 = 0.0f;

    Luz.Range = 250.0f;

    //Asignamos el punto de luz al dispositivo en la posición 0
    if(FAILED(m_pD3DDev->SetLight(0, &Luz)))
    {
        return FALSE;
    }

    // "encendemos" la luz de la posición 0 (la que asignamos antes)
    if(FAILED(m_pD3DDev->LightEnable(0, TRUE)))

```

```

{
    return FALSE;
}

//Ponemos el nivel de luz ambiental
if(FAILED(m_pD3DDev->SetRenderState(D3DRS_AMBIENT, D3DCOLOR_XRGB(64,64,64))))
{
    return FALSE;
}

return TRUE;
}

```

Lo primero es inicializar la estructura a cero. Luego pasamos a asignar los valores que van a definir nuestra luz. La hemos creado de tipo **D3DLIGHT\_POINT** ya que queremos una luz puntual. Si quisiéramos un direccional pondríamos **D3DLIGHT\_DIRECTIONAL** y si quisiésemos un foco pondríamos **D3DLIGHT\_SPOT**. Luego hemos definido sus valores así: color blanco para la componente difusa, nada para ambiental y verde para la especular. Su posición en (0,0,0) y su atenuación a 1. Por fin el rango lo hemos puesto a un máximo de 250. Para asignar la luz al dispositivo utilizamos **SetLight()**, función miembro del objeto dispositivo pasándole la estructura de la luz y un índice que identificará a la luz. El número máximo de este índice (o número máximo de luces) viene determinado por el dispositivo. Proseguimos a encender la luz mediante **LightEnable()** también miembro del mismo objeto y finalmente ponemos el color de la luz ambiente mediante un *RenderState*. Ponemos un gris intermedio para que las cosas que no ilumine nuestra luz se vean un poco más claras y no totalmente oscuras.

La luz ya está creada y ya está emitiendo. Hay que tener en cuenta que al renderizar nuestro cubo (o nuestro panel como se verá más adelante) tenemos que activar su material como si de su textura se tratase. Estas líneas incluidas dentro de la función de renderizar del cubo justo antes de pintarlo activan el material:

```

if(FAILED(D3DDisp->SetMaterial(&m_matMaterial)))
{
    //Fallo al iniciar el material
    return FALSE;
}

```

Pues ya está, lo único que hemos hecho a parte de esto es animar la posición de la luz en cada frame mediante **ActualizarLuces()**:

```

//-----
// Función: ActualizarLuces
// Propósito: Inicializa las luces del juego
//-----
BOOL CJuego::ActualizarLuces(void)
{
    Luz.Position.x=0;
    Luz.Position.y=25*sinf(timeGetTime()/700.0f);
    Luz.Position.z=0;

    if(FAILED(m_pD3DDev->SetLight(0, &Luz)))
    {
        return FALSE;
    }

    return TRUE;
}

```

Pues ya tenemos la luz lista y oscilando sobre el eje Y. Nos queda ver cómo hemos hecho el suelo y ese destello que acompaña a nuestra luz. Para ello hemos creado un clase llamada **CPanel** que no tiene mucho misterio comparándola con **CCubo**. Para el suelo hemos utilizado una textura normal y corriente y hemos inicializado el objeto para este panel de tal forma que repita unas cuantas veces la textura a base de crear más triángulos en la malla del panel. Para el destello lo hemos hecho del mismo modo, aunque sólo hemos puesto que se repita una vez, pero justo antes de pintarlo, en su función de **CargarEstados()**, hemos activado el Alpha blending. ¿Y esto qué es?

El Alpha blending es una de las técnicas más utilizadas en los juegos de hoy en día. Pensemos en todos los objetos a través de los cuales se puede ver lo que hay al otro lado. Todos estos objetos utilizan el alpha blending para este fin.

El canal alpha de una imagen no es más que una imagen de 8 bits en escala de grises. D3D (y OGL) interpretan esta información como sigue:

- Un valor alpha de 0 (color negro) indica transparencia total, es decir, que la zona del polígono afectada por este valor no se verá.
- Un valor de 128 (color gris "puro") indica semitransparencia.
- Un valor de 255 (color blanco) indica opacidad total.

Esta forma de interpretación la podemos variar mediante *RenderStates* pero por ahora nos vendrá bien así. A partir de ahora usaremos el formato TGA para las texturas que queramos transparentes. La razón principal por la que utilizaremos este formato es porque en su interior se puede almacenar, junto con los valores RGB (red, green, blue) un canal adicional que es precisamente este canal alpha, por lo que el TGA pasa de ocupar 24 bits a 32 bits.

Hay tres formas de pasar a D3D estas componentes alpha para realizar transparencias:

- A través de los vértices de nuestra geometría.
- A través de un material.
- A través de una textura.

Evidentemente este último es el que vamos a utilizar. Una de las ventajas de este método es que podemos asignar componentes alpha a zonas concretas de nuestra geometría, cosa que con un material o a través de los vértices no podemos hacer.

Para crear imágenes con canal Alpha utilizaremos Photoshop u otro programa gráfico que lo soporte. Para empezar vamos a abrir la textura destello.tga para verla en Photoshop. Podemos comprobar que es toda blanca, excepto el canal alpha que tiene dibujado el destello en sí en escala de grises. Con esto vamos a conseguir que mediante éste canal pintemos algo del blanco original con la transparencia correspondiente al canal alpha.

Para crear el efecto transparente en nuestra aplicación lo primero es especificarle a Direct3D las funciones de mezcla. Hay varios tipos. Por ejemplo el *additive blending* para crear un flare (también llamado corona o glow) y el *filter blending* que se utiliza por ejemplo para los lightmaps. Pero en este tutorial, el blending que necesitamos es el *alpha blending*. Para ello debemos activarlo con la función **SetRenderState()**. Los argumentos que esta función debe tomar para activar el alpha blending debe ser: para **D3DRS\_SRCBLEND** activamos el tipo **D3DBLEND\_SRCALPHA** y para **D3DRS\_DESTBLEND** activamos **D3DBLEND\_INVSRCALPHA**. Para entender esto primero

veremos la fórmula de este tipo de blending:

$$\begin{aligned}\text{Red\_Final} &= \text{Red\_Fuente} * \text{Alpha} + \text{Red\_Destino} * (1 - \text{Alpha}) \\ \text{Green\_Final} &= \text{Green\_Fuente} * \text{Alpha} + \text{Green\_Destino} * (1 - \text{Alpha}) \\ \text{Blue\_Final} &= \text{Blue\_Fuente} * \text{Alpha} + \text{Blue\_Destino} * (1 - \text{Alpha})\end{aligned}$$

Ahora veamos un ejemplo. Cojamos el caso en el que nos encontramos. Tenemos que la parte del panel que coincide con los píxeles negros de la textura del destello no debe visualizarse pero no porque sea negra sino porque la parte del canal alpha que le corresponde tiene un valor alpha = 0, es decir, 100% transparente. Eso quiere decir que el color en esa zona debe ser el del fondo y no el negro de la textura (porque esta es 100% transparente). Con el resto de la gama de grises hasta el blanco la transparencia no es 100% completa pero sí semitransparente, dejando ver algo del blanco original de la imagen.

Veamos ahora el código que activa nuestro alpha blending para el panel en **CargarEstados()**:

```
.
.
.
D3DDisp->SetRenderState(D3DRS_ALPHABLENDENABLE, m_bCanalAlpha);
D3DDisp->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
D3DDisp->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);
.
.
.
```

Si está activada la variable miembro **m\_bCanalAlpha** se activará el blending. Las otras dos líneas son las operaciones antes mencionadas para que Direct3D lleve acabo la operación de mezcla. Recomiendo mirar en el SDK los posibles valores que pueden tomar para jugar con ellos y ver los resultados que producen. Sólo con la práctica se dominará perfectamente esta técnica de transparencia. También decir que no es necesario tener un canal alpha en la textura, en este caso Direct3D tomará el alpha de la propia luminancia de los colores RGB en la textura, aunque requiere de vuestra práctica para verlo más claramente.

## Capítulo 7 - Utilización de Mallas

En este tutorial veremos cómo cargar mallas con el formato de DirectX (.x) y cómo renderizarlas ajustándolas para la iluminación. Hemos tomado la misma luz oscilante del tutorial anterior. Los cambios más significativos se encuentran en **CJuego** y **CMalla**. Para el ejemplo hemos tomado un archivo .x del SDK de DirectX que representa un tigre con su correspondiente textura, pero podríamos haber cogido cualquier malla creada con algún paquete gráfico y exportada a este formato.



Centrémonos en la clase **CMalla**. Veamos la definición de la clase:

```
//-----
// Clase: CMalla
//-----

#define D3DFVF_VERTMALLA (D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1)

class CMalla
{
public:
    DWORD Renderizar();
    CMalla(LPDIRECT3DDEVICE8 pD3DDevice, LPSTR pNombreArchivo);
    ~CMalla();

private:
    void CargarEstados(void);

    LPDIRECT3DDEVICE9 m_pD3DDevice;
    DWORD m_dwNumMateriales;
    LPD3DXMESH m_pMalla;
    D3DMATERIAL9* m_pMallaMateriales;
    LPDIRECT3DTEXTURE9* m_pMallaTexturas;
};
```

Como podemos ver en el código no tenemos muchas funciones para esta clase. Al constructor le pasamos el dispositivo y el nombre de archivo de la malla en cuestión. Luego tenemos la función para renderizar nuestra malla, la función que carga los estados que necesitamos y el destructor correspondiente de la clase. Variables tenemos las siguientes: la variable para el dispositivo, un

indicador de tipo **DWORD** que nos dirá el número de materiales en la malla, un array dinámico de materiales y otro de texturas y por fin el puntero a la malla de tipo **LPD3DXMESH**. Veamos ahora el constructor:

```
//-----
// Función: Constructor
//-----
CMalla::CMalla(LPDIRECT3DDEVICE9 pD3DDevice, LPSTR pNombreArchivo)
{
    LPD3DXBUFFER pMaterialsBuffer = NULL;
    LPD3DXMESH pMesh = NULL;

    m_pD3DDevice = pD3DDevice;

    if(FAILED(D3DXLoadMeshFromX(pNombreArchivo, D3DXMESH_SYSTEMMEM, m_pD3DDevice,
    NULL,
    &pMaterialsBuffer, NULL, &m_dwNumMateriales, &pMesh)))
    {
        m_pMalla = NULL;
        m_pMallaMateriales = NULL;
        m_pMallaTexturas = NULL;

        return;
    }

    D3DXMATERIAL* matMaterials = (D3DXMATERIAL*)pMaterialsBuffer-
    >GetBufferPointer();

    //Crea dos arrays, uno para los materiales y otro para las texturas
    m_pMallaMateriales = new D3DMATERIAL8[m_dwNumMateriales];
    m_pMallaTexturas = new LPDIRECT3DTEXTURE8[m_dwNumMateriales];

    for(DWORD i = 0; i < m_dwNumMateriales; i++)
    {
        // Copia el material
        m_pMallaMateriales[i] = matMaterials[i].MatD3D;

        // Pone el color ambiental del material (D3DX no lo hace sólo)
        m_pMallaMateriales[i].Ambient = m_pMallaMateriales[i].Diffuse;

        // Crea la textura
        if(FAILED(D3DXCreateTextureFromFile(m_pD3DDevice,
        matMaterials[i].pTextureFilename,
        &m_pMallaTexturas[i])))
        {
            m_pMallaTexturas[i] = NULL;
        }
    }

    //Hemos terminado con el material así que lo eliminamos
    BorrarInter(pMaterialsBuffer);

    // Nos aseguramos que las normales están listas para nuestra malla
    pMesh->CloneMeshFVF(D3DXMESH_MANAGED, D3DFVF_VERTMALLA, m_pD3DDevice,
    &m_pMalla);
    BorrarInter(pMesh);

    D3DXComputeNormals(m_pMalla, NULL);
}
```



```
}
```

El constructor toma dos parámetros, un puntero al dispositivo y una cadena que nos indica donde se encuentra el archivo .x. Para cargar la malla en memoria utilizamos la función **D3DXLoadMeshFromX()**. Justo después creamos dos *arrays* que van a llevar las texturas y los materiales de nuestro modelo. Rellenamos los *arrays* con los materiales de la malla y las texturas que use. Después clonamos la malla y la pasamos al objeto que finalmente la contendrá, para al finalizar computar sus normales para inicializarlas con la función **D3DXComputeNormals()**. En el destructor de la clase nos encargamos de borrar estos dos *arrays* y el objeto de la malla en cuestión:

```
//-----
// Función: Destructor
//-----
CMalla::~CMalla()
{
    BorrarObj(m_pMallaMateriales);

    if(m_pMallaTexturas != NULL)
    {
        for(DWORD i = 0; i < m_dwNumMateriales; i++)
        {
            if(m_pMallaTexturas[i])
            {
                BorrarInter(m_pMallaTexturas[i]);
            }
        }

        BorrarObj(m_pMallaTexturas);
        BorrarInter(m_pMalla);
    }
}
```

Para renderizarla tan sólo tenemos que recorrer todas las subpartes de la malla y renderizarlas con su textura, material y estados apropiados:

```
//-----
// Función: Renderizar
// Propósito: Renderiza la malla
//-----
DWORD CMalla::Renderizar()
{
    CargarEstados();

    if(m_pMalla != NULL)
    {
        for(DWORD i = 0; i < m_dwNumMateriales; i++)
        {
            m_pD3DDevice->SetMaterial(&m_pMallaMateriales[i]);
            m_pD3DDevice->SetTexture(0, m_pMallaTexturas[i]);

            m_pMalla->DrawSubset(i);
        }

        return m_pMalla->GetNumFaces();
    }
    else
    {
        return 0;
    }
}
```

Sólo queda puntualizar que al escalar nuestra malla con matrices de transformación también estaremos escalando sus normales con lo que puede que esto afecte a la iluminación (si la agrandamos se verá más oscura). Para solucionar esto debemos activar el estado de render siguiente:

```
m_pD3DDev->SetRenderState(D3DRS_NORMALIZENORMALS, TRUE);
```

Terminado. Ya podemos incluir en nuestros programas mallas de tipo .x sin más complicación.

## Capítulo 8 - Cargando niveles del Quake 3 (Parte 1)

En esta primera parte veremos cómo manejar la geometría incluida en los niveles .bsp del juego Quake 3. Para ello he creado un pequeño nivel con Q3Radiant (el editor de niveles del Quake 3) y lo he compilado para que nos genere el archivo bsp. El nivel así como las texturas que usa están incluidos en el código fuente de este artículo. Una vez tengamos nuestro archivo .bsp hay que proceder a abrirlo desde nuestro programa, pero para eso debemos tener muchas cosas bastante claras. Paciencia porque este "tuto" es muuuyy laaargooo :D

Lo primero que debemos aprendernos es el formato de archivo del .bsp, así como la forma de manejarlo. Hay que recalcar que aquí voy a estar hablando siempre del .bsp de Quake 3 que es distinto de otros como por ejemplo Half-Life. Este formato almacena gran parte de la información del nivel. Hay otros archivos en Quake 3 como los .shader .arena y .ass que almacenan otras cosas como información sobre texturas, bots, etc.

El .bsp esta dividido en zonas llamadas bloques (*lumps* o *chunks* en inglés), que almacenan cada parte del archivo (texturas, lightmaps, polígonos...). Cada bloque tiene su longitud y su desplazamiento dentro del archivo (offset). Aquí tenemos el enumerado que contiene todos los bloques existentes en el archivo y su orden:

```
enum eBloquess
{
    kEntidades = 0, // Almacena posicion de jugadores, objetos, etc
    kTexturas,      // Almacena informacion de texturas
    kPlanos,        // Almacena los planos de particion
    kNodos,         // Almacena los nodos BSP
    kHojas,         // Almacena las hojas de los nodos
    kPoliHojas,     // Almacena los indices de las hojas en las caras
    kVoluHojas,     // Almacena los indices de las hojas en los volúmenes
    kModelos,       // Almacena la informacion de los modelos
    kVolumenes,     // Almacena la información de los volúmenes (para colisión)
    kVoluLados,     // Almacena la información de las superficies del volumen
    kVertices,      // Almacena los verices del nivel
    kVertMalla,     // Almacena los desplazamientos de los vertices de modelos
    kShaders,       // Almacena los archivos de shader
    kPolis,         // Almacena los poligonos del nivel
    kLightmaps,     // Almacena los lightmaps del nivel
    kLightVolumes,  // Almacena informacion extra sobre iluminacion
    kVisData,       // Almacena el PVS (Potential Visibility Set) para el culling
    kMaxBloques     // Una constante para saber el numero de bloques
};
```

Como hemos dicho antes cada una de estas secciones contiene un desplazamiento y un tamaño en bytes para poder ser leídas. Aquí tenemos la estructura para los bloques. El desplazamiento es la posición en el archivo que indica el comienzo del bloque actual y la longitud es el número de bytes que ocupa el bloque.

```
struct tBSPBloque
{
    int desplaz; // El desplazamiento en el archivo para el comienzo de este
    bloque
    int longitud; // Longitud en bytes del bloque
};
```

Lo primero que debemos hacer es leer del archivo la información sobre los bloques:

```
tBSPBloque bloques[kMaxBloques] = {0};
```

```
fread(&bloques, kMaxBloques, sizeof(tBSPBloque), pArchivo);
```

Con esto ya tenemos en el array de bloques la información sobre los mismos. Vamos a poner un ejemplo de cómo leer los vértices (**kVertices**) del nivel. Para saber el número de vértices contenidos en el bloque de vértices procederemos así:

```
NumDeVerts = bloques[kVertices].longitud / sizeof(tBSPVertice);
```

Es sencillo, nos dirigimos al bloque indicado por **kVertices** (que contiene los vértices según la definición del enumerado) y dividimos su longitud entre el número de bytes que ocupa una única estructura de vértices (ya que el bloque de vértices está compuesto por N estructuras de vértices). **tBSPVertice** ya la definiremos más adelante. Si la longitud del bloque es (por ejemplo) de 3388 bytes y la estructura de vértices ocupa 44, el resultado será  $3388 / 44 = 77$ . Ahora podemos saber que el número de vértices en el .bsp es de 77. Luego ya podremos posicionarnos en el comienzo del bloque con el desplazamiento y comenzar a leer 77 estructuras **tBSPVertice** en un array dinámico específico para ello.

Ya que conocemos cómo leer cada bloque del archivo vamos a ver las estructuras que usaremos en esta parte. Ya hemos visto **tBSPBloque**, veamos ahora la cabecera:

```
struct tBSPHeader
{
    char strID[4];      // Siempre deberá ser 'IBSP'
    int version;        // Debe ser 0x2e para archivos del Quake 3
};
```

Es lo primero que debe leerse del archivo, contiene 4 bytes que indican el ID y un *integer* para la versión. Para leerla haremos esto:

```
tBSPHeader header = {0};

fread(&header, 1, sizeof(tBSPHeader), pArchivo);
```

Así de simple. Veamos ahora la estructura de los vértices de la que hablábamos antes:

```
struct tBSPVertice
{
    VECTOR3BSP vPosicion;      // Posicion (x, y, z)
    VECTOR2BSP vTexturaCoord;  // Coordenadas de textura (u, v)
    VECTOR2BSP vLightmapCoord; // Coordenadas de lightmap (u, v)
    VECTOR3BSP vNormal;        // Vector normal (x, y, z)
    byte color[4];             // color RGBA para el vértice
};
```

Esta estructura contiene la posición del vértice, coordenadas de textura y lightmap, normal y un color RGBA para el vértice. Como vemos el tipo de vector usado no es el de DirectX, (más que nada porque Quake 3 fue hecho para OpenGL). Quake 3 utiliza 3 *floats* para los vectores 3D y 2 para los 2D. También en algún momento utiliza vectores de 3 *ints* pero eso ya lo veremos más adelante.

```
typedef struct sVECTOR3BSP
{
    float x, y, z;
} VECTOR3BSP;

typedef struct sVECTOR2BSP
{
    float x, y;
```

```

} VECTOR2BSP;

typedef struct sVECTOR3BSPi
{
    int x, y, z;
}VECTOR3BSPi;

```

Veamos ahora la estructura de los polígonos ("polis" en el código fuente):

```

struct tBSPPoli
{
    int texturaID;           // El índice en el array de texturas
    int efecto;              // El índice para los efectos (-1 significa
desactivado)
    int tipo;                // 1=poligono, 2=superficie curva, 3=malla,
4=billboard
    int startVertIndex;      // El primer índice para el primer vertice del
poligono
    int numDeVerts;          // Numero de vertices del poli
    int meshVertIndex;       // El índice en el primer vertice de malla
    int numMeshVerts;        // El numero de vertices en la malla
    int lightmapID;          // El índice de textura para el lightmap
    int lMapCorner[2];        // La esquina del lightmap en la imagen
    int lMapSize[2];         // El tamaño de la sección del lightmap
    VECTOR3BSP lMapPos;       // El punto de origen 3D del lightmap.
    VECTOR3BSP lMapVecs[2];   // Coordenadas de los vectores unitarios s y t
    VECTOR3BSP vNormal;       // La normal del polígono
    int size[2];             // Los tamaños de la superficie curva
};

```

Esta estructura almacena todos los datos del polígono en cuestión. Lo más importante son los índices a los vértices que usa y la textura. Pero ya veremos cómo usar esto para renderizar. Sólo queda aclarar que si el tipo es igual a '1' se tratará de un polígono normal y se podrá renderizar como un *triangle fan*. El resto de tipos son:

- '2': Indica que es una superficie Bezier.
- '3': Indica que es una malla (funciona como el '1').
- '4': Indica que es un billboard.

La última estructura que nos queda por ver en esta parte es la de las texturas:

```

struct tBSPTextura
{
    char strNombre[64]; // Nombre de la textura sin la extension
    int flags;           // Flags de la superficie (desconocido)
    int contenidos;      // Contenido de los flags (desconocido)
};

```

En esta estructura se almacena el nombre de la textura y otros unos *flags* no muy conocidos hasta ahora.

Bueno, ya tenemos la estructuras definidas, pasemos a ver lo que hemos codificado en nuestro tutorial. La estructura de clases es la misma de siempre. Hemos añadido al conjunto la clase **CNivelQ3** que encapsulará todo lo que expliquemos aquí. La forma de crear el objeto para el nivel es la habitual en **CJuego::InicializarJuego()** que podemos ver aquí:

```

Nivel= new CNivelQ3(m_pD3DDev);

if (FAILED(Nivel->CargarBSP(NOMBRE_NIVEL)))

```

```

{
    Util->EscribirLog("\n\t*- ¡Imposible cargar el nivel BSP!!\n");
    return E_FAIL;
}

```

Como podemos observar el constructor recibe el dispositivo 3D y la función **CargarBSP()** recibe el nombre del nivel previamente definido con un **#define** en principal.h. Esta es la cabecera de la clase **CNivelQ3**:

```

class CNivelQ3 {

private:

    LPDIRECT3DDEVICE9 m_pD3DDev;
    LPDIRECT3DVERTEXBUFFER9 m_pBufferVert;
    LPDIRECT3DTEXTURE9 *m_pTextura; // Array de texturas

    tBSPVertice *m_pVerts;
    tBSPPoli *m_pPolis;

    CBitset m_PolisPintados;

    HRESULT BuscarExtTextura(char *strArchivo);
    HRESULT CargarTextura(const char *szArchivo, int indice);
    void RenderizarPoli(int iNumDePoli);
    HRESULT CrearBufferVert(int NumVertices);
    BOOL ActualizarVertices(void);
    void CargarEstados(void);

public:

    int m_iNumDeVerts;
    int m_iNumDePolis;
    int m_iNumDeTexturas;
    int m_iVertsPintados;
    int m_iPolisPintados;

    CNivelQ3(LPDIRECT3DDEVICE9 pDispD3D);

    ~CNivelQ3(void);

    HRESULT CargarBSP(const char *strNombreArchivo);
    void RenderizarNivel(const D3DXVECTOR3 &vPos);
    void Finalizar(void);

};

```

La clase tiene un dispositivo 3D, un búfer para los vértices, un puntero que nos servirá para crear un array de texturas, otros dos punteros para crear arrays de vértices y de polígonos, y un conjunto de funciones para operar con los miembros. También va a tener un objeto de la nueva clase **CBitset**. Esta clase crea objetos que son en esencia arrays de bits, por lo que hemos creado un array de bits que lo utilizaremos para saber qué polígonos han sido ya pintados en el *frame* actual. A cada polígono del nivel le asociamos un bit de este array, si el bit está activo es que ya ha sido pintado. Pasemos a ver la función más importante, **CargarBSP()**:

```

FILE *fp = NULL;
int i = 0;

////////////////////////////////////

```

```
// Miramos si existe el archivo
if((fp = fopen(strNombreArchivo, "rb")) == NULL)
{
return E_FAIL;
}

////////////////////////////////////

// Inicializamos la cabecera y los bloques
tBSPHeader header = {0};
tBSPBloque bloques[kMaxBloques] = {0};

// Leemos la cabecera y los bloques
fread(&header, 1, sizeof(tBSPHeader), fp);

fread(&bloques, kMaxBloques, sizeof(tBSPBloque), fp);
```

Abrimos el archivo, y leemos la cabecera y los bloques (en este orden) como dijimos anteriormente...

```
tBSPTextura *textura;

// Reservamos memoria para el array de vertices
m_iNumDeVerts = bloques[kVertices].longitud / sizeof(tBSPVertice);
m_pVerts = new tBSPVertice [m_iNumDeVerts];

// Reservamos memoria para el array de poligonos
m_iNumDePolis = bloques[kPolis].longitud / sizeof(tBSPPoli);
m_pPolis = new tBSPPoli [m_iNumDePolis];

// Reservamos memoria para el array de texturas
m_iNumDeTexturas = bloques[kTexturas].longitud / sizeof(tBSPTextura);
textura = new tBSPTextura [m_iNumDeTexturas];
```

Calculamos el número de vértices, polis y texturas para luego crear su correspondiente array dinámico donde los cargaremos. El array de texturas no es necesario que sea global a la clase ya que una vez cargadas las texturas nos podremos deshacer de él.

```
// Buscamos las estructuras de los poligonos en el archivo
fseek(fp, bloques[kPolis].desplaz, SEEK_SET);

// Leemos los polis
fread(m_pPolis, m_iNumDePolis, sizeof(tBSPPoli), fp);

////////////////////////////////////

// Buscamos las estructuras de los vertices en el archivo
fseek(fp, bloques[kVertices].desplaz, SEEK_SET);

// Leemos los vertices
fread(m_pVerts, m_iNumDeVerts, sizeof(tBSPVertice), fp);

if(FAILED(CrearBufferVert(m_iNumDeVerts)))
{
return E_FAIL;
}

if(!ActualizarVertices())
{
return E_FAIL;
}
```

Lo siguiente es desplazarnos por el archivo para leer los polígonos y los vértices, y una vez cargados

podemos crear el búfer de vértices con el número de vértices del nivel para posteriormente actualizarlo. Veamos la función **ActualizarVertices()**:

```

BOOL CNivelQ3::ActualizarVertices(void)
{
    VERTICE *Vertice;

    if(FAILED(m_pBufferVert->Lock(0, 0, (VOID**)&Vertice, 0)))
    {
        return FALSE;
    }

    for (int i=0; i<m_iNumDeVerts; i++)
    {
        Vertice[i].pos.x=m_pVerts[i].vPosicion.x;
        Vertice[i].pos.y=m_pVerts[i].vPosicion.z;
        Vertice[i].pos.z=m_pVerts[i].vPosicion.y;

        Vertice[i].normal.x=m_pVerts[i].vNormal.x;
        Vertice[i].normal.y=m_pVerts[i].vNormal.z;
        Vertice[i].normal.z=m_pVerts[i].vNormal.y;

        Vertice[i].u=m_pVerts[i].vTexturaCoord.x;
        Vertice[i].v=m_pVerts[i].vTexturaCoord.y;

        Vertice[i].lu=m_pVerts[i].vLightmapCoord.x;
        Vertice[i].lv=m_pVerts[i].vLightmapCoord.y;
    }

    m_pBufferVert->Unlock();

    return TRUE;
}

```

Con esta función pasamos los vértices que hemos cargado al búfer pero con alguna peculiaridad. Hay que tener en cuenta que el Quake 3 fue creado para OpenGL y este API utiliza el eje Z a la contra que el nuestro por lo que hay que invertirlo, pero eso no es todo. Al señor Carmack le gusta tener el eje Z negativo intercambiado con el Y positivo (manías suyas), por lo que esto también hay que corregirlo. Para corregir todo este embrollo y convertirlo a DirectX hay que seguir estos pasos: Intercambiar la coordenada Y con la Z y negar esta última, con esto conseguiremos pasar del sistema del Q3 a OpenGL y finalmente hay que volver a negar la Z para pasarlo a DirectX. El resultado final es simplemente intercambiar Y con Z. He resaltado estos cambios en rojo. Las coordenadas de textura y lightmaps también se ven afectadas pero al cambiar dos veces el signo quedan como están. El tema de los lightmaps y multitextura se verá en el próximo tutorial, de momento sólo hay que denotar que en el formato de vértice empleado para el nivel manejamos dos pares de coordenadas para texturas (texturas normales y lightmaps). Sigamos con la carga del nivel, ahora les toca el turno a las texturas:

```

// Buscamos las estructuras de las texturas en el archivo
fseek(fp, bloques[kTexturas].desplaz, SEEK_SET);

// Leemos las texturas
fread(textura, m_iNumDeTexturas, sizeof(tBSPTextura), fp);

// Ya tenemos los datos de las texturas, ahora las cargamos:

// Creamos el array y lo inicializamos a null

```



```

if (m_iNumDeTexturas)
    m_pTextura = new LPDIRECT3DTEXTURE9 [m_iNumDeTexturas];

for (i=0;i<m_iNumDeTexturas;i++)
{
    m_pTextura[i]=NULL;
}

// Pasamos por todas las texturas
for(i = 0; i < m_iNumDeTexturas; i++)
{
    // Busca la extension y la añade al nombre
    if (SUCCEEDED(BuscarExtTextura(textura[i].strNombre)))
    {
        // crea la textura a partir del archivo
        CargarTextura(textura[i].strNombre,i);
    }
}

// podemos borrar el array temporal de estructuras de textura
BorrarObj(textura);

```

Procedemos igual que con las otras estructuras, nos posicionamos en el archivo, las leemos y en este caso tendremos que crear el array de **LPDIRECT3DTEXTURE8** para cargar las texturas en un formato que pueda leer Direct3D. Quake 3 guarda el nombre de las texturas en los niveles sin la extensión por lo que nos toca a nosotros añadirlo. Esta extensión puede ser .tga o .jpg dependiendo del caso. Para saber que extensión es la que tiene realmente la textura probaremos a ver si existe primero la textura en formato jpg y sino en tga. Esto lo haremos con la función **BuscarExtTextura()**. En caso de no encontrar ningún archivo esto quiere decir que o bien la textura no existe, o es una textura interna del juego (no esta en directorios) o se trata de un shader. Un shader es otro tipo de archivos que maneja información sobre texturas; aquí no veremos shaders por ser un tema mucho más avanzado del Quake 3. Para cargar finalmente la textura llamaremos a **CargarTextura()** y le pasaremos el nombre definitivo con extensión.

Ya hemos terminado de cargar el nivel, sólo nos queda cerrar el archivo y crear el array de bits con el número de polígonos que tiene el nivel:

```

// Creamos la tabla de bits de poligonos pintados
m_PolisPintados.Inicializar(m_iNumDePolis);

// Cerramos el archivo
fclose(fp);

return S_OK;

```

Pues bien, ya tenemos el nivel en memoria, tanto el búfer de vértices como las texturas y los arrays con los vértices y los polis. ahora nos queda renderizarlo en cada frame. Esto es fácil, veamos la función **RenderizarNivel()**:

```

void CNivelQ3::RenderizarNivel(const D3DXVECTOR3 &vPos)
{
    m_PolisPintados.DesactivarTodos();

    m_iVertsPintados=0;
    m_iPolisPintados=0;

    //Ponemos la matriz identidad en WORLD por si algo la habia cambiado
    D3DXMATRIX Tras;
    D3DXMatrixIdentity(&Tras);

```

```

m_pD3DDev->SetTransform(D3DTS_WORLD, &Tras);

// elegimos nuestro tipo de vértice y cargamos estados
m_pD3DDev->SetStreamSource(0, m_pBufferVert, 0, sizeof(VERTICE));
m_pD3DDev->SetFVF(D3DFVF_VERTICE);

CargarEstados();

// recorremos todos los polis y los pintamos uno a uno
// si son poligonos solidos normales y no se han pintado ya
for (int i=0; i<m_iNumDePolis;i++)
{
    // Aseguramos que es un poli normal
    if(m_pPolis[i].tipo != POLIGONO_SOLIDO) continue;

    // Si el poli no se ha dibujado ya...
    if(!m_PolisPintados.EstaActivado(i))
    {
        // Pintamos el poli y lo activamos como pintado
        RenderizarPoli(i);
        m_PolisPintados.Activar(i);
    }
}
}

```

La función recibe un vector **vPos** que será la posición de la cámara pero esto no lo usaremos hasta la tercera parte de este tutorial. Lo primero que hacemos es desactivar todos los bits del array de polis pintados e inicializamos unos contadores para sacar luego información por pantalla del número de vértices y polis pintados en cada *frame*. Cargamos la matriz identidad y los estados necesarios y acto seguido nos disponemos a renderizar COMPLETAMENTE todos los polígonos del nivel. Los recorremos todos con un bucle y si se trata de un poli normal y no ha sido pintado anteriormente lo renderizamos con la función **RenderizarPoli()**. Justo después activamos este poli para advertir de que ya ha sido pintado. Veamos por fin la función **RenderizarPoli()**:

```

void CNivelQ3::RenderizarPoli(int iNumDePoli)
{
    //creamos esto para manejarnos mejor
    //dentro del array de polis cogemos el que tenga nuestro indice iNumDePoli
    tBSPoli *pPoli = &m_pPolis[iNumDePoli];

    //si existe el array de texturas ponemos la que le corresponde
    if (m_pTextura != NULL)
    {
        m_pD3DDev->SetTexture(0, m_pTextura[pPoli->texturaID]);
    }

    // lo pintamos
    m_pD3DDev->DrawPrimitive(D3DPT_TRIANGLEFAN, pPoli->startVertIndex, pPoli->numDeVerts-2);

    // acutlizamos contadores
    m_iVertsPintados += pPoli->numDeVerts;
    m_iPolisPintados ++;
}

```

Seleccionamos la textura indicada por el polígono cuyo índice es el que recibe la función y después lo pintamos. Como dijimos antes los polígonos son *triangle fans* así que es es justamente lo que tendremos que pintar. Para pintarlo tendremos que pasarle su vértice inicial y el número de triángulos que tiene el

*fan*. Este número de triángulos siempre será el número de vértices del *fan* menos 2. Para finalizar actualizamos los contadores de información.

Pues hasta aquí la primera parte. En la siguiente veremos cómo usar los lightmaps del Quake 3 para crear iluminación estática muy realista.

## Tutorial 9 - Cargando niveles del Quake 3 (Parte 2)

En esta segunda parte del tutorial veremos cómo cargar los lightmaps del nivel para simular una iluminación estática muy realista. Pero antes vamos a describir lo que es un lightmap. Un lightmap es básicamente una textura que se aplica a la textura original del nivel. Esta nueva textura contiene información de la iluminación en esa zona del nivel. Por medio del multitexturing se fusionan las dos texturas para simular esta iluminación. Aquí hay un pequeño artículo donde se puede ver un claro ejemplo de ello: [http://www.flipcode.com/articles/article\\_lightmaps.shtml](http://www.flipcode.com/articles/article_lightmaps.shtml)

Pasemos al código: Al igual que con cualquier otro bloque de datos en el archivo del nivel necesitaremos una estructura para definir el bloque de los lightmaps.

```
struct tBSPLightmap
{
    byte mapaDeBits[128][128][3]; // Datos RGB en una imagen de 128x128
};
```

Esta es la estructura que usaremos. Como se puede observar cada lightmap es un mapa de bits de 128x128 y 24 bits color, 8 para cada tono RGB. Veamos que ha cambiado en la función **CNivelQ3::CargarBSP()**...

```
.
.
.
// Reservamos memoria para el array de lightmaps
m_iNumDeLightmaps = bloques[kLightmaps].longitud / sizeof(tBSPLightmap);
lightmap = new tBSPLightmap [m_iNumDeLightmaps];
.
.
.
// Creamos el array y lo inicializamos
if (m_iNumDeLightmaps)
m_pLightmap = new LPDIRECT3DTEXTURE9 [m_iNumDeLightmaps];
for (i=0;i<m_iNumDeLightmaps;i++)
{
    m_pLightmap[i]=NULL;
}

// Buscamos las estructuras de los lightmaps en el archivo
fseek(fp, bloques[kLightmaps].desplaz, SEEK_SET);

// Los recooremos todos y los leemos
for(i = 0; i < m_iNumDeLightmaps; i++)
{
    // Leemos los valores rgb para cada lightmap
    fread(&lightmap[i], 1, sizeof(tBSPLightmap), fp);

    // Creamos un lightmap de 128x128 por cada estructura q haya
    CrearTexturaLightmap(i,lightmap[i].mapaDeBits, 128, 128);
}

// podemos borrar el array temporal de estructuras de lightmaps
BorrarObj(lightmap);
.
.
.
```

Como es habitual con cualquier otro bloque lo primero que hacemos es reservar memoria para el array de lightmaps y justo después de haber cargado las texturas nos encontramos con el código que carga los

lightmaps. Lo primero que debemos hacer es comprobar que hay lightmaps en el nivel y después crear un array de texturas con el mismo número que lightmaps existan para que podamos albergarlos. Nuestro objetivo es pasar los datos de los lightmaps del nivel a texturas en memoria. Para ello nos posicionamos en el archivo y uno a uno vamos leyendo todos los lightmaps. Una vez tengamos la estructura leída del archivo se la pasamos a **CrearTexturaLightmap()** que transformará estos datos en una textura en memoria. Veamos cómo trabaja esta función:

```
HRESULT CNivelQ3::CrearTexturaLightmap(int indice, byte pMapaDeBits[128][128][3],
int ancho, int alto)
{

D3DXCreateTexture(m_pD3DDev, 128, 128, 0, 0, D3DFMT_R5G6B5, D3DPOOL_MANAGED, &m_pLightmap[
indice]);

IDirect3DSurface9 *TextureSurface;

m_pLightmap[indice]->GetSurfaceLevel(0, &TextureSurface);

D3DLOCKED_RECT pLockedRect;
RECT rectangleToLock;
rectangleToLock.top = 0;
rectangleToLock.left = 0;
rectangleToLock.bottom = 128;
rectangleToLock.right = 128;

CambiarGamma((unsigned char *)pMapaDeBits, ancho*alto*3, (float)GFX_OPC_GAMMA);

if(FAILED(TextureSurface->LockRect(&pLockedRect, NULL, 0 )))
    PostQuitMessage(0);

USHORT* pData = (USHORT*)pLockedRect.pBits;

for(int p = 0; p<128;p++)
{
    for(int q = 0; q<128; q++)
    {
        WORD r = pMapaDeBits[q][p][0]>>3;
        WORD g = pMapaDeBits[q][p][1]>>2;
        WORD b = pMapaDeBits[q][p][2]>>3;
        WORD lightmap_pixel_565 =RGB_16BIT565 (r,g,b);
        pData[(q * (pLockedRect.Pitch/2)) + p] = lightmap_pixel_565;
    }
}

if(FAILED(TextureSurface->UnlockRect()))
    PostQuitMessage(0);

TextureSurface->Release();

return S_OK;
}
```

Como se puede ver le pasamos el índice, el array de información y el tamaño que finalmente tendrá en memoria. Lo primero que hace esta función es crear una textura y asignar una superficie a ella (esto se hace para poder modificar dicha textura a través de la superficie). Llamamos a **CambiarGamma()** para ajustar el brillo a nuestro gusto (lo explicaremos a continuación). Finalmente se bloquea la superficie para que, pixel a pixel, traduzcamos la información que había en el array a la superficie adquirida (que

finalmente será la textura que usaremos). He decidido crear las texturas de los lightmaps en 16 bits de color ya que apenas se nota la diferencia y se gana mucho en velocidad y memoria. Por ello al pasar del array a la superficie se necesitan todas esas operaciones de bits para hacer el cambio. Con eso creamos el array de texturas que albergará los lightmaps del nivel. Ya sólo queda renderizarlos a la vez que las texturas normales. Esto es muy fácil, veamos...

```
void CNivelQ3::RenderizarPoli(int iNumDePoli)
{
    //creamos esto para manejarnos mejor
    //dentro del array de polis cogemos el que tenga nuestro indice iNumDePoli
    tBSPoli *pPoli = &m_pPolis[iNumDePoli];

    //si existe el array de texturas ponemos la que le corresponde
    if (m_pTextura != NULL)
    {
        m_pD3DDev->SetTexture(0, m_pTextura[pPoli->texturaID]);
    }

    //si existe el array de lightmaps ponemos el que le corresponde
    if (m_pLightmap != NULL)
    {
        m_pD3DDev->SetTexture(1, m_pLightmap[pPoli->lightmapID]);
    }

    // lo pintamos
    m_pD3DDev->DrawPrimitive(D3DPT_TRIANGLEFAN, pPoli->startVertIndex, pPoli->numDeVerts-2);

    // acutlizamos contadores
    m_iVertsPintados += pPoli->numDeVerts;
    m_iPolisPintados ++;
}
```

De la misma manera que seleccionamos la textura que debemos poner a un polígono también seleccionamos su lightmap pero en vez de ponerle el texture stage 0 le pondremos el 1. Esto sirve para que pinte las dos texturas a la vez. Existen hasta 8 texture stages (en algunas tarjetas gráficas sólo 2 y en otras 16!). Pero no todo es pintar a la vez, debemos decirle a Direct3D cómo debe ser esta fusión entre texturas a través de los estados:

```
m_pD3DDev->SetTextureStageState(1,D3DTSS_COLOROP, D3DTOP_MODULATE);
```

Que quiere decir que module (multiplique) la textura del stage anterior (el 0 claro) con la actual (el 1) con lo que conseguimos ese efecto de ensombrecer la textura original. Para ver sólo las texturas de los lightmaps sería curioso desactivar el stage 0, de esta manera sólo se verían las luces en sí, pero eso lo dejo a vuestra merced... Por último veamos la función **CambiarGamma()** que comentamos antes:

```
void CNivelQ3::CambiarGamma(byte *pImagen, int tamanno, float factor)
{
    for(int i = 0; i < tamanno / 3; i++, pImagen += 3)
    {
        float scale = 1.0f, temp = 0.0f;
        float r = 0, g = 0, b = 0;

        r = (float)pImagen[0];
        g = (float)pImagen[1];
        b = (float)pImagen[2];

        r = r * factor / 255.0f;
```

```
g = g * factor / 255.0f;
b = b * factor / 255.0f;

if(r > 1.0f && (temp = (1.0f/r)) < scale) scale=temp;
if(g > 1.0f && (temp = (1.0f/g)) < scale) scale=temp;
if(b > 1.0f && (temp = (1.0f/b)) < scale) scale=temp;

scale*=255.0f;
r*=scale; g*=scale; b*=scale;

pImagen[0] = (byte)r;
pImagen[1] = (byte)g;
pImagen[2] = (byte)b;
}
}
```

Es un pelín liante pero si miras atentamente lo único que hace es cambiar los valores de color por un factor de escala que previamente le pasamos. Según sea ese factor, los lightmaps quedarán más brillantes u oscuros. Esto nos brinda la posibilidad de jugar con distintas configuraciones y es una buena opción para poder cambiar el "brillo" en los niveles.