Programación de juegos con OpenGL

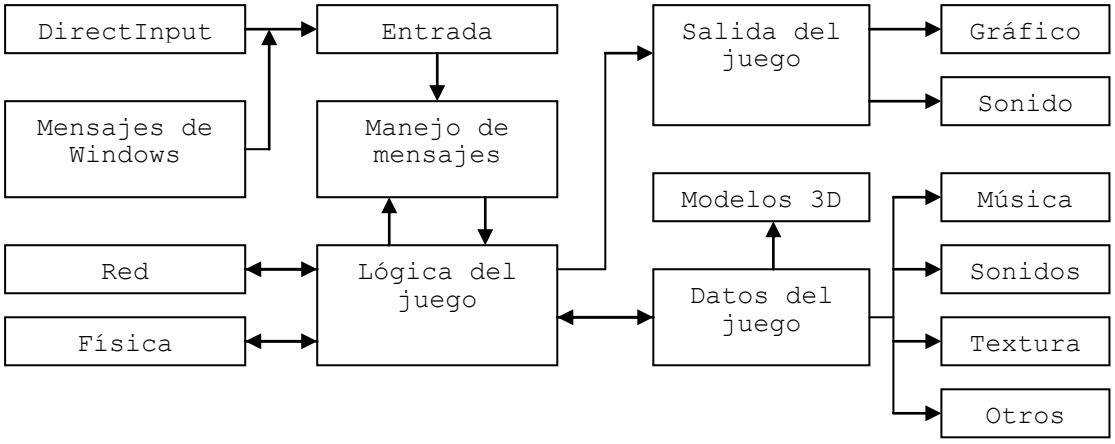
Indice

Parte I: Introducción a OpenGL y DirectX.

Capítulo 1: Empezando a explorar: OpenGL y DirectX.

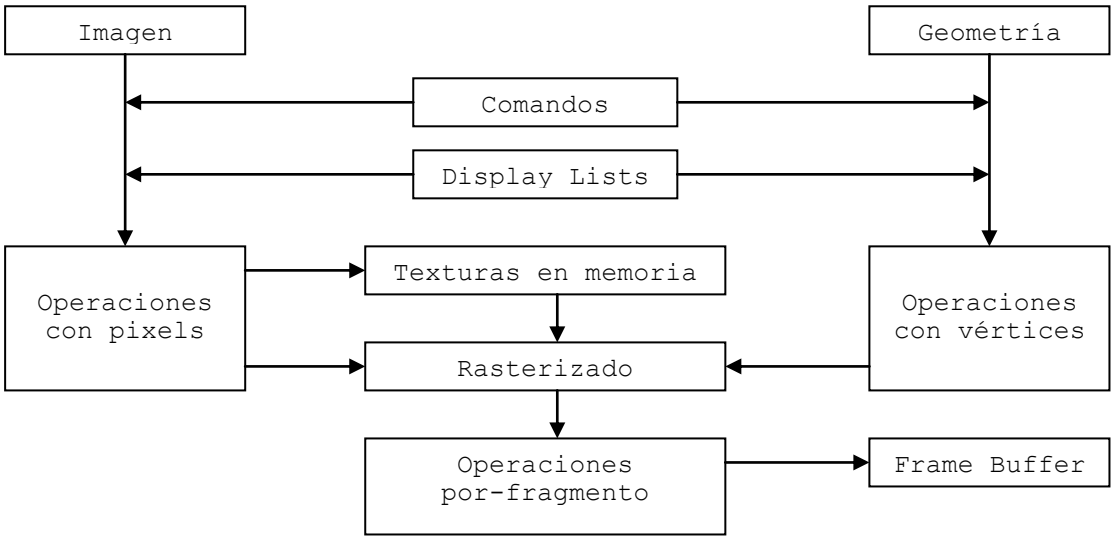
1) Los elementos de un juego:

Un juego a día de hoy es una aplicación compuesta de múltiples elementos, como los del siguiente diagrama:



2) OpenGL:

OpenGL es una librería para renderizado 3D multiplataforma, que se puede emplear para representar la "escena" de nuestro juego. Es importante saber un poco de la arquitectura interna de OpenGL, por ejemplo que se trata de una máquina de estados, que dependiendo de la configuración que le demos, renderizará de una forma u otra. Hay que tener en cuenta, que OpenGL es una librería de bajo nivel, al contrario que un motor, como el del Quake. La pipeline de OpenGL sigue el siguiente esquema:



3) DirectX:

Para fomentar la creación de juegos bajo windows, se creó DirectX, para facilitar el manejo de temas multimedia, a todos esos programadores ya enraizados en MS-DOS. Esta librería ofrece soporte para renderizado 3D, manejo de la entrada por teclado, ratón y joysticks, manejo del sonido, manejo de videos, manejo de red, y alguna cosilla más.

Capítulo 2: Usando Ventanas con OpenGL.

1) La función WinMain():

```
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nShowCmd)
{
    MessageBox(NULL, "¡Hola mundo!", "Hola", NULL);
}
```

Esta es la función de entrada de todo programa en windows, incluido Windows XP. Los parámetros que recibe son:

+ **hInstance**: Es el identificador de la aplicación, que se emplea para interactuar con la api del windows.
+ **hPrevInstance**: Vale NULL, ya que es un parámetro mantenido para garantizar la retrocompatibilidad con windows 3.1, y ya no se usa.
+ **lpCmdLine**: Es una cadena que contiene los argumentos con los que fue llamada la aplicación ("a.exe jose pepe juan" -> "jose pepe juan").
+ **nShowCmd**: Indica de que modo será mostrada la aplicación al ejecutarse.

2) El manejador de eventos:

Windows se comunica con las aplicaciones mediante eventos, el usuario realiza acciones, windows las captura y las manda como mensajes al programa. Por ello tendremos que crear una función, tal que sirva para comunicarse con windows:

```
LRESULT CALLBACK MiManejador (HWND hWnd, UINT message,
                              WPARAM wParam, LPARAM lParam)
{
    switch(message)
    {
        case WM_CREATE:
            return 0;

        case WM_CLOSE:
            PostQuitMessage(0); //Mandar un mensaje de salir.
            return 0;

        //...
    }
    return DefWindowProc(hWnd, message, wParam, lParam);
}
```

3) La "clase" ventana:

Para crear una ventana hay que indicar bastantes parámetros y por ello la api del windows tiene la siguiente estructura:

```
typedef struct _WNDCLASSEX {
    UINT    cbSize;        //= sizeof(WNDCLASSEX);
    UINT    style;         //Estilo de la ventana.
    WNDPROC lpfnWndProc;   //Puntero al manejador de eventos.
    int     cbClsExtra;    //= 0; Información extra de la clase.
    int     cbWndExtra;    //= 0; Información extra de la ventana.
    HANDLE  hInstance;     //Identificador de la aplicación.
    HICON   hIcon;         //Identificador del icono de la ventana.
    HCURSOR hCursor;       //Identificador del cursor del ratón.
    HBRUSH  hbrBackground; //Color de fondo de la ventana.
    LPCTSTR lpszMenuName;  //Nombre del menú de la ventana.
    LPCTSTR lpszClassName; //Nombre de la "clase".
    HICON   hIconSmall;    //Identificador del icono pequeño.
} WNDCLASSEX;
```

El estilo de la ventana y los demás campos se suelen rellenar, por ejemplo de esta forma:

```
WNDCLASSEX wc;

wc.cbSize        = sizeof(WNDCLASSEX);
wc.style         = CS_HREDRAW | CS_VREDRAW;
wc.lpfnWndProc   = (WNDPROC) MiManejador;
wc.cbClsExtra    = 0;
wc.cbWndExtra    = 0;
wc.hInstance     = hInstance;
wc.hIcon         = LoadIcon(NULL, IDI_WINLOGO);
wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
wc.hbrBackground = NULL;
wc.lpszMenuName  = NULL;
wc.lpszClassName = "Mi juego";
wc.hIconSmall    = NULL;

RegisterClassEx(&wc); //Si devuelve 0, ha ocurrido un error.
```

La última llamada registra a la actual aplicación la "clase" de ventana que acabamos de configurar.

4) Creando la ventana:

Tras registrar la clase, tenemos que crear la ventana con la función:

```
HWND CreateWindowEx (
   DWORD dwExStyle,      //= NULL; Estilo extendido.
   LPCTSTR lpClassName,  //Nombre de la clase.
   LPCTSTR lpWindowName, //Título de la ventana.
   DWORD dwStyle,        //Estilo de la ventana (*).
   int x, int y,         //Coordenadas de la ventana.
   int nWidth,           //Ancho de la ventana.
   int nHeight,          //Alto de la ventana.
   HWND hWndParent,      //= NULL; Ventana padre.
   HMENU hMenu,          //= NULL; Menú de la ventana.
   HINSTANCE hInstance,  //= hInstance;
   LPVOID lpParam);      //= NULL;

(*) = WS_CAPTION | WS_MINIMIZEBOX | WS_SYSMENU | WS_VISIBLE;
```

Entre los estilos que podemos elegir para la ventana, en las páginas 34-35 se encuentra la lista completa. Esta función devuelve el manejador o identificador de la ventana. Si nos devolviera NULL, es que no se ha podido crear la ventana. Una vez creada con exito podemos llamar a:

```
ShowWindow(hWnd, nCmdShow);
```

5) El bucle de mensajes:

Las aplicaciones de windows van por eventos y para poder recibirlos y tratarlos tenemos el siguiente ejemplo:

```
while(!done)
{
    if(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
    {
        if(msg.message == WM_QUIT)
        {
            done = true;
        }
        else
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
}

return msg.wParam;
```

De este ejemplo sacamos que a **PeekMessage** tenemos que pasarle la dirección de una estructura **MSG**, donde se recoge la información de los mensajes de windows, para que dicha función nos de el último mensaje de la cola de eventos. Devuelve true siempre que haya logrado recoger un evento de la cola. **TranslateMessage** y **DispatchMessage** sirve la primera para traducir el mensaje para la aplicación y el segundo para enviarlo al manejador de eventos.

6) Ejemplo completo de aplicación windows:

(Páginas 38-43).

7) Introducción al WGL:

Para poder crear un contexto en el que OpenGL pueda pintar su buffer, windows tiene unas funciones:

```
HGLRC wglCreateContext (HDC hDC);
BOOL  wglDeleteContext (HGLRC hRC);
BOOL  wglMakeCurrent   (HDC hDC, HGLRC hRC);
BOOL  SwapBuffers      (HDC hDC);
```

La primera crea el contexto, y para obtener el HDC de la aplicación basta con llamar a GetDC(hWnd). La segunda borra el contexto. La tercera lo activa. Y la cuarta sirve para actualizar la pantalla, cuando se está usando doble buffer. Todo esto suele usarse del siguiente modo:

```
LRESULT CALLBACK MiManejador (HWND hWnd, UINT message, WPARAM wParam,
                              LPARAM lParam)
{
    static HGLRC hRC;
    static HDC hDC;

    switch(message)
    {
        case WM_CREATE:
            hDC = GetDC(hWnd);
            hRC = wglCreateContext(hDC);
            wglMakeCurrent(hDC, hRC);
            return 0;

        case WM_DESTROY:
            wglMakeCurrent(hDC, NULL);
            wglDeleteContext(hRC);
            PostQuitMessage(0);
            return 0;

        //...
    }

    //...
}
```

8) El formato del pixel:

Antes de llamar a wglCreateContext() es recomendable cambiar el formato del pixel a nuestras necesidades con la estructura:

```
typedef struct _PIXELFORMATDESCRIPTOR
{
    WORD  nSize;           //= sizeof(PIXELFORMATDESCRIPTOR);
    WORD  nVersion;        //= 1;
    DWORD dwFlags;         //= PFD_DRAW_TO_WINDOW |
                           //  PFD_SUPPORT_OPENGL | PFD_DOUBLEBUFFER;
    BYTE  iPixelType;      //= PFD_TYPE_RGBA;
    BYTE  cColorBits;      //= 32; (32, 24, 16, 8).
    BYTE  cRedBits;        //= 0;
    BYTE  cRedShift;       //= 0;
    BYTE  cGreenBits;      //= 0;
    BYTE  cGreenShift;     //= 0;
    BYTE  cBlueBits;       //= 0;
    BYTE  cBlueShift;      //= 0;
    BYTE  cAlphaBits;      //= 0;
    BYTE  cAlphaShift;     //= 0;
    BYTE  cAccumBits;      //= 0; Tamaño del buffer de acumulación.
    BYTE  cAccumRedBits;   //= 0;
    BYTE  cAccumGreenBits; //= 0;
    BYTE  cAccumRedBits;   //= 0;
    BYTE  cAccumAlphaBits; //= 0;
    BYTE  cDepthBits;      //= 16; (32, 16) Buffer de profundidad.
```

```
    BYTE  cStencilBits;    //= 0; Tamaño del buffer de stencil.
    BYTE  cAuxBuffers;     //= 0;
    BYTE  iLayerType;      //= PFD_MAIN_PLANE;
    BYTE  bReserved;       //= 0;
    BYTE  dwLayerMask;     //= 0;
    BYTE  dwVisibleMask;   //= 0;
    BYTE  dwDamageMask;    //= 0;
} PIXELFORMATDESCRIPTOR;
```

Configurado el tipo de pixel que necesitemos, tendremos que establecerlo como el tipo actualmente activado:

```
int nPixelFormat;
PIXELFORMATDESCRIPTOR pfd = {...};
nPixelFormat = ChoosePixelFormat(hDC, &pfd);
SetPixelFormat(hDC, nPixelFormat, &pfd);
```

9) Aplicación OpenGL de ejemplo:

(Páginas 49-58).

10) Modo pantalla completa:

Para poder ejecutar la aplicación en modo pantalla completa, tendremos que configurar un par de cosillas:

```
DEVMODE devModeScreen;

memset(&devModeScreen, 0, sizeof(devModeScreen));
devModeScreen.dmSize       = sizeof(devModeScreen);
devModeScreen.dmPelsWidth  = screenWidth;
devModeScreen.dmPelsHeight = screenHeight;
devModeScreen.dmBitsPerPel = screenBpp;
devModeScreen.dmFields     = DM_PELSWIDTH | DM_PELSHEIGHT |
                             DM_BITSPERPEL;

if(ChangeDisplaySettings(&devModeScreen, CDS_FULLSCREEN) !=
   DISP_CHANGE_SUCCESSFUL)
    fullscreen = false;

if(fullScreen)
{
    winStyle = WS_POPUP;
    ShowCursor(FALSE);
}
else
{
    winStyle = WS_CAPTION | WS_MINIMIZEBOX | WS_SYSMENU | WS_VISIBLE;
}

RECT rect;

rect.top    = 0;
rect.left   = 0;
rect.bottom = screenHeight;
rect.right  = screenWidth;
```

```
AdjustWindowRectEx(&rect, winStyle, FALSE, extendedWinStyle);
```

Capítulo 3: Un vistazo a la teoría de los gráficos 3D.

1) Introducción:

Hay unos cuantos conceptos importantes en geometría:

+ La escala: Que sirve como magnitud con la que se mide la distancia en el espacio geométrico.
+ Los puntos: Indican una posición en el espacio. Para uno de 3 dimensiones estaría compuesto por tres valores escalares.
+ Los vectores: Corresponde a una dirección en el espacio, que tiene una magnitud asociada, representada como una linea. Da igual si empiezan en diferentes puntos del espacio, si tienen la misma dirección y tamaño, ambos vectores serán iguales.

2) Operaciones con vectores:

a) La magnitud de un vector:

La magnitud equivale al tamaño del mismo y se calcula con:

$$|A| = sqrt(A.x^2 + A.y^2 + A.z^2)$$

Normalmente esto sirve para normalizar un vector.

b) Normalizar un vector:

Esto consiste en reducir cada componente a un valor entre 0.0 y 1.0, que suele ser usado en calculos varios de geometría:

$$n = N / |N| = (N.x / |N|, N.y / |N|, N.z / |N|)$$

c) Suma de vectores:

$$A + B = (A.x + B.x, A.y + B.y, A.z + B.z)$$

d) El producto con un escalar:

$$s * A = (s * A.x, s * A.y, s * A.z)$$

e) El producto escalar: (Dot product)

Esta operación suele ser usada para calcular el ángulo entre dos vectores, y hay dos formas de calcularl este producto:

$$A * B = A.x * B.x + A.y * B.y + A.z * B.z$$
$$A * B = |A| * |B| * \cos \alpha$$

Con la segunda fórmula nos queda:

α = arccos((A * B) / (|A| * |B|))

f) El producto vectorial: (Cross product)

Usado con bastante frecuencia en temas como detección de colisiones, iluminación, física, etcétera, esta operación se realiza:

A x B = |A| * |B| * sin α * n

Donde n es el vector normalizado. Pero como de este modo la verdad es que es un poco horrible el cálculo, también se puede hacer:

A x B = (A.y * B.z - A.z * B.y, A.z * B.x - A.x * B.z, A.x * B.y - A.y * B.x)

R.x = A.y * B.z - A.z * B.y
R.y = A.z * B.x - A.x * B.z
R.z = A.x * B.y - A.y * B.x

g) Nota final sobre los vectores:

A pesar de como son representados los vectores en el espacio, su representación en datos es de un punto (x, y, z), cya magnitud dibujada correspondería a la linea desde el origen de coordenadas hasta el punto. Pero en realidad los vectores no son lineas, ni cosas representables de forma visible en el mundo, solo hacen referencia a valores con una dirección como por ejemplo la velocidad, la fuerza, etcétera.

Otro punto curioso en OpenGL es la forma de diferenciar un punto de un vector, ya que hay una cuarta dimensión para ese propósito, la w. Si w vale 1 se trata de un punto 3D en el espacio, sin embargo de valer 0 se tratará de un vector.

3) Matrices:

Una matriz es una tabla de m filas por n columnas, en la que cada posición a(i, j) corresponde a un valor numérico:

```
             (a1.1 a1.2 ... a1.n)
A (m * n) = (a2.1 a2.2 ... a2.n)
             (a3.1 a3.2 ... a3.n)
             (...  ...  ... ... )
             (am.1 am.2 ... am.n)
```

a) La matriz identidad:

Esta matriz es todo ceros salvo en la diagonal, y su propiedad más interesante es que es el elemento neutro de la multiplicación: I * M = M.

```
    (1 0 0 0)
I = (0 1 0 0)
    (0 0 1 0)
    (0 0 0 1)
```

b) Suma y resta de matrices:

```
A + B = (a b) + (e f) = (a + e  b + f)
        (c d)   (g h)   (c + g  d + h)

A - B = (a b) - (e f) = (a - e  b - f)
        (c d)   (g h)   (c - g  d - h)
```

c) Producto con matrices:

```
A * k = (a b) * k = (a * k  b * k)
        (c d)       (c * k  d * k)

A * B = (a b) * (e f) = (a*e+b*g  a*f+b*h)
        (c d)   (g h)   (c*e+d*g  c*f+d*h)

A * B = C(i, j) = Sum(x = 1..n; A(i, x) * B(x, j))
```

4) Transformaciones:

Para mover, rotar y redimensionar objetos 3D hemos de multiplicar cada punto por matrices especialmente preparadas.

a) Traslaciones:

```
        (1 0 0 dx)   (x)   (x + dx)
T * p = (0 1 0 dy) * (y) = (y + dy)
        (0 0 1 dz)   (z)   (z + dz)
        (0 0 0 1 )   (1)   (  1   )
```

b) Rotaciones:

```
         (1   0      0   0) (x)   (        x        )
RX * p = (0 cos α -sin α 0)*(y) = (y*cos α - z*sin α)
         (0 sin α  cos α 0) (z)   (y*sin α - z*cos α)
         (0   0      0   1) (1)   (        1        )

         ( cos α 0 sin α 0)   (x)   (x*cos α + z*sin α)
RY * p = (   0   1   0   0) * (y) = (        y        )
         (-sin α 0 cos α 0)   (z)   (z*cos α - x*sin α)
         (   0   0   0   1)   (1)   (        1        )

         (cos α -sin α 0 0)   (x)   (x*cos α - y*sin α)
RZ * p = (sin α  cos α 0 0) * (y) = (y*sin α + y*cos α)
         (  0      0   1 0)   (z)   (        z        )
         (  0      0   0 1)   (1)   (        1        )
```

c) Escalado:

```
        (sx 0  0  0)   (x)   (x * dx)
S * p = (0  sy 0  0) * (y) = (y * dy)
        (0  0  sz 0)   (z)   (z * dz)
```

```
(0  0  0  1)   (1)   (  1   )
```

5) Proyecciones:

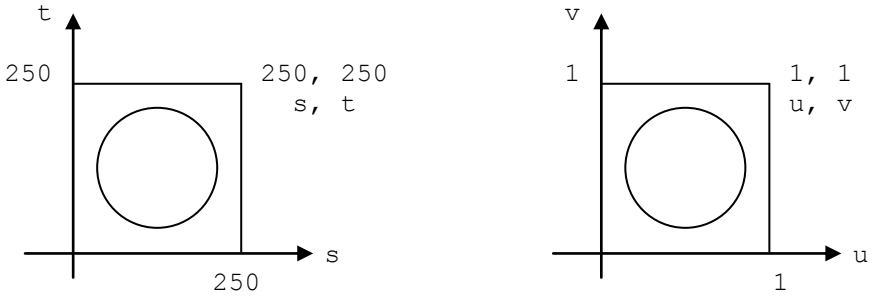
Hay dos tipos de proyecciones:
+ Paralela u ortográfica: Usada en herramientas CAD.
+ Perspectiva o cónica: Se usa en juegos 3D o herramientas de modelado. Cuanto más cerca esté el plano de focalización más grande será el ángulo del FOV (Field Of View). Además para evitar errores de cálculo, hemos de crear un volumen de vista, para realizar el clipping, es decir, lo que está fuera de ese espacio no será renderizado.

6) Iluminación:

Hay tres tipos de luces:
+ Luz ambiental: Es una luz que no viene de ninguna dirección en particular, por lo que ilumina toda la escena con la misma intensidad.
+ Luz difusa: Es una luz que parte de un punto y tiene dirección. Además es reflejada en las superficies de forma suave, como una luz fluorescente.
+ Luz especular: Como la anterior tiene un punto de emisión, y una dirección, sin embargo esta es reflejada con más intensidad, hasta el punto de crear puntos brillantes sobre una superficie, como lo haría un rayo de luz solar.

7) Mapeado de texturas:

Las figuras geométricas del mundo 3D suelen llevar texturas para dar color a cada pixel de esta. Estas tienen dos sistemas de coordenadas:
+ De la textura (s, t): Que sirve para indicar la posición de cada pixel dentro de la propia textura en cuestión.
+ Paramétricas (u, v): Que sirve para indicar la escala sobre el polígono que es pintada la textura.



Parte II: Usando OpenGL.

Capítulo 4: Los estados y primitivas de OpenGL.

1) Funciones de estados:

Ya que OpenGL es una máquina de estados existen funciones para obtener y modificar los parámetros que la hacen funcionar de un modo u otro. Para obtener datos referentes a un estado se usan las funciones:

```
void glGetBooleanv (GLenum pname, GLboolean * params);
void glGetDoublev  (GLenum pname, GLdouble  * params);
void glGetFloatv   (GLenum pname, GLfloat   * params);
void glGetIntegerv (GLenum pname, GLint     * params);
```

En el libro entre las páginas 93 y 98 hay una lista de los parámetros de la máquina de estados de OpenGL. Además hay algunos de estos que lo único que se puede hacer con ellos es activarlos o desactivarlos, para ellos existen:

```
GLboolean glIsEnable (GLenum pname); //GL_TRUE o GL_FALSE.
void      glEnable   (GLenum pname);
void      glDisable  (GLenum pname);
```

Así que glEnable activa el parámetro en la máquina de estados, glDisable lo desactiva, y glIsEnable te indica si está o no activado. Para los otros parámetros que tienen subparámetros para ser configurados, son modificados con funciones específicas y no un conjunto de glSets.

2) Manejando primitivas:

OpenGL para pintar nos permite usar puntos, lineas, triángulos, cuadrilateros y polígonos. Aunque lo más usado son los triángulos, ya que al definir una forma en pantalla sus "atomos" deben de estar en el mismo plano, para evitar errores en el renderizado, y una de las propiedades de los triángulos es que con tres puntos se define un plano. Para definir cualquier forma tendremos que ir indicando los vértices de esta, y para poder hacerlo en OpenGL tendremos que usar bloques de funciones:

```
void glBegin (GLenum mode);
void glEnd   (void);
```

Con glBegin indicamos a la máquina de estados que empezaremos a mandar vértices, y con glEnd que ya hemos terminado el bloque actual de vértices. El argumento **mode** de glBegin sirve para indicar el tipo de primitiva:

+ GL_POINTS = Puntos sueltos.
+ GL_LINES = Lineas sueltas.
+ GL_LINE_STRIP = Lineas encadenadas.
+ GL_LINE_LOOP = Lineas encadenadas, que formarán un circuito cerrado.
+ GL_TRIANGLES = Triángulos sueltos.
+ GL_TRIANGLE_STRIP = Triángulos encadenados para formar una tira.
+ GL_TRIANGLE_FAN = Triángulos encadenados para formar un círculo.
+ GL_QUADS = Cuadriláteros sueltos.
+ GL_QUAD_STRIP = Cuadriláteros encadenados para formar una tira.

+ GL_POLYGON = Polígonos de múltiples vértices.

Es importante saber que no se puede meter un bloque glBegin-glEnd, dentro de uno ya abierto. Además hay un repertorio específico de funciones de OpenGL que se pueden llamar dentro de un bloque de estos: `glVertex`, `glColor`, `glIndex`, `glNormal`, `glTexCoord`, `glEvalCoord`, `glEvalPoint`, `glMaterial`, `glEdgeFlag`, `glCallList` y `glCallLists`. Llamando a otras funciones se producirían errores en la máquina de estados interna de OpenGL. Y de todas las funciones de la lista la más básica es:

```
void glVertex[2,3,4][d,f,i,s][v] (...);
```

La función glVertex sirve para indicar los vértices que forman la figura que queremos pintar en pantalla.

a) Dibujando puntos:

Dentro de un bloque glBegin-glEnd podemos pintar tantos puntos como veces llamemos a glVertex. En general, con las otras formas de pintado también podemos poner tantas llamadas como queramos, siempre que se respeten una serie de reglas implicitas a la primitiva en cuestión.

+ Cambiando el tamaño de los puntos:

Por defecto el tamaño de los puntos es de 1.0, pero existe una función para modificar este aspecto:

```
void glPointSize (GLfloat size);
```

Pero volviendo al primer punto de este capítulo, hay una forma de recuperar el tamaño del punto que estamos usando actualmente, invocando a glGet, con el parámetro **GL_POINT_SIZE**:

```
float tam;
glGetFloatv(GL_POINT_SIZE, &tam);
```

+ Suavizando el dibujado de los puntos:

Normalmente, al pintar un punto se pinta como si fuera un pixel, con forma de cuadrado. A tamaños pequeños no es un problema, pero con tamaños grandes es un poco feo. Por ello hay un parámetro para suavizar los contornos, que es **GL_POINT_SMOOTH**, que ciertamente activaremos con glEnable.

Sin embargo, activado el parámetro no está garantizado que se vayan a suavizar los puntos para cualquier tamaño, por ello podemos obtener cual es el tamaño mínimo y máximo del rango aceptado, y el intervalo entre cada elemento de este rango. Por ejemplo podemos tener un rango del 0.5 al 10.0, con una granularidad de 0.1, es decir, que para que se note algún cambio si tenemos un tamaño de 0.5, hemos de pasar a 0.6.

Para obtener los datos relacionados con estos límites tenemos:

```
GLfloat sizes[2], granularity;

glGetFloatv(GL_POINT_SIZE_RANGE, sizes);
```

```
GLfloat minPointSize = sizes[0];
GLfloat maxPointSize = sizes[1];

glGetFloatv(GL_POINT_SIZE_GRANULARITY, &granularity);
```

b) Dibujando lineas:

+ Modificar el ancho de linea:

```
void glLineWidth (GLfloat width);
```

El ancho por defecto es 1.0, y en caso de cambiarlo podemos preguntar a glGetFloatv por **GL_LINE_WIDTH**.

+ Suavizando las lineas:

Para poder suavizar las lineas hay que activar **GL_LINE_SMOOTH** con glEnable. Una vez activado, para preguntar por el rango y la granularidad como lo hacíamos en los puntos, usamos **GL_LINE_WIDTH_RANGE** y **GL_LINE_WIDTH_GRANULARITY**.

+ Indicando un patrón:

Para poder usar patrones al pintar lineas, hemos de activar con glEnable el parámetro **GL_LINE_STIPPLE**. Después solo hay que indicar el patrón a usar con la función:

```
void glLineStipple (GLint factor, GLushort pattern);
+ factor = N° de veces que se repite cada bit en pattern (1-256).
+ pattern = Patrón a seguir en binario (16bits).
```

Para recuperar estos valores con glGet tenemos los parámetros **GL_LINE_STIPPLE_PATTERN** y **GL_LINE_STIPPLE_REPEAT**.

c) Dibujando polígonos:

Por defecto los polígonos son pintados de un modo tal que se ven con relleno, pero eso puede ser cambiado con:

```
void glPolygonMode (GLenum face, GLenum mode);
+ face:
   - GL_FRONT = Las caras delanteras.
   - GL_BACK = Las caras traseras.
   - GL_FRONT_AND_BACK = Ambas caras.
+ mode:
   - GL_POINT = Solo se pintan los puntos de los vértices.
   - GL_LINE = Solo se pintan los bordes del polígono con lineas.
   - GL_FILL = El polígono se pinta con relleno.
```

Para recuperar el modo de pintado tenemos **GL_POLYGON_MODE** como parámetro en glGet.

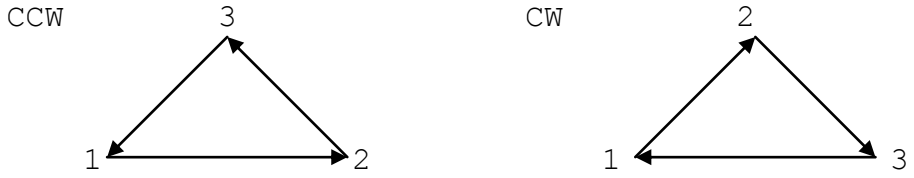
+ Realizando culling:

Como normalmente hay una cara que nunca veremos de los polígonos, existe el parametro **GL_CULL_FACE** que se activa con glEnable, para usar:

```
void glCullFace (GLenum mode);
+ mode:
   - GL_FRONT = No renderiza la parte de delante.
   - GL_BACK = No renderiza la parte de detras (por defecto).
   - GL_FRONT_AND_BACK = No renderiza ningún lado.
```

Existe otra cuestión, ¿cual es la cara delantera y cual la trasera? Para ello hay una función con la que indicarlo:

```
void glFrontFace (GLenum mode);
+ mode:
   - GL_CCW = Sentido contra-reloj (por defecto).
   - GL_CW = Sentido de las agujas del reloj.
```



+ Respecto a los bordes:

Para eliminar los bordes innecesarios a la hora de pintar figuras en modo "alambre" tenemos:

```
void glEdgeFlag (GLboolean isEdge);
+ isEdge:
   - GL_TRUE = Pinta todos los bordes (por defecto).
   - GL_FALSE = Pinta solo los bordes necesarios.
```

+ Suavizando los polígonos:

Como con los puntos y las lineas, activando **GL_POLYGON_SMOOTH** con glEnable, los bordes de los polígonos serán suavizados.

Capítulo 5: Transformaciones de coordenadas y matrices.

1) Comprendiendo las transformaciones de coordenadas:

Las transformaciones sirven para mover, rotar o redimensionar objetos en el mundo 3D. Aunque conceptualmente pueda parecer que lo que hacemos es cambiar los valores (x, y, z) de los vertices del objeto a pintar, la realidad es que lo que es modificado es el eje de coordenadas para ese objeto. Por lo que si movemos 100 unidades algo, el eje de coordenadas de dicho objeto estará 100 unidades más lejos. Hay diferentes tipos de transformaciones:

+ Viewing: Para indicar la posición de la camara.
+ Modeling: Para indicar la posición de los objetos.
+ Projection: Para definir el volumen de vista y los planos para el clipping.
+ Viewport: Realiza una proyección 2D sobre la ventana de windows.
+ Modelview: Es la combinación de las transformaciones viewing y modeling.

Es importante recordar que las transformaciones han de seguir un orden lógico, por ejemplo no se puede realizar una de modeling antes que la de viewing. Y las de projection y viewport se han de realizar antes de completar el renderizado. De tal modo que internamente OpenGL realiza el siguiente recorrido:

```
Vertex Data (x, y, z, w) -> [Modelview Matrix] -> Eye Coordinates
-> [Projection Matrix] -> Clip Coordinates -> [Perspective Division]
-> Normalized Device Coordinates -> [Viewport Transformation]
-> Window Coordinates (x, y)

P' = Mv * Mp * Mmv * P
```

2) Las coordenadas de la cámara y el ojo:

El sistema de coordenadas de OpenGL es un sistema cartesiano de mano derecha, esto es que el eje Y es el que indica la altura (arriba +, abajo -), el X desde la izquierda a la derecha (de menos a más), y el eje Z indica la profundidad, de tal modo que cuanto más lejos menos vale Z, y cuanto más cerca más vale Z.

Por defecto la camara está mirando en paralelo al eje Z, con un vector direccional (0, 0, -1), es decir mirando hacia el menos infinito de la Z. Así que cada vez que modificamos el eje de coordenadas de un objeto, lo modificamos con respecto a las coordenadas de nuestra camara u ojo. De ese modo, conceptualmente, para mover la "camara" por el mundo, deberíamos mover el mundo entero.

3) Transformaciones de tipo Viewing:

O también podría ser llamado transformaciones de la cámara. Siempre que vayamos a pintar objetos esta es la primera transformación que debemos aplicar, antes de mandar los vertices. Pero antes de mandar cualquier orden, hay que limpiar la matriz en uso cargando la matriz identidad con:

```
void glLoadIdentity (void);
```

Y después de borrar lo cambios de la matriz, tenemos varias formas de cambiar la camara, la primera es llamando a la función gluLookAt():

```
void gluLookAt (GLdouble eyex, GLdouble eyey, GLdouble eyez,
                GLdouble centerx, GLdouble centery, GLdouble centerz,
                GLdouble upx, GLdouble upy, GLdouble upz);
```

Los parametros eye hacen referencia a la posición del ojo o de la cámara, los parametros center indican hacia donde está mirando la cámara, y los parametros up indican cual es la dirección que apunta arriba (0.0f, 1.0f, 0.0f en el caso de OpenGL). Otro método es usando las funciones glRotate*() y glTranslate*(), pudiendo hacerte tu propia función de mover la camara:

```
void MueveCamara (GLfloat planex, GLfloat planey, GLfloat planez,
                  GLfloat roll, GLfloat pitch, GLfloat yaw)
{
    // roll indica el giro en el eje Z
    glRotatef(roll, 0.0f, 0.0f, 1.0f);
```

```
    // yaw indica el giro en el eje Y
    glRotatef(yaw, 0.0f, 1.0f, 0.0f);

    // pitch indica el giro en el eje X
    glRotatef(pitch, 1.0f, 0.0f, 0.0f);

    // y movemos la camara a unas coordenadas
    glTranslatef(-planex, -planey, -planez);
}
```

4) Las transformaciones de tipo Modelview:

+ La matriz Modelview:

Como vimos al principio del capítulo hay varios tipos de matrices que sirven para transformar a los vertices que mandamos a renderizar. Esta en particular sirve para situar los objetos que vamos a pintar con respecto a la posición de la camara. Para indicarle a OpenGL que queremos modificar esta matriz hemos de usar:

```
void glMatrixMode (GLenum mode);
+ mode:
   - GL_MODELVIEW: Indica que usaremos la matriz de modelado-vista.
   - GL_PROJECTION: Indica que vamos a modificar la proyección.
   - GL_TEXTURE: Indica que vamos a modificar una textura.
```

Así que cuando vayamos a pintar los objetos, llamamos a glMatrixMode, luego a glLoadIdentity, realizamos las transformaciones que necesitemos y mandamos los vertices del objeto a OpenGL, encargandose este de realizar las transformaciones acumuladas.

+ Moviendo los objetos:

Para mover objetos por el espacio tridimensional usaremos:

```
void glTranslatef (GLfloat x, GLfloat y, GLfloat z);
void glTranslated (GLdouble x, GLdouble y, GLdouble z);
```

Cuyos parametros indican el punto donde queremos colocar el objeto:

```
glMatixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(5.0f, 5.0f, 5.0f);
DrawCube();
```

+ Rotando los objetos:

Para rotar objetos por el espacio tridimensional usaremos:

```
void glRotatef (GLfloat angle, GLfloat x, GLfloat y, GLfloat z);
void glRotated (GLdouble angle, GLdouble x, GLdouble y, GLdouble z);
```

Cuyos parametros son angle, que representa el número de grados que desamos que rote el objeto, y un vector (x, y, z) que indica el eje sobre el que hay que rotar. La forma de medir los grados es contra-reloj. Por ejemplo: glRotatef(90.0f, 0.0f, 1.0f, 0.0f); realiza un giro hacia la izquierda de 90º sobre el eje Y.

+ Redimensionando los objetos:

Para redimensionar objetos por el espacio tridimensional usaremos:

```
void glScalef (GLfloat x, GLfloat y, GLfloat z);
void glScaled (GLdouble x, GLdouble y, GLdouble z);
```

Cuyos parametros indican el tamaño que deben tener sobre cada eje. Por ejemplo, si queremos hacer algo el doble de grande llamaremos a: glScalef(2.0f, 2.0f, 2.0f);

+ Las pilas de matrices:

Una gran herramienta interna que tiene OpenGL con las matrices, es las pilas de matrices. Hay una pila por cada uno de los tres tipos de matrices que glMatrixMode puede invocar. Hay básicamente dos instrucciones que se pueden realizar:

```
void glPushMatrix (void);
void glPopMatrix  (void);
```

Haciendo un push lo que le decimos a OpenGL es que haga una copia de la matriz actual, y la meta en la pila de matrices. Y con pop que elimine la matriz que está actualmente en la cima. De este modo OpenGL siempre trabajará con la matriz que esté en la cima. ¿Para qué podría servir este sistema? Es útil para dibujar objetos compuestos, que pudieran estar animados, por ejemplo un cuerpo humano. Dibujaríamos primero el cuerpo ya posicionado, y luego cada extremidad en relación a la posición del cuerpo.

5) Las transformaciones de la proyección:

Hay dos tipos principales de proyecciones: ortográfica y paralela. Para poder trabajar con ellas hemos de invocar a glMatrixMode(GL_PROJECTION); y así OpenGL sabrá que tiene que trabajar con la matriz de proyección.

+ Proyección ortográfica:

Este tipo de proyección proyecta los objetos sobre la pantalla con su tamaño real, sin importar cuan lejos estén del observador. Y para poder cambiarla tenemos la función:

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

Los parametros son bien sencillos, hay que definir en este orden la posición del plano izquierdo, del derecho, del bajo, del alto, del cercano y del lejano. Siempre de menos a más en los eje X e Y, pero de más a menos en el Z. También está:

```
void gluOrtho2D (GLdouble left, GLdouble right,
                 GLdouble bottom, GLdouble top);
```

Solo que en esta función da por supuesto que near es 1.0 y far -1.0, lo cual es útil si vamos realmente a trabajar en 2D todo el rato, sin necesidad de que haya una porfundidad muy grande para trabajar.

+ Proyección con perspectiva:

Este tipo de proyección es la que se suele llamar cónica, pues cuan más lejos están los objetos a ser proyectados, más pequeños se verán. Para poder cambiarla se usa:

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

Los parametros vuelven a ser como los de glOrtho(). Pero normalmente se suele emplear otra función llamada:

```
void gluPerspective (GLdouble fov, GLdouble aspect,
                     GLdouble near, GLdouble far);
```

El parametro fov es el campo de visión del observador, indicado en grados, así que lo normal sería poner 90.0f, para indicar un campo de visión de 90º. El siguiente, aspect, sirve para indicar la relación entre el ancho y el alto, así que se le suele pasar el ancho dividido entre el alto. Y por último near y far, sirven para indicar donde está el plano más cercano (1.0f normalmente) y el más lejano (cualquier valor positivo mayor que near).

+ El viewport:

Por último el viewport es el area de trabajo sobre la que vamos a renderizar, en la ventana o contexto que hayamos creado. Y se cambia con la función:

```
void glViewport (GLint x, GLint y, GLsizei width, GLsizei height);
```

Básicamente hay que indicar en qué posición, y cual es el ancho y alto, de la región sobre la que OpenGL volcará la proyección de la escena tridimensional que hemos mandado a renderizar.

6) Usando matrices propias:

Aparte de todo lo visto anteriormente, OpenGL da la oportunidad al usuario de manejar las matrices de las que dispone de una forma más avanzada. Para ello primero tenemos que saber como se almacena una matriz en OpenGL. Hasta ahora sabemos que es una matriz de 4x4, pero además podemos tener matrices de tipo GLfloat o GLdouble. La forma de declararlas y almacenarlas es la siguiente:

```
GLfloat identity[16] = {1.0f, 0.0f, 0.0f, 0.0f,
                        0.0f, 1.0f, 0.0f, 0.0f,
                        0.0f, 0.0f, 1.0f, 0.0f,
                        0.0f, 0.0f, 0.0f, 1.0f};

GLdouble identity[16] = {1.0, 0.0, 0.0, 0.0,
                         0.0, 1.0, 0.0, 0.0,
                         0.0, 0.0, 1.0, 0.0,
                         0.0, 0.0, 0.0, 1.0};
```

Sabiendo esto ya podemos realizar el primer paso, que consiste en cargar una matriz personalizada, en OpenGL. Para ello usaremos:

```
void glLoadMatrixf (const GLfloat * matrix);
void glLoadMatrixd (const GLdouble * matrix);
```

Bien, ya podemos trastear bastante, pero además no estaría mal que pudieramos hacer otras cosas, como por ejemplo multiplicar una matriz a la que hay actualmente almacenada en la máquina OpenGL:

```
void glMultMatrixf (const GLfloat * matrix);
void glMultMatrixd (const GLdouble * matrix);
```

Y esto es todo lo que necesitamos para trabajar con las matrices propias en OpenGL.

Capítulo 6: Añadiendo colores, fusiones e iluminación.

1) Manejando colores en OpenGL:

El color está compuesto por tres tipos diferentes de color: el rojo, el verde y el azul (red, green y blue en inglés). Juntos forman las siglas RGB, que suele ser bastante utilizada. Además en las apis gráficas y diversos formatos de ficheros existe una cuarta componente, el canal Alpha, que sirve para indicar el nivel de transparencia de ese pixel. Así que a este modo se le denomina RGBA y sobre esta base se sustenta OpenGL.

+ La profundidad de color:

La profundidad de color consiste básicamente en el número de colores que se pueden representar con un pixel. Los diferentes valores estandar que hay son 8, 16, 24 y 32. Pero a día de hoy se utiliza principalmente 32, que permite 8bits para cada componente del modo RGBA.

+ El espectro de colores:

Aunque el módo clásico de indicar los colores en el 2D era con un entero (de 0 a 255 para cada componente en el modo 32bits de profundidad), en OpenGL normalmente se emplea 0.0f para indicar ausencia total de dicho componente, y 1.0f para indicar presencia plena del componente en cuestión. Esto es una ventaja si llegara algún día hipotéticamente a ampliar el rango de valores para cada componente.

Con este sistema, indicando la cantidad de cada componente, podemos abarcar el espectro completo de colores en la luz, desde el negro hasta el blanco, pasando por el azul, el rojo, el verde, el cyan, el amarillo, el magenta, y demás colores.

+ El modo RGBA:

Entonces, para poder especificar un color en el modo RGBA, tenemos que indicar cada componente a la función glColor*(). Esta función tiene muchas combinaciones posibles, pero las que más se suelen usar son:

```
void glColor3f  (GLfloat r, GLfloat g, GLfloat b);
void glColor4f  (GLfloat r, GLfloat g, GLfloat b, GLfloat a);
void glColor3fv (const GLfloat * v);
void glColor4fv (const GLfloat * v);
```

Con estas funciones podemos indicar el color que usarán las primitivas que pintemos en pantalla, que normalmente indicaremos un color para cada primitiva. Pero se puede

dentro de la primitiva, para cada vértice indicar un color distinto. Luego también hay una función para poder indicar el color con el que vamos a realizar el borrado de la pantalla:

```
void glClearColor (GLfloat r, GLfloat g, GLfloat b, GLfloat a);
```

Con esta función y la función glClear(), podremos borrar la pantalla del siguiente modo:

```
glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);
```

+ El modo paleta:

Antaño en MS-DOS, con la falta de recursos se usaban paleta de colores, para poder superar las limitaciones del hardware. OpenGL también dispone mecanismos para poder usar una paleta de colores. Una vez definida la paleta, para elegir un color se usa:

```
void glIndexf (GLfloat c);
```

También podemos indicar un color de la paleta, como color de borrado de pantalla, con:

```
void glClearIndex (GLfloat cindex);
```

+ Suavizado o shading:

Como se indicó en el apartado del modo RGBA, podemos indicar para cada vértice de una primitiva, un color distinto. Esto por defecto supondría un pequeño problema, porque ¿qué tendría que hacer OpenGL con los colores? Bueno, hay una función para poder resolver este conflicto:

```
void glShadeModel (GLenum mode);
+ mode:
   - GL_FLAT = Sin suavizado.
   - GL_SMOOTH = Con suavizado.
```

Si elegimos GL_FLAT, OpenGL cogerá el último color indicado en los vértices, y ese será con el que se pinte la primitiva entera. Sin embargo, con GL_SMOOTH, cogerá todos los colores y hará un fusionado progresivo entre ellos. Este modo de "sobreado" se le llama también *Gouraud shading*.

2) Iluminación en OpenGL:

Y finalmente uno de los puntos un tanto cruciales, para dar realismo a un espacio tridimensional, la iluminación. Pero antes de dar código y funciones, hay que comprender algunas cuantas nociones sobre teoría básica del tema.

+ La iluminación en OpenGL y en el mundo real:

En el mundo real para que podamos ver los colores de las cosas, necesitamos la luz. Esto es así porque la luz lleva fotones, que al impactar sobre las cosas, son absorbidos o reflejados. Si una pared es de pintura roja, es porque los fotones que abarcan el espectro del color rojo, no son totalmente absorbidos por dicho material. De este modo hay materiales que absorbe muy poca cantidad de un color y lo refleja con tal intensidad, que es lo que llamamos el brillo.

Así que en OpenGL nosotros debemos indicar donde están las luces y de qué tipo son, luego hemos de pintar los objetos, e indicar el tipo de material del que están compuesto. Luego internamente se calculará como es iluminado los objetos con la luz, y como estos absorben los componentes RGB de la luz. Pero a parte del color, hay cuatro tipos de luz:

+ Luz ambiental: No tiene un punto de origen específico, por lo que no tiene dirección. Las superficies que reciben esta luz, la reflejan en todas las direcciones.
+ Luz difusa: Es una luz con un punto de origen y una dirección determinada, que al impactar contra una superficie, es reflejada en todas las direcciones. De ese modo, la superficie brillará con la misma intensidad, desde cualquier punto de vista distinto.
+ Luz especular: Comparte con la difusa que tiene un punto de origen, y que tiene una dirección. Sin embargo es reflejada con más intensidad, en una determinada dirección, creando puntos brillantes sobre una superficie, para determinados puntos de vista.
+ Luz emisiva: Esto ocurre cuando un objeto emite una luz, que no afecta a los demás objetos de la escena. Y la luz de la escena tampoco afecta al objeto.

+ Los materiales:

En OpenGL los materiales hacen referencia al modo en que reflejan la luz los objetos. Es decir, que las propiedades de un material son unos valores RGB por cada tipo de luz soportados por el api (ambiental, difusa y especular). Con esto tenemos bastante control, y podemos decidir que un objeto tenga capacidad de reflejar los colores que deseemos, con diferentes tipos de luz a la vez.

3) La importancia de las normales:

¿Qué son las normales? Bien, la normal a un plano es un vector perpendicular a este. ¿Y para qué sirve esto? Bueno, es un dato bastante útil a la hora de calcular la iluminación sobre un polígono, para que OpenGL pueda calcular el color y sombreado del mismo, a la hora de renderizarlo. Dicho cálculo consiste en obtener el angulo formado de la normal con el vector direccional de la luz.

Sin embargo una normal por cada polígono, sería un poco cutre y por ello se puede indicar una normal para cada vértice de este. Así pues, OpenGL se encargaría de realizar una mezcla o fusión de la iluminación entre los diferentes vértices del polígono. Algo parecido a la fusión de colores, cuando queríamos que cada vértice tuviera un color diferente.

+ Calculando las normales:

Para calcular la normal de un vértice N, necesitamos dos vectores, que los formaría N con los vértices N - 1 y N + 1. Una vez se tiene los vectores, la formula es:

A = Nsig - N
B = Nant - N

A x B = (A.y * B.z - A.z * B.y, A.z * B.x - A.x * B.z, A.x * B.y - A.y * B.x)

El resultado es un nuevo vector, que dependiendo de la regla del sacacorchos no será lo mismo hacer A x B que B x A. De hecho en un espacio 2D, la regla del sacacorchos nos dice que busquemos el camino más corto entre el primer operando y el segundo, si para ello hemos de girar hacia la izquierda, la normal saldrá del plano hacia arriba, si hay que girar a la derecha, saldrá hacia abajo.

Esto dicho de una forma más seria es, que el angulo entre el primer operando A y el segundo B, de ser mayor de 180º, la normal saldrá hacia abajo del plano, y si es menor hacia arriba. Lo curioso es que para el caso de que el ángulo sea 0º o 180º, la magnitud de la normal vale cero, esto se da si los vectores son paralelos.

```
void ProductoVectorial (Punto3D pnant, Punto3D pnact, Punto3D pnsig,
                        Vector3D * normal)
{
    Vector3D A, B;

    A.x = pnsig.x - pnact.x;
    A.x = pnsig.y - pnact.y;
    A.x = pnsig.z - pnact.z;

    B.x = pnant.x - pnact.x;
    B.x = pnant.y - pnact.y;
    B.x = pnant.z - pnact.z;

    normal->x = A.y * B.z - A.z * B.y;
    normal->y = A.z * B.x - A.x * B.z;
    normal->z = A.x * B.y - A.y * B.x;
}
```

Con esta función podríamos calcular la normal de un vértice, siempre que metamo los puntos bien, es decir, sigamos el orden contra-reloj para elegir el punto siguiente al punto actual N, porque sino la normal saldría por la cara que no se pinta.

+ Usando las normales:

Así que sabiendo que son, y como podemos calcularlas, ahora toca saber como podemos modificarlas a nuestro antojo. Para ello tenemos:

```
void glNormal3f  (GLfloat nx, GLfloat ny, GLfloat nz);
void glNormal3fv (const GLfloat * normal);
```

Que tendremos que llamarla antes de cada vértice cuya normal querramos modificar.

+ La normal unitaria o unidad:

Para realizar los cálculos de iluminación, OpenGL necesita convertir todas las normales que recibe con glNormal3f(), a normales unitarias. Para ello hay que normalizar el vector de la normal, que consiste en que ninguna de las componentes del vector valga mayor que 1.0f. La formula era:

u = N / |N|

|N| = sqrt(N.x^2, N.y^2, N.z^2)

Aunque podría uno hacerse una función para realizar este cálculo, con glEnable() podemos activar GL_NORMALIZE, y OpenGL se encargará de realizar el cálculo de la normalización para cada vector normal que metamos.

```
glEnable(GL_NORMALIZE);
```

También está GL_RESCALE_NORMALIZE, que se usa cuando teníamos ya las normales normalizadas, y queremos que cada componente sea multiplicado por la matriz de Modelview.

4) Usando la iluminación de OpenGL:

Vistos algunos conceptos necesarios, ahora hablaremos sobre las luces. OpenGL por escena permite hasta 8, ya que más luces por cada vértice supondrían un coste de cálculos exagerado. Así que para montar esto hay que seguir los pasos siguientes:

+ Calcular el vector normal de cada vértice, de cada objeto. Pues las normales indican cual es la orientantción con respecto la luz.
+ Crear, seleccionar y posicionar todas las fuentes de luz.
+ Crear y seleccionar el modelo de iluminación. Esto es para definir la luz ambiental y la localización del punto de vista (la camara) para los cálculos de iluminación.
+ Definir el material para cada objeto de la escena.

Ya que lo primero ya lo sabemos hacer, es el resto de cosas las que necesitamos aprender a hacer. Lo primero es saber como almacenamos los datos en cuestión:

```
float * AmbientLight   = {0.3f, 0.5f, 0.8f, 1.0f};
float * DiffuseLight   = {0.25f, 0.25f, 0.25f, 1.0f};
float * LightPosition  = {0.0f, 0.0f, 0.0f, 1.0f};
float * LightDirection = {0.0f, 0.0f, 1.0f, 0.0f};

float * MatAmbient = {1.0f, 1.0f, 1.0f, 1.0f};
float * MatDiffuse = {1.0f, 1.0f, 1.0f, 1.0f};
```

Como se puede ver, usamos arrays de 4 posiciones (x, y, z, w), tanto para indicar puntos, como vectores, como colores en modo RGBA. Las dos primeras variables son los colores que tendrán la luz, a dos niveles diferentes (el ambiental y el difuso). Lo tercero es la posición de la luz, lo cuarto su dirección. Y las dos últimas son el material que se va a usar, para cada nivel de luz. Cuando vale 1.0f significa que no absorverá nada del color, y este será reflejado tal como ha llegado.

```
void Initialize (void)
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f); //Color de borrado el negro.

    glShadeModel(GL_SMOOTH); // Usar suavizado para los polígonos.
    glEnable(GL_DEPTH_TEST); // Eliminar los polígonos no visibles.
    glEnable(GL_CULL_FACE);  // No renderizar las caras traseras.
    glFrontFace(GL_CCW);     // Cara delantera en modo contra-reloj.

    glEnable(GL_LIGHTING);   // Activar la iluminación.

    glMaterialfv(GL_FRONT, GL_AMBIENT, MatAmbient);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, MatDiffuse);

    glLightfv(GL_LIGHT0, GL_AMBIENT, AmbientLight);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, DiffuseLight);
    glLightfv(GL_LIGHT0, GL_POSITION, LightPosition);

    glEnable(GL_LIGHT0); // Activar luz número 0.
}
```

En esta función hemos activado algunos parámetros, y definido un material para los objetos, y una luz, que hemos decidido que fuera la número cero. Como hemos dicho antes, hay un máximo de ocho, así que tendremos desde GL_LIGHT0 hasta GL_LGITH7.

```
void Render (void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    glTranslatef(0.0f, 0.0f, -3.0f);

    glBegin(GL_QUADS);
        glNormal3f( 0.0f,  0.0f, 1.0f);
        glVertex3f( 0.5f,  0.5f, 0.5f);
        glVertex3f(-0.5f,  0.5f, 0.5f);
        glVertex3f(-0.5f, -0.5f, 0.5f);
        glVertex3f( 0.5f, -0.5f, 0.5f);
    glEnd();

    glFlush();
    SwapBuffers(g_hDC);
}
```

Y así pintariamos un cuadrado iluminado por una y con un material especificado.

+ Creando fuentes de iluminación:

Como se puede apreciar en el anterior ejemplo para crear y definir una luz haremos uso de la función glLightfv():

```
void glLightfv (GLenum light, GLenum pname, TYPE * param);
void glLightf  (GLenum light, GLenum pname, TYPE param);
+ light: Número de luz -> GL_LIGHT0..GL_LIGHT7
+ pname:
   - GL_AMBIENT: Intensidad ambiental de la luz.
   - GL_DIFFUSE: Intensidad difusa de la luz.
   - GL_SPECULAR: Intensidad especular de la luz.
   - GL_POSITION: Posición de la luz (x, y, z, w).
   - GL_SPOT_DIRECTION: Vector direccional del foco de luz.
   - GL_SPOT_EXPONENT: Exposición del foco de luz.
   - GL_SPOT_CUTOFF: Ángulo de corte para un foco de luz.
   - GL_CONSTANT_ATTENUATION: Valor para una atenuación constante.
   - GL_LINEAR_ATTENUATION: Valor para una atenuación lineal.
   - GL_QUADRATIC_ATTENUATION: Valor para una atenuación cuadrática.
```

Así que si queremos darle color a la luz, y queremos que tenga diferentes niveles de luz, tendremos que ir llamando a la función glLightfv(), indicándole un color RGBA a los parametros GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, como ocurría en el ejemplo anterior del cuadrado. Los valores por defecto si no configuramos las luces es (0.0f, 0.0f, 0.0f, 0.0f) como color, para los parametros de nivel de luz.

+ Posicionando las fuentes de iluminación:

Entrando en más detalle respecto a las posiciones de las luces, hay dos formas de configurar el parametros GL_POSITION, pasandole un punto o un vector:

```
float * LightPosition  = {0.0f, 0.0f, 0.0f, 1.0f};
float * LightDirection = {0.0f, 0.0f, 1.0f, 0.0f};
```

```
glLightfv(GL_LIGHT0, GL_POSITION, LightPosition);
glLightfv(GL_LIGHT1, GL_POSITION, LightDirection);
```

En este ejemplo tenemos que la luz cero está en el punto (0, 0, 0) de su eje de coordenadas, pero la luz uno no está en ningún punto. ¿Entonces donde está? Bueno en vez de estar en un lugar, diríamos que hemos definido cual es la dirección que toman sus rayos de luz. Esto es útil para cuando queremos definir luces como la del sol, que están en un lugar, pero es tan lejano que prácticamente la dirección de sus rayos de luz, será idéntica para toda la escena. A esto se le llama "luz direccional", y lo anterior "luz posicional".

+ Atenuación:

Toda luz tiene una propiedad que es la atenuación. Esto consiste en que a medida que un rayo va alejándose de la fuente va perdiendo intensidad, y viene bien para hacer por ejemplo una farola en la oscuridad de la noche... Porque no sería normal que iluminara todo el escenario como si fuera un sol. Sin embargo la atenuación no tiene mucho sentido para las luces direccionales, ya que estas son "infinitas" en cierto modo.

Hay tres grados de atenuación: la constante, la lineal y la cuadrática. Por defecto cada una vale 1.0, 0.0 y 0.0, y para configurarlas se haría del siguiente modo:

```
glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION,  4.0f);
glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION,    1.0f);
glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.25f);
```

A lo único que no afecta la atenuación es a los valores de emisión y luces de ambiente globales, al resto sí. Y obviamente dada la cantidad de operaciones a realizar cuando se activa la atenuación, el rendimiento de la aplicación se verá mermado.

+ Focos de luz o spotlights:

Por defecto, las luces lanzan rayos en todas las direcciones, desde la posición en la que fueron situadas (salvo que sea una luz direccional), y para crear efectos como el de una farola, eso es un poco excesivo. Por ello podemos crear luces que sean focos de luz. El primer parametro importante es GL_SPOT_CUTOFF:

```
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 15.0f);
```

Este parametro sirve para indicar en grados, la mitad del ángulo de dispersión o de corte de la luz. Es decir, una luz normal sin ser un foco de luz, tendría este parametro a 180.0f. Vale ahora que hemos delimitado el area de efecto de la luz, le daremos una dirección:

```
float * SpotlightDirection = {0.0f, -1.0f, 0.0f};
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, SpotlightDirection);
```

Así que ahora que hemos indicado la amplitud y la dirección, solo nos queda la exposición, que consiste en la cantidad de concentración de luz en el punto central al que apunta el foco (que por defecto es 1.0f):

```
glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, 10.0f);
```

Y con estos tres parametros queda configurado un foco de luz.

+ Definiendo materiales:

En el primer ejemplo, se veía como crear un material. Y ahora veremos la función que se usa para realizar esta tarea en más profundidad:

```
void glMaterialfv (GLenum face, GLenum pname, TYPE * param);
void glMaterialf  (GLenum face, GLenum pname, TYPE param);
+ face:
   - GL_FRONT = Solo las caras delanteras usarán el material.
   - GL_BACK = Solo las caras traseras usarán el material.
   - GL_FRONT_AND_BACK = Ambas caras usarán el material.
+ pname:
   - GL_AMBIENT = Color de ambiente del material.
   - GL_DIFFUSE = Color de difusión del material.
   - GL_AMBIENT_AND_DIFFUSE = Color de ambiente y de difusión.
   - GL_SPECULAR = Color especular del material.
   - GL_SHININESS = Exposición especular del material.
   - GL_EMISSION = Color de emisión del material.
```

De los parametros que podemos configurar, salvo GL_SHININESS, que recibe un float con la exposición máxima que el material puede recibir de una luz de tipo especular, el resto lo que recibe como valor es un color de tipo RGBA. Todo lo que pintemos después de llamar a glMaterial*(), tendrá asociado el último material definido por el programador.

Otra forma de establecer un material, es mediante el color del objeto activando con glEnable() el parametro GL_COLOR_MATERIAL, y llamando a la función:

```
void glColorMaterial (GLenum face, GLenum pname);
```

Los argumentos face y pname son los mismos que glMaterial*(), ¿y como se le indica el color que le podíamos con la otra función indicar? Pues del siguiente modo:

```
glEnable(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT, GL_DIFFUSE);
glColor3f(1.0f, 0.0f, 0.0f);
glBegin(GL_TRIANGLES);
   // vértices...
glEnd();
```

Ciertamente es más limitado este mecanismo, pero para cosas sencillas no viene mal.

+ Modelos de iluminación:

Todavía no hemos comentado nada sobre el modelo de iluminación, que deberíamos crear para que nuestra escena tenga una buena iluminación, que nos permite definir:

+ La intensidad de la luz ambiental de la escena.
+ Si la localización de la cámara es local o infinita (afecta al cálculo del ángulo de la reflexión-especular).
+ Iluminación para un lado o ambos.
+ Si el color de la luz especular está separado del ambiental y el difuso.

Y para poder realizar estos cambios usaremos:

```
void glLightModeli  (GLenum pname, TYPE param);
void glLightModelf  (GLenum pname, TYPE param);
void glLightModeliv (GLenum pname, TYPE * param);
```

```
void glLightModelfv (GLenum pname, TYPE * param);
+ pname:
   - GL_LIGHT_MODEL_AMBIENT = La intensidad del ambiente de la escena
     con un color RGBA. Por defecto es (0.2, 0.2, 0.2, 1.0).
   - GL_LIGHT_MODEL_LOCAL_VIEWER = Si la camara está en una distancia
     local o infinita. GL_TRUE -> local, GL_FALSE -> infinita. Por
     defecto vale GL_FALSE.
   - GL_LIGHT_MODEL_TWO_SIDE = Iluminación por un lado o ambos, por
     defecto vale GL_FALSE (por un solo lado).
   - GL_LIGHT_MODEL_COLOR_CONTROL = Si el color especular ha de ser
     calculado separado del ambiental y del difuso. Por defecto vale
     GL_SINGLE_COLOR (sin separación), pero para calcularlos por separado
     se tiene que pasarle GL_SEPARATE_SPECULAR_COLOR.
```

El primer parametro GL_LIGHT_MODEL_AMBIENT hace referencia a la luz de ambiente global, y con esta función podemos darle el color que queramos. Cuanto más cerca del blanco la aproximemos, más intensidad tendrá la escena.

```
float * AmbientColor = {0.5f, 0.5f, 0.5f, 1.0f};
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, AmbientColor);
```

El segundo parametro GL_LIGHT_MODEL_LOCAL_VIEWER tiene que ver con los cálculos relacionados con la luz especular. Dependiendo de si el observador está en el "infinito", o está presente en la escena de forma local, se harán unos cálculos o no. Obviamente si está de forma local será más realista el renderizado, aunque más lento.

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

El tercer parametro GL_LIGHT_MODEL_TWO_SIDE se refiere a si para ambas caras del polígono tienen que ser calculada su iluminación. Normalmente las caras traseras de los polígonos no son visibles, por lo que este parametro no se suele activar.

```
glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);
```

El cuarto parametro GL_LIGHT_MODEL_COLOR_CONTROL indica a OpenGL si queremos que los cálculos de la luz especular, se hagan aparte de los del resto. ¿Para qué podría servir esto? Bueno, al mapear texturas a los polígonos, se da a veces el caso de que determinadas texturas no son bien iluminadas, ya que los cálculos realizados resultan poco favorecedores. Así que para esos casos, le mandamos a OpenGL realizar por separado el cálculo de la luz especular con:

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

Así que se crean dos colores para el vertice, uno el primario el de las demás luces, y el secundario, resultado de la luz especular reflejada. Así que cuando la textura es mapeada, esta es aplicada con el color primario, y es después cuando se le añade encima el color secundario calculado, para realzar el brillo especular sobre el objeto.

+ Efectos de iluminación especular:

Con lo visto hasta ahora ya podemos crear una luz, que cree puntos brillantes sobre una superficie preparada con el material apropiado, como ocurre en los juegos de coches. Pero claro, el sol no es un foco de luz, así que necesitamos una luz especular para conseguir dicho brillo reflectante. Para ello haríamos lo siguiente:

```
float * SpecularLight = {1.0f, 1.0f, 1.0f, 1.0f};
float * LightPosition = {0.0f, 0.0f, 0.0f, 1.0f};
```

```
...
glEnable(GL_LIGHTING);
...
glLightfv(GL_LIGHT0, GL_SPECULAR, SpecularLight);
glLightfv(GL_LIGHT0, GL_POSITION, LightPosition);
glEnable(GL_LIGHT0);
```

Con este código habremos creado una luz especular con una posición en particular, si quisieramos crear un sol, pues en vez de una posición, le pasaríamos un vector direccional. Además el color que le hemos pasado es el blanco, con lo que esta luz será muy brillante. Después de esto tendremos que configurar el material:

```
float * MatSpecular = {1.0f, 1.0f, 1.0f, 1.0f};

glMaterialfv(GL_FRONT, GL_SPECULAR, MatSpecular);
glMaterialf(GL_FRONT, GL_SHININESS, 10.0f);
```

Y ya está el material definido, uno que reflecta toda la luz que le llega, al haberle pasado el blanco como el color que refleja el material si le alcanza una luz especular. En la última linea se le indica cuan grande es la exposición de la luz en un mismo punto de brillo. Se le puede pasar cualquier valor entre 1.0 y 128.0, ya que con 0.0 no conseguiría la luz focalizarse, y con más de 128.0 aquello sería una fiesta del brillo.

+ Moviendo y rotando luces:

Por último, para finalizar con las luces, ¿qué hace falta para mover o rotar una luz? Bueno pues nada especial, tan solo usar glTranslate*() y glRotate*(), antes de posicionar la luz. Ya que los cambios que hagamos a la matriz Modelview también afectan a los ejes de coordenadas de las luces posicionales.

5) Mezcla o blending:

La mezcla de colores o blending en inglés, nos permite añadir un efecto de transparencia, que resulta útil para simular agua, ventanas, cristal y otros objetos del mundo real a través de los cuales se puede ver. Para conseguirlo tendremos que hacer caso a la componente alpha, que contiene la información necesaria para este proceso.

Como primer detalle tendremos que llamar a glEnable(), activando el parametro GL_BLEND, para realizar las transparencias. Una vez activado el parametro, lo siguiente es llamar a glBlendFunc(), que por defecto tiene como fuente la función GL_ONE, y como destino la fuente GL_ZERO:

```
void glBlendFunc (GLenum source, GLenum destination);
+ source: La función que se le aplica a la superficie fuente.
   - GL_ZERO = El color fuente es igual a (0, 0, 0, 0).
   - GL_ONE = Usa el color actual de la fuente.
   - GL_DST_COLOR = Multiplica el color fuente por el color destino.
   - GL_ONE_MINUS_DST_COLOR = Multiplica el color fuente por el resultado
           de (1, 1, 1, 1) menos el color destino.
   - GL_SRC_ALPHA = Multiplica el color fuente por el alpha de la fuente.
   - GL_ONE_MINUS_SRC_ALPHA = Multiplica el color fuente por el resultado
           de (1, 1, 1, 1) menos el alpha de la fuente.
   - GL_DST_ALPHA = Multiplica el color fuente por el alpha del destino.
   - GL_ONE_MINUS_DST_ALPHA = Multiplica el color fuente por el resultado
           de (1, 1, 1, 1) menos el alpha del destino.
```

```
   - GL_SRC_ALPHA_SATURATE = Multiplica el color fuente por el mínimo
           entre la fuente y 1 menos el destino).
+ destination: La función aplicada a la superficie destino.
   - GL_ZERO = El color destino es igual a (0, 0, 0, 0).
   - GL_ONE = Usa el color actual del destino.
   - GL_SRC_COLOR = Multiplica el color destino por el color fuente.
   - GL_ONE_MINUS_SRC_COLOR = Multiplica el color destino por el resultado
           de (1, 1, 1, 1) menos el color fuente.
   - GL_SRC_ALPHA = Multiplica el color destino por el alpha de la fuente.
   - GL_ONE_MINUS_SRC_ALPHA = Multiplica el color destino por el resultado
           de (1, 1, 1, 1) menos el alpha de la fuente.
   - GL_DST_ALPHA = Multiplica el color destino por el alpha del destino.
   - GL_ONE_MINUS_DST_ALPHA = Multiplica el color destino por el resultado
           de (1, 1, 1, 1) menos el alpha del destino.
   - GL_SRC_ALPHA_SATURATE = Multiplica el color destino por el mínimo
           entre la fuente y 1 menos el destino).
```

Hay que recordar que cuanto más cercano a 0 es el alpha, menos opaco es el color, por consiguiente cuanto más cercano a 1 sea, más opaco será. Y a pesar de que hay muchas posibles combinaciones, para realizar transparencias lo más usado suele ser:

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

6) Transparencias:

Pero para garantizar transparencias en el mundo 3D, hace falta algunos detalles más, pues lo visto solo serviría para experimentos 2D bastante simples. Lo primero de todo es activar GL_DEPTH_TEST con glEnable(). Esto no es que sirva para la transparencia, sino para indicarle a OpenGL que no renderice los objetos que están ocultos tras otros objetos. Pero esto estropearía supuestamente cualquier transparencia, por ello primero hemos de pintar todos los objetos opacos, usando el buffer de profundidad de forma normal. Después hemos de usar la función glDepthMask(), para pasar el buffer de profundidad a modo lectura, y así salvar la información registrada al pintar los objetos opacos, para que al pintar los objetos transparentes, se realice correctamente la transparencia:

```
void glDepthMask (GLboolean flag);
+ flag: Cambia el modo de escritura del buffer de profundidad.
   - GL_TRUE  = Modo escritura/lectura.
   - GL_FALSE = Modo solo lectura.
```

Así que como ejemplo la cosa quedaría así:

```
void Render (void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    // Poner luces, pintar objetos sólidos de la escena...

    glEnable(GL_BLEND);
    glDepthMask(GL_FALSE);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE);

    // Dibujamos los objetos transparentes...

    glDepthMask(GL_TRUE);
```

```
    glDisable(GL_BLEND);

    glFlush();
    SwapBuffers(g_hDC);
}
```

Capítulo 7: Bitmaps e imagenes con OpenGL.

1) Los bitmaps en OpenGL:

Originariamente, los bitmaps para OpenGL son máscaras de un bit, que si vale un pixel 0 no se pinta, y si vale 1 se pinta con el color seleccionado con glColor*(). Que es un buen método para renderizar fuentes, pero también se pueden utilizar imagenes BMP o TGA.

+ Dando posición a un bitmap:

Para poder dar posición a un bitmap, usaremos la función glRasterPos*(), a la que le pasaremos unas coordenadas (x, y, z), que indicarán donde se va a posicionar la imagen en pantalla. Hay que recordar que el sistema de coordenadas de OpenGL no es como el de las típicas apis 2D, así que dicho punto representa la esquina inferior izquierda del bitmap, y no la esquina superior izquierda como venía siendo típico.

```
void glRasterPos[234][sifd]  (TYPE x, TYPE y, TYPE z, TYPE w);
void glRasterPos[234][sifd]v (TYPE * coords);
```

Así que normalmente usaremos cosas como glRasterPos2i(30, 10); para indicar la posición de imagenes que van a ser dibujadas. Para saber si la posición que le hemos indicado es válida, con pedir el parametro GL_CURRENT_RASTER_POSITION_VALID a la función glGetBooleanv(), nos devolverá GL_FALSE si es una posición no valida, o GL_TRUE en caso de serlo. Tampoco hay que olvidar que hemos de tener en cuenta, que las coordenadas irán en base al viewport y la proyección que hayamos indicado:

```
glViewport(0, 0, w, h);

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0f, w - 1.0f, 0.0f, h - 1.0f);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

+ Dibujando el bitmap:

Bien una vez determinada la posición en la que va a ser pintado, tendremos que pintarlo:

```
void glBitmap (GLsizei width, GLsizei height, GLfloat xOrigin,
               GLfloat yOrigin, GLfloat xIncrement, GLfloat yIncrement,
               const GLubyte * bitmap);
```

Los parametros de esta función son el ancho y alto del bitmap en cuestión. Luego unas coordenadas (xOrigin, yOrigin) relativas a la posición indicada en glRasterPos*(), por si queremos pintarlo en otras coordenadas a partir de la base. Después están unas coordenadas (xIncrement, yIncremente), que al terminar de pintar el bitmap, dichos

valores se sumarán a la posición base indicada en glRasterPos*(). Es decir que si queremos cambiar la posición base, después de pintar algo, no necesitamos llamar a glRasterPos*() de nuevo, sino pasar el valor incremental. Finalmente el último parametro es un puntero al bitmap que queremos pintar en pantalla.

2) Usando imagenes:

Pero claro, tener imagenes con pixels de un bit de profundidad, como que es un poco limitado. Por ello OpenGL también da soporte para pintar imagenes cuyos pixeles tengan una mayor profundidad de color (8, 16, 24, 32).

+ Pintando una imagen:

Una vez que tengamos los datos en un buffer en memoria, y hayamos llamado a glRasterPos*(), para indicar la posición final de la imagen, para pintarla usaremos:

```
void glDrawPixels (GLsizei width, GLsizei height, GLenum format,
                   GLenum type, const GLvoid * pixels);
+ width: Ancho de la imagen.
+ height: Alto de la imagen.
+ format: Formato del pixel.
   - GL_ALPHA       = Solo canal alpha.
   - GL_BGR         = Canales azul, verde y rojo.
   - GL_BGRA        = Canales azul, verde, rojo y alpha.
   - GL_COLOR_INDEX = Indice de color con paleta.
   - GL_BLUE        = Solo canal azul.
   - GL_GREEN       = Solo canal verde.
   - GL_RED         = Solo canal rojo.
   - GL_RGB         = Canales rojo, verde y azul.
   - GL_RGBA        = Canales rojo, verde, azul y alpha.
+ type: Tipo de cada componente del pixel.
   - GL_BITMAP         = Entero de 1-bit.
   - GL_BYTE           = Entero con signo de 8-bits.
   - GL_UNSIGNED_BYTE  = Entero sin signo de 8-bits.
   - GL_SHORT          = Entero con signo de 16-bits.
   - GL_UNSIGNED_SHORT = Entero sin signo de 16-bits.
   - GL_INT            = Entero con signo de 32-bits.
   - GL_UNSIGNED_INT   = Entero sin signo de 32-bits.
```

El último parametro es un puntero al buffer en memoria que tiene la imagen.

+ Leyendo de la pantalla:

¿Qué tendríamos que hacer para realizar una captura de la pantalla? Bien, para realizar eso tendríamos que hacer una "foto" de la pantalla con esta función:

```
void glReadPixels (GLint x, GLint y, GLsizei width, GLsizei height,
                   GLenum format, GLenum type, GLvoid * pixels);
```

Los argumentos son bien sencillos, los cuatro primeros sirven para definir el area que deseamos capturar. Los dos siguientes es el tipo de pixel que tendrá la imagen, para ser almacenado en el último. El buffer apuntado por pixels tiene que estár creado.

+ Copiando datos de la pantalla:

Además de poder pintar y leer en la pantalla, podemos copiar una región que nos apetezca. Para ello tenemos la función:

```
void glCopyPixels (GLint x, GLint y, GLsizei width, GLsizei height,
                   GLenum buffer);
+ buffer: Buffer del que vamos a copiar los pixels.
   - GL_COLOR   = Copiar del buffer de color.
   - GL_DEPTH   = Copiar del buffer de profundidad.
   - GL_STENCIL = Copiar del buffer de stencil.
```

Los parametros de la función son simples, hay que definir el area que deseamos copiar y de que buffer. ¿Pero y donde se pinta? Bien para poder indicarle donde la debe de pintar hay que usar la función glRasterPos*(), igual que hacíamos para pintar imagenes.

+ Agrandar, reducir y voltear:

Pintar y copiar está bien, pero falta algo de relativa utilidad, el poder pintar imagenes agrandándolas o reduciéndolas. Para ello se usa la función:

```
void glPixelZoom (GLfloat xZoom, GLfloat yZoom);
```

Por defecto los valores xZoom e yZoom valen ambos 1.0, por lo que 2.0 haría la imagen el doble de grande, y 0.5 la mitad de pequeña. ¿Pero qué pasa si pones un valor negativo? Bueno, si a xZoom se le pasa -1.0 se vería la imagen espejada horizontalmente, y si el -1.0 se le pasa a yZoom sería espejada verticalmente. Y lógicamente esta función para que surja efecto tiene que ser llamada antes de mandar a pintar cualquier imagen en pantalla que querramos modificar.

+ Manejando el almacenamiento de los pixels:

Por último tenemos una función para indicar a OpenGL como los datos de las imagenes están almacenados de forma empaquetada en memoria:

```
void glPixelStorei (GLenum pname, TYPE param);
+ pname:
   - GL_PACK_ALIGNMENT   = Como son los datos empaquetados en memoria.
   - GL_UNPACK_ALIGNMENT = Como se desempaquetan los datos de memoria.
+ param: 1, 2, 4 u 8.
```

Por defecto ambos parametros pack y unpack valen 4. Esto está pensado porque algunas arquitecturas, por ejemplo la Intel32, suele intentar alinear la memoria de 4 en 4 bytes, para poder mover la información más rápido de la memoria al micro. Así que poniendo algo como:

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
```

Le decimos a OpenGL que los pixels de la imagen están almacenados en "paquetes" de un byte, así que tiene que traer de memoria la información de un byte en un byte.

3) Los bitmaps de Windows:

Sabiendo como pintar imagenes, ahora solo nos falta saber cargarlas en memoria desde un fichero en el disco duro. Para ello primero usaremos BMPs.

+ El formato BMP:

La estructura interna de un fichero BMP consta de:

+ Una cabecera que identifica el formato.
+ Otra cabecera que da información sobre la imagen.
+ Una paleta opcional para el modo de 8bits.
+ Y los pixels de la imagen almacenados de modo BGR.

De hecho no solo el orden de los colores está invertido, sino que además la imagen está invertida de forma horizontal. Pero por suerte a OpenGL esto le viene bien, y no hace falta invertirla para pintarla en pantalla. La primera de las cabeceras es:

```
typedef struct tagBITMAPFILEHEADER
{
    WORD  bfType;      // Tipo del fichero, tiene que ser "BM" (0x4D42).
    DWORD bfSize;      // = sizeof(BITMAPFILEHEADER);
    WORD  bfReserved1; // Reservado, vale 0.
    WORD  bfReserved2; // Reservado, vale 0.
    DWORD bfOffBits;   // Offset de la imagen desde la cabecera.
} BITMAPFILEHEADER;
```

Es bastante simple, y lo único que hay que comprobar es que el tipo coincida con el número mágico para los bmps. Luego para posicionarnos en los pixels de la imagen, podremos usar el campo bOffBits, con la función fseek(). La otra cabecera es:

```
typedef struct tagBITMAPINFOHEADER
{
    DWORD biSize;          // = sizeof(BITMAPINFOHEADER);
    LONG  biWidth;         // Ancho de la imagen en pixels.
    LONG  biHeight;        // Alto de la imagen en pixels.
    WORD  biPlanes;        // Número de planos de color (= 1).
    WORD  biBitCount;      // Bits por pixel (1, 4, 8, 16, 24 o 32).
    DWORD biCompression;   // Tipo de compresión.
    DWORD biSizeImage;     // Tamaño de la imagen en bytes.
    LONG  biXPelsPerMeter; // Número de pixels por metro en el eje X.
    LONG  biYPelsPerMeter; // Número de pixels por metro en el eje Y.
    DWORD biClrUsed;       // Número de colores usados en el bitmap.
    DWORD biClrImportant;  // Número de colores que son importantes.
} BITMAPINFOHEADER;
```

Tampoco es nada del otro mundo, y en general solo usaremos los datos de la profundidad del pixel, el ancho y alto de la imagen, y el tamaño de esta. Pero el campo biCompression, sirve para informarnos de si la imagen está comprimida o no, para comprobar que no está comprimida tendríamos que compararlo con BI_RGB.

+ Cargando un fichero BMP:

La función para cargar una imagen BMP bien podría ser como esta:

```
unsigned char * LoadBitmapFile (char * name, BITMAPINFOHEADER * info)
{
    FILE *           file;
    BITMAPFILEHEADER header;
    unsigned char *  buffer;
    DWORD            imgidx;
    unsigned char    temprgb;
```

```
    // Abrimos el fichero.
    file = fopen(name, "rb");
    if(file == NULL) return NULL;

    // Leemos la cabecera y comprobamos que sea un bmp.
    fread(&header, sizeof(BITMAPFILEHEADER), 1, file);

    if(header.bfType != 0x4D42)
    {
        fclose(file);
        return NULL;
    }

    // Leemos la otra cabecera y comprobamos que no está comprimido.
    fread(info, sizeof(BITMAPINFOHEADER), 1, file);

    if(info->biCompression != BI_RGB)
    {
        fclose(file);
        return NULL;
    }

    // Movemos el puntero a la sección con los pixels.
    fseek(file, header.bfOffBits, SEEK_SET);

    // Pedimos memoria para el buffer de la imagen.
    buffer = (unsigned char *) malloc (info->biSizeImage);

    if(buffer == NULL)
    {
        fclose(file);
        return NULL;
    }

    // Leemos el contenido de la imagen.
    fread(buffer, 1, info->biSizeImage, file);

    // Y pasamos los pixels de modo BGR a RGB.
    for(imgidx = 0; imgidx < info->biSizeImage; imgidx += 3)
    {
        temprgb = buffer[imgidx];
        buffer[imgidx] = buffer[imgidx + 2];
        buffer[imgidx + 2] = temprgb;
    }

    // Cerramos el fichero y devolvemos el buffer.
    fclose(file);
    return buffer;
}
```

Y con este ejemplo se ve como funciona la función y como pintar el bmp:

```
BITMAPINFOHEADER ImagenInfo;
unsigned char *  ImagenData;

ImagenData = LoadBitmapFile("test.bmp", & ImagenInfo);
glPixelStorei(GL_UNPACK_ALIGNMENT, 4);
glRasterPos2i(100, 100);
glDrawPixels(ImagenInfo.biWidth, ImagenInfo.biHeight, GL_RGB,
             GL_UNSIGNED_BYTE, ImagenData);
```

+ Salvando un fichero BMP:

Y para salvar un BMP tendríamos esta otra función:

```
int SaveBitmapFile (char * name, int w, int h, unsigned char * data)
{
    FILE *           file;
    BITMAPFILEHEADER header;
    BITMAPINFOHEADER info;
    DWORD            imgidx;
    unsigned char    temprgb;

    // Abrimos el fichero.
    file = fopen(name, "wb");
    if(file == NULL) return 0;

    // Configuramos la cabecera del bmp.
    header.bfType      = 0x4D42;
    header.bfSize      = sizeof(BITMAPFILEHEADER);
    header.bfReserved1 = 0;
    header.bfReserved2 = 0;
    header.bfOffBits   = sizeof(BITMAPFILEHEADER) +
                         sizeof(BITMAPINFOHEADER);

    // Configuramos la cabecera de información.
    info.biSize          = sizeof(BITMAPINFOHEADER);
    info.biWidth         = w;
    info.biHeight        = h;
    info.biPlanes        = 1;
    info.biBitCount      = 24;
    info.biCompression   = BI_RGB;
    info.biSizeImage     = w * h * 3;
    info.biXPelsPerMeter = 0;
    info.biYPelsPerMeter = 0;
    info.biClrUsed       = 0;
    info.biClrImportant  = 0;

    // Pasamos los pixels de modo RGB a BGR.
    for(imgidx = 0; imgidx < info.biSizeImage; imgidx += 3)
    {
        temprgb = data[imgidx];
        data[imgidx] = data[imgidx + 2];
        data[imgidx + 2] = temprgb;
    }

    // Y escribimos el contenido del fichero.
    fwrite(&header, sizeof(BITMAPFILEHEADER), 1, file);
    fwrite(&info, sizeof(BITMAPINFOHEADER), 1, file);
    fwrite(data, 1, info.biSizeImage, file);

    // Cerramos el fichero y devolvemos que todo ha ido bien.
    fclose(file);
    return 1;
}
```

Y la forma de usarla para capturar la pantalla sería por ejemplo:

```
void SaveScreenshot (int w, int h)
{
    unsigned char * data = (unsigned char *) malloc (w * h * 3);
```

```
    memset(data, 0, w * h * 3);

    glReadPixels(0, 0, w, h, GL_RGB, GL_UNSIGNED_BYTE, data);

    SaveBitmapFile("pantalla.bmp", w, h, data);

    free(data);
}
```

4) Las imagenes targa:

Los bmps no están mal, pero no estaría mal poder tener también imagenes con transparencias, es decir imagenes con canal alpha. Y por eso se creó el formato TGA.

+ El formato TGA:

El formato TGA es un poco más sencillo que el BMP, pues tan solo tiene dos partes, la cabecera y los datos. Y la cabecera está formada por los siguientes datos:

```
typedef struct tagTARGAFILEHEADER
{
   BYTE      imageIDLength;     // N° de caracteres del campo de ID (=0)
   BYTE      colorMapType;      // = 0;
   BYTE      imageTypeCode;     // = 2; (RGB no comprimido)
                                // = 3; (Escala de grises no comprimida)
   short int colorMapOrigin;    // = 0;
   short int colorMapLength;    // = 0;
   short int colorMapEntrySize; // = 0;
   short int imageXOrigin;      // = 0;
   short int imageYOrigin;      // = 0;
   short int imageWidth;        // Ancho en pixeles de la imagen.
   short int imageHeight;       // Alto en pixeles de la imagen.
   BYTE      bitCount;          // Profundidad de color (16, 24, 32)
   BYTE      imageDescriptor;   // 24bits = 0x00, 32bits = 0x08
} TARGAFILEHEADER;
```

Realmente la cabecera no tiene mucho que explicar, y hay más valores para el campo imageTypeCode, pero normalmente siempre se usará el valor 2. Así que para almacenar una imagen TGA podríamos hacerlo en una estructura como:

```
typedef struct
{
    BYTE      imageTypeCode;
    short int imageWidth;
    short int imageHeight;
    BYTE      bitCount;
    BYTE *    imageData;
} TGAFILE;
```

+ Cargando un fichero TGA:

Así que ahora veamos como se carga un fichero TGA:

```
int LoadTGAFile (char * name, TGAFILE * tgaFile)
{
    FILE * file;
```

```
DWORD imgidx, imagesize;
BYTE temprgb, colormode;
TARGAFILEHEADER header;

// Abrimos el fichero.
file = fopen(name, "rb");
if(file == NULL) return 0;

// Leemos el tipo de fichero que es.
fread(&(header.imageIDLength), 1, 1, file);
fread(&(header.colorMapType),  1, 1, file);
fread(&(header.imageTypeCode), 1, 1, file);

// Comprobamos que es uno válido.
if((header.imageTypeCode != 2) && (header.imageTypeCode != 3))
{
    fclose(file);
    return 0;
}

// Y leemos el resto de parametros de la cabecera.
fread(&(header.colorMapOrigin),    2, 1, file);
fread(&(header.colorMapLength),    2, 1, file);
fread(&(header.colorMapEntrySize), 1, 1, file);

fread(&(header.imageXOrigin),    2, 1, file);
fread(&(header.imageYOrigin),    2, 1, file);
fread(&(header.imageWidth),      2, 1, file);
fread(&(header.imageHeight),     2, 1, file);
fread(&(header.bitCount),        1, 1, file);
fread(&(header.imageDescriptor), 1, 1, file);

// Salvamos la información necesaria en la estructura TGAFILE.
tgaFile->imageTypeCode = header.imageTypeCode;
tgaFile->imageWidth    = header.imageWidth;
tgaFile->imageHeight   = header.imageHeight;
tgaFile->bitCount      = header.bitCount;

// Calculamos el tamaño en bytes de un pixel y el de la imagen.
colormode = tgaFile->bitCount / 8;
imagesize = tgaFile->imageWidth * tgaFile->imageHeight * colormode;

// Pedimos memoria para el buffer de la imagen.
tgaFile->imageData = (BYTE *) malloc(imagesize);

if(tgaFile->imageData == NULL)
{
    fclose(file);
    return 0;
}

// Leemos el contenido de la imagen.
fread(tgaFile->imageData, sizeof(BYTE), imagesize, file);

// Y pasamos los pixels de modo BGR a RGB.
for(imgidx = 0; imgidx < imagesize; imgidx += colormode)
{
    temprgb = tgaFile->imageData[imgidx];
    tgaFile->imageData[imgidx] = tgaFile->imageData[imgidx + 2];
    tgaFile->imageData[imgidx + 2] = temprgb;
}
```

```
    // Cerramos el fichero y devolvemos que todo ha ido bien.
    fclose(file);
    return 1;
}
```

Te preguntarás porque no leemos con fread la cabecera entera, que sería más comodo. Bueno resulta que los compiladores de hoy en día tienen la manía de alinear las estructuras que creamos, y aunque son 18bytes lo que tenemos que leer, el compilador ha decidido que la estructura ocupa 20 porque le viene mejor. Por ello es más seguro este otro modo, lo cual no quiere decir que pudiera funcionar del otro modo si el compilador hiciera, lo que desearíamos que hiciera pero posiblemente no haga. El caso es que la forma de usar todo esto para cargar una imagen con transparencia sería:

```
TGAFILE imagen;
LoadTGAFile("test.tga", &imagen);

glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

glPixelStorei(GL_UNPACK_ALIGNMENT, 4);
glRasterPos2i(0, 0);
glDrawPixels(imagen.imageWidth, imagen.imageHeight, GL_RGBA,
             GL_UNSIGNED_BYTE, imagen.imageData);
```

+ Salvando un fichero TGA:

Y finalmente para salvar un TGA tan solo tenemos que hacer:

```
int SaveTGAFile (char * name, short int w, short int h, byte bpp,
                 unsigned char * data)
{
    FILE * file;
    DWORD imgidx, imagesize;
    BYTE temprgb, colormode;
    TARGAFILEHEADER header;

    // Abrimos el fichero.
    file = fopen(name, "wb");
    if(file == NULL) return 0;

    // Configuramos los valores de la cabecera.
    header.imageIDLength = 0;
    header.colorMapType  = 0;
    header.imageTypeCode = 2;

    header.colorMapOrigin    = 0;
    header.colorMapLength    = 0;
    header.colorMapEntrySize = 0;

    header.imageXOrigin    = 0;
    header.imageYOrigin    = 0;
    header.imageWidth      = w;
    header.imageHeight     = h;
    header.bitCount        = bpp;

    if(bpp == 32)
        header.imageDescriptor = 0x08;
    else
```

```
        header.imageDescriptor = 0x00;

    // Escribimos la cabecera.
    fwrite(&(header.imageIDLength), 1, 1, file);
    fwrite(&(header.colorMapType),  1, 1, file);
    fwrite(&(header.imageTypeCode), 1, 1, file);

    fwrite(&(header.colorMapOrigin),    2, 1, file);
    fwrite(&(header.colorMapLength),    2, 1, file);
    fwrite(&(header.colorMapEntrySize), 1, 1, file);

    fwrite(&(header.imageXOrigin),    2, 1, file);
    fwrite(&(header.imageYOrigin),    2, 1, file);
    fwrite(&(header.imageWidth),      2, 1, file);
    fwrite(&(header.imageHeight),     2, 1, file);
    fwrite(&(header.bitCount),        1, 1, file);
    fwrite(&(header.imageDescriptor), 1, 1, file);

    // Calculamos el tamaño en bytes de un pixel y el de la imagen.
    colormode = bpp / 8;
    imagesize = w * h * colormode;

    // Y pasamos los pixels de modo BGR a RGB.
    for(imgidx = 0; imgidx < imagesize; imgidx += colormode)
    {
        temprgb = data[imgidx];
        data[imgidx] = data[imgidx + 2];
        data[imgidx + 2] = temprgb;
    }

    // Escribimos el contenido de la imagen.
    fwrite(data, sizeof(unsigned char), imagesize, file);

    // Cerramos el fichero y devolvemos que todo ha ido bien.
    fclose(file);
    return 1;
}
```

Para salvar una captura de la pantalla en formato TGA sería igual que con el BMP, pero usando esta función en vez de la que se usaría para salvar un BMP.

Capítulo 8: Mapeado de texturas.

1) Introducción al mapeado de texturas:

En este capítulo aprenderemos a cargar y usar texturas con los objetos de nuestra escena, para aumentar el realismo de lo que dibujamos en pantalla. Normalmente se trabaja con texturas de dos dimensiones (que tienen ancho y alto), pero también existen texturas de una y tres dimensiones (estas últimas también tienen profundidad).

La forma de trabajar es cargar las imagenes que vayamos a usar como texturas, reservar un indentificador para cada una, configurar los parametros asociados, cargar en la memoria de la tarjeta de video la textura, y asociarla a cada objeto que pintemos. En un ejemplo básico en el que solo tuvieramos una textura y un objeto haríamos algo como:

```
unsigned int texture;
```

```
glEnable(GL_TEXTURE_2D);

// Creamos el objeto textura.
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);

// Lo configuramos.
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

// Y cargamos la textura en la memoria de la tarjeta.
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, ImagenInfo.biWidth,
             ImagenInfo.biHeight, 0, GL_RGB, GL_UNSIGNED_BYTE, Imagen);
```

2) Aprendiendo a cargar las texturas:

+ Texturas 2D:

Una vez tengamos creada la textura en OpenGL después de llamar a glBindTexture(), para cargar una de dos dimensiones utilizaremos la función:

```
void glTexImage2D (GLenum target, GLint level, GLint internalformat,
                   GLsizei width, GLsizei height, GLint border,
                   GLenum format, GLenum type, const GLvoid * texels);
+ target: GL_TEXTURE_2D o GL_PROXY_TEXTURE_2D son los valores admitidos
   por la función, y normalmente será GL_TEXTURE_2D el que usemos.
+ level: Este parametro sirve para indicar a que nivel de detalle
   pertencerá esta textura, si no fueramos a usar el mipmapping o nivel
   de detalle para la textura, este parametro valdrá cero.
+ internalformat: Esta argumento indica el formato interno en el que los
   texels serán guardados en la tarjeta, y hay múltiples valores, pero
   habitualmente se usan: GL_LUMINANCE, GL_LUMINANCE_ALPHA, GL_RGB y
   GL_RGBA.
+ width: El ancho de la imagen que estamos cargando.
+ height: El alto de la imagen que estamos cargando.
+ border: Indica si la textura tiene borde (= 1) o no (= 0).
+ format: Formato de los pixel de la imagen que estamos cargando, entre
   los valores soportados están: GL_COLOR_INDEX, GL_LUMINANCE, GL_RGB,
   GL_RGBA, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA y GL_LUMINANCE_ALPHA.
+ type: Indica el número de bytes por cada componente del formato de
   pixel: GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT,
   GL_INT, GL_UNSIGNED_INT, GL_FLOAT, GL_BITMAP,
```

Un detalle para tarjetas que no soportan OpenGL 2.0, es que el ancho y el alto de las texturas que deseamos cargar, tienen que ser potencia de 2, es decir: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, etcétera... El último parametro de la función, texels es el puntero al buffer donde está almacenada la textura en memoria.

+ Texturas 1D:

Las texturas de una dimensión tienen ancho, pero no alto (que sería 1 pixel de altura solamente). Y para poder cargar una hay que usar la función:

```
void glTexImage1D (GLenum target, GLint level, GLint internalformat,
                   GLsizei width, GLint border, GLenum format,
                   GLenum type, const GLvoid * texels);
```

Los parametros de la función son los mismos que la anterior, salvo por la salvedad de que height ya no está como uno de ellos.

+ Texturas 3D:

Para poder cargar una textura tridimensional se utiliza la función:

```
void glTexImage3D (GLenum target, GLint level, GLint internalformat,
                   GLsizei width, GLsizei height, GLsizei depth,
                   GLint border, GLenum format, GLenum type,
                   const GLvoid * texels);
```

Como con las texturas 2D los parametros son los mismos, solo que a este tipo de textura se le añade un nuevo parametro depth, que indica la profundidad de la textura. El problema de estas texturas, es que a pesar del fabuloso efecto gráfico que producen, consumen una cantidad de memoria abismal, y ello hace que no se usen demasiado.

3) Los objetos textura:

Los "objetos" textura, no son objetos como lo que se entiende en C++, por ejemplo. Con las funciones anteriores podemos trabajar, pero tienen un serio problema, y es que si tenemos varias texturas, cada vez que querramos usar una determinada, tendríamos que estar cargándolas en la memoria de la tarjeta constantemente, y eso no sería muy provechoso. Así que OpenGL ofrece un sistema para poder referenciar las texturas ya cargadas en memoria, modificar sus propiedades e invocarlas al renderizar.

+ Generando un nombre para la textura:

El primer paso para poder referenciar una textura es darle nombre, solo que en OpenGL el "nombre" es un número entero sin signo distinto de cero, por ejemplo: 1, 37, 42, 1000. Para poder garantizar que no estamos usando el mismo nombre para una misma textura existe una función que nos devuelve el primer nombre libre que encuentra:

```
void glGenTextures (GLsizei n, GLuint * texnames);
```

El primer parametro n, indica el número de nombres de texturas que queremos generar. Y el segundo es donde van a ser almacenados esos nombres, por ello texnames tiene que ser un array de tamaño igual o mayor que el valor que le pasemos a n.

+ Creando y usando objetos textura:

Con el nombre obtenido, tenemos que crear el objeto en la tarjeta, para ello se utilizará la función:

```
void glBindTexture (GLenum target, GLuint texname);
+ target: Tipo de textura que deseamos crear o invocar.
   - GL_TEXTURE_1D = Textura de una dimensión.
   - GL_TEXTURE_2D = Textura de dos dimensiones.
   - GL_TEXTURE_3D = Textura de tres dimensiones.
+ texname: Nombre de la textura a crear o invocar.
```

Así que llamándola por primera vez se creará un objeto textura, del tipo que le hayamos indicado en target, con todos los valores asociados por defecto. ¿Qué pasa si se vuelve a llamar? Pues que activa la textura asociada al nombre que le pasamos, como la

textura actual a utilizar. Esto viene bien para cambiar parametros, o simplemente para cuando deseamos pintar varios objetos en la escena, con texturas distintas cada uno.

4) El filtrado en las texturas:

Un punto peliagudo en el mapeado de las texturas es qué pasa cuando un texel de la textura, ocupa menos o más que un pixel de la pantalla. En estos casos OpenGL realiza un filtrado para calcular que es lo que tendría que pintar en ese texel. Cuando el pixel es menor que el texel se le denomina magnificación (la textura está agrandada), y cuando el pixel es mayor que el texel se le llama minificación (la textura está empequeñecida). Y esto viene a que hay una función para determinar como tiene OpenGL que actuar frente a esas situaciones:

```
void glTexParameteri (GLenum target, GLenum pname, GLint param);
+ target: GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D.
+ pname:
   - GL_TEXTURE_MAG_FILTER = Configurar el filtrado para la magnificación.
   - GL_TEXTURE_MIN_FILTER = Configurar el filtrado para la minificación.
+ param:
   - GL_NEAREST = Usa el texel más cercano al centro del pixel, que está
      siendo texturizado.
   - GL_LINEAR = Usa una interpolación lineal (una media ponderada) con
      los cuatro texels, que están más cerca del centro del pixel que está
      siendo texturizado.
   - GL_NEAREST_MIPMAP_NEAREST = Usa la imagen más cercana a la resolución
      del polígono y luego le aplica un filtrado de tipo GL_NEAREST.
   - GL_NEAREST_MIPMAP_LINEAR = Usa la imagen más cercana a la resolución
      del polígono y luego le aplica un filtrado de tipo GL_LINEAR.
   - GL_LINEAR_MIPMAP_NEAREST = Usa una interpolación lineal entre los dos
      mipmaps más cercanos a la resolución del polígono y luego le aplica
      un filtrado de tipo GL_NEAREST.
   - GL_LINEAR_MIPMAP_LINEAR = Usa una interpolación lineal entre los dos
      mipmaps más cercanos a la resolución del polígono y luego le aplica
      un filtrado de tipo GL_LINEAR.
```

Las opciones mipmap sirven para el filtrado cuando se ha activado texturas con nivel de detalle, y se comentarán más seriamente en el apartado dedicado a ese tema.

5) Funciones de texturas:

OpenGL ofrece la opción de aplicar una función a las texturas al ser mapeadas, para determinar cual será su comportamiento respecto al color, para ello hay que utilizar la función siguiente:

```
void glTexEnvi (GLenum target, GLenum pname, GLint param);
+ target: GL_TEXTURE_ENV.
+ pname: GL_TEXTURE_ENV_MODE.
+ param:
   - GL_BLEND = El color de la textura es multiplicado por el color de
      pixel y combinado con un color constante.
   - GL_DECAL = La textura reemplaza a los pixels existentes.
   - GL_MODULATE = El color de la textura es multiplicado por el color de
      pixel actualmente seleccionado.
```

Existen más valores para target y pname, pero para modificar el comportamiento del color al ser mapeadas las texturas, usaremos los valores ahí expuestos. El valor por defecto es GL_MODULATE, y por ello cuando se vaya a poner una textura a un objeto, es recomendable que el color de relleno sea el blanco, ya que ese color es el elemento neutro en la multiplicación (1.0, 1.0, 1.0, 1.0).

6) Las coordenadas de una textura:

Al renderizar la escena, en cada polígono hay que indicar las coordenadas (s, t) de la textura, que van desde 0 hasta 1 en cada eje, para cada vértice del polígono. Para ello existe la función:

```
void glTexCoord2f  (GLfloat s, GLfloat t);
void glTexCoord2fv (const GLfloat * coords);
```

De este modo las coordenadas (s, t) equivale la (0, 0) a la esquina inferior izquierda, la (1, 0) a la esquina inferior derecha, la (1, 1) a la esquina superior derecha, y la (0, 1) a la esquina superior izquierda. Y para ver como funciona un ejemplo:

```
glBegin(GL_QUADS);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(0.0f, 0.0f, 0.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f(5.0f, 0.0f, 0.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f(5.0f, 5.0f, 0.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(0.0f, 5.0f, 0.0f);
glEnd();
```

Pero los valores (s, t) podemos modificarlos y poner los que nos de la gana, para poder aplicar una sola textura a un objeto formado de multiples polígonos, de tal forma que parezca que ha sido mapeado por una sola textura. Que es lo que ocurre con los modelos animados del Quake e incluso el escenario.

7) Repetición y estiramiento:

Por defecto, cuando una textura es más pequeña que el polígono en el que es mapeada, la textura es repetida para cubrir el hueco "vacío". Pero esto puede ser modificado con la función glTexParameteri():

```
void glTexParameteri (GLenum target, GLenum pname, GLint param);
+ target: GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D.
+ pname: GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T.
+ param: GL_REPEAT, GL_CLAMP.
```

Como parametro target podemos indicarle de que tipo de textura se trata, luego con pname podemos decirle en que eje, si el s o el t, queremos cambiar el comportamiento. Comportamiento que por defecto toma el valor de GL_REPEAT, pero poniendole GL_CLAMP en param, queda un efecto bastante curioso, que consiste en que se pinta la textura y el hueco vacío es rellenado con el último pixel en el borde.

8) Mipmaps y niveles de detalle:

Y por fin un tema bastante importante, los mipmpas y los niveles de detalle. Para agilizar cálculos y para mejorar la calidad del renderizado, es recomendable para las texturas que

tengan varios niveles de detalle, para poder elegir el que más se ajuste a la situación. Ello ahorra cálculos, y en la mayoría de las veces esos cálculos no quedan del todo bien.

Para realizar esta técnica hemos de configurar el filtrado de la textura y cargar varios niveles de la textura, con imagenes a diferentes resoluciones. Por ejemplo:

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

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 64, 64, 0, GL_RGB,
             GL_UNSIGNED_BYTE, texImage0);
glTexImage2D(GL_TEXTURE_2D, 1, GL_RGB, 32, 32, 0, GL_RGB,
             GL_UNSIGNED_BYTE, texImage1);
glTexImage2D(GL_TEXTURE_2D, 2, GL_RGB, 16, 16, 0, GL_RGB,
             GL_UNSIGNED_BYTE, texImage2);
glTexImage2D(GL_TEXTURE_2D, 3, GL_RGB,  8,  8, 0, GL_RGB,
             GL_UNSIGNED_BYTE, texImage3);
glTexImage2D(GL_TEXTURE_2D, 4, GL_RGB,  4,  4, 0, GL_RGB,
             GL_UNSIGNED_BYTE, texImage4);
glTexImage2D(GL_TEXTURE_2D, 5, GL_RGB,  2,  2, 0, GL_RGB,
             GL_UNSIGNED_BYTE, texImage5);
glTexImage2D(GL_TEXTURE_2D, 6, GL_RGB,  1,  1, 0, GL_RGB,
             GL_UNSIGNED_BYTE, texImage6);
```

Así es como podemos realizar esta técnica, pero claro esto tiene un problema, que tendríamos que crear por cada textura varias alternativas de menor resolución. Por suerte hay una función que las puede crear automáticamente:

```
int gluBuild2DMipmaps (GLenum target, GLint internalFormat, GLint width,
                       GLint height, GLenum format, GLenum type,
                       void * texels);
```

Recibe prácticamente los mismos parametros que glTextImage2D(), y lo que hace es generar de forma interna los niveles de detalle, a partir de la textura que le pasemos en texels (posiblemente va dividiendo entre 2 hasta que llegue a 1x1 pixel). Así que el ejemplo anterior quedaría en:

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

gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, 64, 64, GL_RGB,
                  GL_UNSIGNED_BYTE, texImage0);
```

9) Ejemplos del libro:

+ La bandera ondulante: Páginas de la 248 a la 263.
+ Un terreno con alturas: Páginas de la 263 a la 279.

Capítulo 9: Mapeado de texturas avanzado.

1) Multitexturizado:

Bien el primer efecto avanzado de mapeado de texturas es el multitexturizado o multitexturing en inglés. Dicha técnica consiste en aplicar varias texturas a un mismo polígono. En esta técnica a las texturas se las llama unidades. Y para utilizar este método hay que efectuar una serie de pasos:

+ Comprobar que este efecto está soportado por la tarjeta.
+ Localizar el puntero de las funciones que intervienen.
+ Establecer las unidades.
+ Y especificar las coordenadas de las texturas.

+ Verificando el soporte para el multitexturizado:

Como se ha indicado en la introducción, hay que comprobar si la tarjeta de nuestro equipo soporta el multitexturizado, y para ello tenemos en OpenGL las extensiones, que es el modo en el que se actualiza esta api. Lo primero es pedir una lista de las extensiones soportadas en la tarjeta con:

```
const GLubyte * glGetString (GLenum pname);
+ pname: Parametro que deseamos obtener.
   - GL_VENDOR = Devuelve quien a sido el responsable de implementar el
     driver que estás usando de OpenGL. El nombre este no cambia de una
     versión a otra, normalmente.
   - GL_RENDERER = Devuelve el nombre del renderizador, que es un nombre
     específico de una configuración de hardware determinada. Tampoco
     cambia el nombre de una versión a otra.
   - GL_VERSION = Devuelve el número de versión.
   - GL_EXTENSIONS = Devuelve la lista de extensiones soportadas, en una
     cadena de caracteres separada por espacios.
```

Así que usaremos el parametro GL_EXTENSIONS para buscar en la lista que esté el nombre GL_ARB_multitexture. En caso de encontarse en la lista, significa que la extensión está soportada y podemos usar el efecto. ¿Pero como buscamos el valor dentro de la lista? Bueno OpenGL tiene una función para ayudarnos con eso:

```
GLboolean gluCheckExtension (char * extName, const GLubyte * extString);
```

Que devolverá GL_TRUE, si extName es encontrado dentro de extString. Otra forma, en caso de que gluCheckExtension() no esté soportada en nuestra versión de OpenGL, sería usar la función del ANSI C strstr(), que hace prácticamente lo mismo:

```
char * extensiones = (char *) glGetString(GL_EXTENSIONS);

//(gluCheckExtension("GL_ARB_multitexture", extensiones))
if(strstr(extensiones, "GL_ARB_multitexture") != NULL)
{
   cout << "Felicidades, el multitexturizado está soportado..." << endl;
}
```

Por cierto que glGetString() no puede ser llamada antes de inicializar OpenGL, creando al menos el contexto de renderizado, para que sea la librería inicializada.

+ Como acceder a las funciones de las extensiones:

Bien, una vez que hemos comprobado que tenemos el efecto disponible en nuestra tarjeta, lo siguiente es obtener las direcciones de las funciones que usaremos para el

multitexturizado con la función wglGetProcAddress(), que busca en el driver cargado de OpenGL la función que le pasamos con una cadena de texto:

```
PROC wglGetProcAddress (LPCSTR lpszProc);
```

Las funciones que vamos a necesitar cargar son las siguientes:

+ **glMultiTexCoord**[**1234**]**fARB**: Esta función sirve para indicar las coordenadas de la textura para la multitexturación.
+ **glActiveTextureARB**: Establece cual es la textura unidad actual.
+ **glClientActiveTextureARB**: Sirve para lo mismo que la anterior, pero se utiliza cuando utilizamos arrays de vértices y demás.

Así que el código para poder cargar las funciones sería el siguiente:

```
PFNGLMULTITEXCOORD2FARBPROC     glMultiTexCoord2fARB     = NULL;
PFNGLACTIVETEXTUREARBPROC       glActiveTextureARB       = NULL;
PFNGLCLIENTACTIVETEXTUREARBPROC glClientActiveTextureARB = NULL;

glMultiTexCoord2fARB = (PFNGLMULTITEXCOORD2FARBPROC)
  wglGetProcAddress("glMultiTexCoord2fARB");

glActiveTextureARB = (PFNGLACTIVETEXTUREARBPROC)
  wglGetProcAddress("glActiveTextureARB");

glClientActiveTextureARB = (PFNGLCLIENTACTIVETEXTUREARBPROC)
  wglGetProcAddress("glClientActiveTextureARB");
```

Por cierto hay un tema interesante que hay que tener en cuenta para la hora de trabajar con extensiones, y es que debemos bajarnos el fichero glext.h de la página de www.opengl.org para poder trabajar de forma "cómoda" con extensiones. Ya que por defecto las cabeceras de OpenGL que vienen con los compiladores de windows no traen las definiciones de tipos de punteros a funciones.

+ Estableciendo las unidades de texturas:

Bien ahora que tenemos las funciones, y que supuestamente también tendremos las texturas cargadas en memoria de la tarjeta, vamos a asignarle una posición o unidad. Aunque antes que todo ello, aprenderemos a preguntar cuantas capas máximo podremos utilizar en la tarjeta que estemos usando:

```
int maxTexUnits;
glGetIntegerv(GL_MAX_TEXTURE_UNITS_ARB, &maxTexUnits);
```

Bien ahora que podremos saber el número máximo de capas, aprenderemos a usar la función que asigna una textura a una unidad:

```
void glActiveTextureARB (GLenum texUnit);
+ textUnit = GL_TEXTURE#_ARB (# = 0 .. maxTexUnits-1)
```

Esta función sirve para activar una unidad de textura, la que le indiquemos en texUnit. Después de activar una unidad hemos de llamar a glEnable(GL_TEXTURE_2D), y después a glBindTexture(), para asociar una textura ya cargada en memoria con la unidad. Hay que advertir que al crear una unidad es solo una especie de referencia para realizar el multitexturizado, por lo que si hacemos cambios a la textura asociada, será porque se los estamos haciendo a la propia textura:

```
glActiveTextureARB(GL_TEXTURE0_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, texture1);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);

glActiveTextureARB(GL_TEXTURE1_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, texture2);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
```

En este ejemplo hemos creado dos unidades, la cero y la uno, con las texturas texture1 y texture2. Hay un detalle importante que señalar, el orden de pintado de las unidades es GL_TEXTURE0_ARB la capa más "externa" del polígono, y GL_TEXTURE31_ARB la más interna posible. Es decir que hemos de asignar las texturas de la exterior a la interior. Por cierto es recomendable definir las unidades cuando vayamos a pintar el objeto, y no al inicio del programa, ya que por la arquitectura de OpenGL eso acarrearía problemas de tal modo que se harían las cosas tal que:

+ Cargar y configurar las texturas en la tarjeta al "inicio" del programa.
+ Y en la función de renderizado:
  - Definir las unidades.
  - Pintar el objeto u objetos que usen las unidades.
  - Borrar las unidades definidas.

De este modo así tenemos la certeza de que las cosas funcionarán relativamente bien. Para borrar las unidades definidas tan solo hay que hacerlo al reves de como han sido definidas:

```
glActiveTextureARB(GL_TEXTURE1_ARB);
glDisable(GL_TEXTURE_2D);
glActiveTextureARB(GL_TEXTURE0_ARB);
glDisable(GL_TEXTURE_2D);
```

De este modo si alguna textura que hayamos usado para una unidad la queremos usar para un objeto haremos lo siguiente:

```
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, texture2);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

glBegin(GL_QUADS);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-100.0f, -100.0f, -50.0f);
    glTexCoord2f(1.5f, 0.0f); glVertex3f(-50.0f, -100.0f, -50.0f);
    glTexCoord2f(1.5f, 1.5f); glVertex3f(-50.0f, -50.0f, -50.0f);
    glTexCoord2f(0.0f, 1.5f); glVertex3f(-100.0f, -50.0f, -50.0f);
glEnd();

// Pintar otras cosas con otras texturas...

glDisable(GL_TEXTURE_2D);
```

La idea de volver a configurar la función de tratamiento del color al mapear la textura es porque de no invocarla, las mezclas producidas al pintar con el multitexturizado se mantendrían presentes al mapear la textura. Y es por ello que se le vuelve a configurar ese parametro para que haga exactamente lo que nosotros queremos. Y el comentario del ejemplo, hace referencia a que podemos usar invocar otras texturas para pintar otros

objetos, sin tener que activar y desactivar el parametro GL_TEXTURE_2D. Tan solo tendremos que hacer ese apaño, antes y después de pintar algo con multitexturizado, para evitar problemas en la máquina de estados de OpenGL.

+ Especificando las coordenadas de las texturas:

Visto el peliagudo tema de configurar la unidades, el indicar las coordenadas de estas es bastante más sencillo, y para mostrarlo un ejemplo, que iría entre la definición y el borrado de las unidades del multitexturado:

```
glBegin(GL_QUADS);
    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0f, 0.0f);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0.0f, 0.0f);
    glVertex3f(-50.0f, -50.0f, -50.0f);

    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 2.0f, 0.0f);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 2.0f, 0.0f);
    glVertex3f(50.0f, -50.0f, -50.0f);

    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 2.0f, 2.0f);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 2.0f, 2.0f);
    glVertex3f(50.0f, 50.0f, -50.0f);

    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0f, 2.0f);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0.0f, 2.0f);
    glVertex3f(-50.0f, 50.0f, -50.0f);
glEnd();
```

Bastante simple, la verdad. Y todo con tan solo utilizar la función:

```
void glMultiTexCoord2fARB (GLenum texUnit, GLfloat s, GLfloat t);
```

Existen otras versiones de esta función con 1, 3 y 4 argumentos, pero la más usada suele ser esta, ya que lo más usado suelen ser las texturas de dos dimensiones.

+ Poniéndolo todo junto:

(Páginas de la 288 a la 298).

2) Mapping envolvente o environment mapping:

Esta técnica es normalmente utilizada para renderizar cosas que "reflejan" lo que les rodea. El truco está en que tenemos de fondo una imagen de las montañas en una textura, y luego en otra textura tenemos esa imagen, pero modificada con un filtro de "ojo de pez" que le llaman en inglés. Es un tipo de deformación que nos vendrá bien para la situación. Así que ponemos el fondo, y al objeto que tiene el reflejo le ponemos la textura especial, pero para dar una buena sensación de reflejo, hay que situar la textura en las coordenadas de textura correctas. Hay diversos algortimos para hacerlo, pero sería perder el tiempo, ya que OpenGL lo hace automático con el siguiente código:

```
glTexGenf(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glTexGenf(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
```

```
// Pintar objeto con textura...

glDisable(GL_TEXTURE_GEN_S);
glDisable(GL_TEXTURE_GEN_T);
```

Así que con ese código calcula la posición en la que habría que poner la textura, para que parezca que está reflejando el entorno que le rodea. No es una técnica perfecta que digamos, pero para dar el pego tampoco está tan mal. Así que en total habría que hacer:

+ Pintar el fondo con una textura.
+ Calcular las coordenadas (s, t) de la textura especial.
+ Pintar el objeto reflectante.
+ Desactivar el cálculo de las coordenadas (s, t).

En el libro hay un ejemplo en las páginas de la 299 a la 303.

3) La matriz de texturas:

En el capítulo cinco se habló de las diferentes matrices internas de OpenGL y se hizo referencia a GL_TEXTURE, que era la forma de invocar a la matriz de texturas. Con esta matriz podemos mover, rotar y redimensionar las texturas que estemos usando, del mismo modo que lo hacíamos con los polígonos. Para seleccionarla se usa:

```
glMatrixMode(GL_TEXTURE);
```

También se pueden utilizar a parte de glTranslate*(), glRotate*() y glScale*(), las funciones glPushMatrix() y glPopMatrix(). Con lo que podremos aplicar todo tipo de cambios que podamos considerar oportunos a las texturas.

4) Mapas de luces:

La iluminación realista siempre es un duro problema, por ello en juegos como los Quake I, II y III, ante la falta de recursos el ingenio. Y este consistía en que si no puedes calcular la iluminación en tiempo real, pues la precalculamos con un algoritmo horriblemente complicado, y obtenemos una textura especial. Esa textura sería una imagen con un formato de pixel de 8bits, que no representarían una paleta, sino la escala de grises (0 = negro, 255 = blanco). Así que cargamos la imagen en memoria y luego la cargamos en la tarjeta de video del siguiente modo:

```
// De este modo si queremos generar los mipmaps:
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_LUMINANCE, width, height,
                  GL_LUMINANCE, GL_UNSIGNED_BYTE, LightmapData);

// O de este si no vamos a usar mipmaps:
glTexImage2D(GL_TEXTURE_2D, 0, GL_LUMINANCE, width, height, 0,
             GL_LUMINANCE, GL_UNSIGNED_BYTE, LightmapData);
```

Con GL_LUMINANCE, le indicamos que el formato que vamos a usar para los pixels es de iluminación, lo que para nosotros sería una escala de grises, donde el negro es sin iluminar, y el blanco es iluminación máxima. El caso es para poder llevar a cabo esta técnica hay que utilizar multitexturizado, una vez tengamos cargadas en memoria la texturas del objeto y del mapa de luz, y hayamos comprobado que está soportada la extensión, tan solo tendremos que usarlas de esta forma:

```
glActiveTextureARB(GL_TEXTURE0_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, TexCuboID);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);

glActiveTextureARB(GL_TEXTURE1_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, TexLightmapID);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

// Pintar el objeto con el lightmap...

glActiveTextureARB(GL_TEXTURE1_ARB);
glDisable(GL_TEXTURE_2D);
glActiveTextureARB(GL_TEXTURE0_ARB);
glDisable(GL_TEXTURE_2D);
```

Y de esa forma tendremos un truco más para "iluminar" la escena que estamos renderizando, sin saturar la tarjeta. El único problema de esta técnica es que es un tanto estática, y para luces dinámicas se utilizan a día de hoy otros métodos más avanzados.

5) Multitexturizado multipase:

Como se ha podido ver en este capítulo el multitexturizado es una extensión bastante útil e importante. ¿Pero que podemos hacer cuando no está soportada? El sentido común nos diría que mandar a la mierda esa tarjeta, pero no estaría de más tener un algoritmo alternativo, aunque solo sea como curiosidad. Así que este algoritmo de simulación multitexturizado, lo que hace es usar las funciones de blending para pintar en varias pasadas las texturas y simular el multitexturizado. Es más lento que el original que va vía aceleración hardware, y el algoritmo sería más o menos el siguiente:

```
// Primer renderizado del objeto
glBindTexture(GL_TEXTURE_2D, tex1);
DrawTexturedCube(0.0f, 0.0f, 0.0f);

// Segundo renderizado del objeto
glEnable(GL_BLEND);
glDepthMask(GL_FALSE);
glDepthFunc(GL_EQUAL);
glBlendFunc(GL_ZERO, GL_SRC_COLOR);

    glBindTexture(GL_TEXTURE_2D, tex2);
    DrawTexturedCube(0.0f, 0.0f, 0.0f);

// Reestablecer los modos anteriores
glDepthMask(GL_TRUE);
glDepthFunc(GL_LESS);
glDisable(GL_BLEND);
```

El resto del código sería parecido al original con multitexturizado, a la hora de cargar las texturas en la tarjeta de video.

Capítulo 10: Las "display lists" y los arrays de vértices.

1) Las "display lists":

Como se habrá podido apreciar en los ejemplos del libro, hay una serie de ordenes que siempre se llaman para pintar los objetos, teniendo que mandar montón de datos a la tarjeta, lo cual en cierto modo resulta un poco aparatoso. Por ello, OpenGL tiene lo que él llama las display lists, que viene a ser listas con ordenes OpenGL para pintar objetos con una forma determinada en la escena. Pero con una gran ventaja, y es que la información se almacena en la memoria de la tarjeta de video, con lo que pintar estos listados de ordenes es mucho más rápido.

+ Creando una Display List:

Para crear una display list tendremos que usar la función:

```
GLuint glGenLists (GLsizei range);
```

Esta función devuelve el primer identificador de lista del número de listas que hemos pedido crear con range. Es decir, que range sirve para decir cuantas listas queremos crear, así que por ejemplo si nos devuelve glGenLists(10), el número 1000, las listas que se han creado van de la 1000 a la 1009, en total 10 listas. En caso de no poder reservar identificadores de lista, devolverá 0. Además tenemos esta otra función:

```
GLboolean glIsList (GLuint listname);
```

Que sirve para saber si listname es un identificador de lista ya usado (GL_TRUE) o no (GL_FALSE). Obviamente de estar siendo usado no deberíamos crear ninguna lista con ese identificador, pues se borraría la anterior.

+ Rellenandola de comandos:

Tras obtener un nombre identificador de tu display list, lo siguiente es crearla de verdad y darle "forma", para poder invocarla en un futuro al pintar la escena. Para este proposito tenemos las funciones:

```
void glNewList (GLuint listname, GLenum mode);
+ listname: Nombre de la lista.
+ mode: Modo de crear la nueva lista.
   - GL_COMPILE = Crea una lista y la compila.
   - GL_COMPILE_AND_EXECUTE = La compila y la ejecuta al crearla.

void glEndList (void);
```

La primera función crea la lista y deja a OpenGL preparado para recibir las ordenes que estarán recogidas por esa lista que hemos creado, y la segunda cierra el bloque de ordenes. En cuanto al modo de creación, lo normal es usar GL_COMPILE, que se limita a compilar la información y dejarla preparada para ser invocada en el futuro. OpenGL permite como comando almacenable en las listas la mayoría de sus funciones, pero hay algunas que no están admitidas, y que en vez de almacenarse en la lista son ejecutadas al instante. Estas funciones "no válidas" son:

```
glColorPointer      glDeleteLists          glDisableClientState
glEdgeFlagPointer   glEnableClientState    glFeedbackBuffer
glFinish            glFlush                glGenLists
glIndexPointer      glInterleavedArrays    glIsEnabled
```

```
glIsList            glNormalPointer        glPopClientAttrib
glPixelStore        glPushClientAttrib     glReadPixels
glRenderMode        glSelectBuffer         glTexCoordPointer
glVertexPointer
```

Otras funciones que tampoco funcionan para las listas son las variantes de glGet*(), o glTexImage() si estamos creando una textura proxy. Así que lo recomendable para una lista es crear geometría y llamar a otras sublistas para formar objetos complejos.

+ Ejecutando una Display List:

Ahora que ya sabemos crear las listas, es cuando necesitamos saber como invocarlas para que se ejecute su contenido. Para ello tenemos:

```
void glCallList (GLuint listname);
```

Pero si quisieramos llamar a varias listas podríamos usar:

```
void glCallLists (GLsizei num, GLenum type, const GLvoid * lists);
+ num: Número de listas a ejecutar.
+ type: GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT,
        GL_UNSIGNED_INT, GL_FLOAT, GL_2_BYTES, GL_3_BYTES, GL_4_BYTES.
+ lists: Array con los nombres de las listas.
```

Pero este sistema tiene una pega, ya que por defecto siempre recorre de la posición 0 a la num - 1. ¿Pero que ocurre si queremos ejecutar un intervalo de listas que está en medio del array? Pues que tenemos que cambiar el offset inicial con:

```
void glListName (GLuint offset);
```

De ese modo el intervalo a recorrer será desde offset a (offset + num - 1). Por cierto que si se cambia el valor, deberíamos tener el detalle de volverlo a poner a cero, para evitar posibles errores. En caso de cambiar el valor del offset inicial podemos recuperarlo con glGet*(), pasándole de parametro GL_LIST_BASE.

+ Advertencias sobre las Display Lists:

Ya que las funciones glCallList() o glCallLists() pueden ser llamadas al crear una lista, hay que andarse con mucho cuidado de no crear un bucle recursivo infinito de listas. También hay que tener cuidado con los estados que cambiemos dentro de una lista de comandos, ya que esto afecta a la máquina de OpenGL, y por consiguiente a lo que venga después. Por ello que sea recomendable usar glPushMatrix(), glPopMatrix(), glPushAttrib() y glPopAttrib(), para salvar el estado anterior al ejecutar la lista.

+ Destruyendo una Display List:

Cuando ya no necesitemos más una lista de comandos, podremos borrarla usando:

```
void glDeleteLists (GLuint listname, GLsizei range);
```

Donde listname será la lista o lista base que queremos borrar, y range el número de listas que deseamos borrar, con base el nombre que le hemos pasado en listname.

+ Las Display Lists y las texturas:

Las principales funciones de tratamiento de texturas como son glBindTexture(), glTextCoord() y glTextEnv(), pueden ser usadas como comandos válidos al crear una lista. De este modo ya definimos la geometría y las texturas que usa.

+ Ejemplo de Display Lists:

(Páginas de la 326 a la 328).

2) Arrays de vértices:

El otro tema importante que vamos a tocar son los arrays de vértices. Hasta ahora hemos visto como pintar vértices de una forma un tanto simple, y que sería un tanto pesada para cosas más avanzadas como pintar un modelo cargado de un fichero. Para esos casos existen los arrays de vértices, para cargar dichos objetos avanzados de un fichero a estos arrays, y luego enviárselos a OpenGL para que los procese y pinte la escena.

+ Activando arrays de vértices:

Para poder utilizar esta herramienta, hay que activar los diferentes tipos de arrays de vértices que querramos utilizar, con la función:

```
void glEnableClientState  (GLenum array);
void glDisableClientState (GLenum array);
+ array: Tipo de array que deseamos activar o desactivar.
   - GL_COLOR_ARRAY = Arrays que contienen el color de cada vértice.
   - GL_EDGE_FLAG_ARRAY = Arrays con los edge flags de cada vértice.
   - GL_INDEX_ARRAY = Arrays que contienen los índices de color de
      paleta para cada vértice.
   - GL_NORMAL_ARRAY = Arrays con las normales de cada vértice.
   - GL_TEXTURE_COORD_ARRAY = Arrays que contienen las coordenadas de
      textura para cada vértice.
   - GL_VERTEX_ARRAY = Array con las posiciones de cada vértice.
```

Como se puede observar hay un tipo de array específico para un tipo de dato en particular para los vértices, detalle que tendremos que tener bien en cuenta.

+ Trabajando con arrays:

Y ahora toca saber cuales son las funciones para indicar a OpenGL donde están los datos. La primera que inspeccionaremos será el array de colores de vértices:

```
void glColorPointer (GLint size, GLenum type, GLsizei stride,
                     const GLvoid * array);
+ size: Número de componentes de los pixels (3 para RGB o 4 para RGBA).
+ type: Tipo de dato del array que le estamos pasando (GL_BYTE,
   GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT,
   GL_UNSIGNED_INT, GL_FLOAT, GL_DOUBLE).
+ stride: Separación entre pixel y pixel en bytes (normalmente 0).
+ array: Puntero al array donde están los datos.
```

En cuanto al array con los edge flags y el array con los indices de colores:

```
void glIndexPointer (GLenum type, GLsizei stride, const GLvoid * array);
+ type: GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE.
```

```
void glEdgeFlagPointer (GLsizei stride, const GLboolean * array);
```

Pero los que posiblemente más usaremos son los arrays con normales, los que tienen coordenadas de texturas y los que tienen posiciones:

```
void glNormalPointer (GLenum type, GLsizei stride,
                      const GLvoid * array);
+ type: GL_BYTE, GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE.

void glTexCoordPointer (GLint size, GLenum type, GLsizei stride,
                        const GLvoid * array);
+ size: 1, 2, 3, 4.
+ type: GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE.

void glVertexPointer (GLint size, GLenum type, GLsizei stride,
                      const GLvoid * array);
+ size: 2, 3, 4.
+ type: GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE.
```

Posiblemente te hayas estado preguntando, ¿hey este sistema no parece hecho para un solo objeto? Bueno la verdad es que sí, si quieres pintar varios objetos puedes recurrir a dos formas de solucionar eso. La primera es juntar todos los datos en un array único, y la segunda es para cada objeto llamar a gl*Pointer(), y pasarle los punteros de los arrays de cada objeto.

- glDrawArrays():

  Una vez sabe OpenGL donde tiene que buscar los datos, la forma de mandarle a dibujar el objeto es con la función:

  ```
  void glDrawArrays (GLenum mode, GLint first, GLsizei count);
  + mode: Es el mismo parametro que el mode de glBegin().
  + first: Indica cual es la posición inicial dentro del array.
  + count: Indica el número de elementos que queremos mandar.
  ```

  De este modo podremos dentro de un array, sin pasarnos de sus límites, dibujar un conjunto reducido si asi lo deseamos de elementos, tal que empieza en first y termina en (first + count - 1).

- glDrawElements():

  Pero en el caso de que no querramos pintar con conjunto secuencial de vértices, podemos usar esta otra función que es más avanzada:

  ```
  void glDrawElements (GLenum mode, GLsizei count, GLenum type,
                       const GLvoid * indices);
  + mode: Es el mismo parametro que el mode de glBegin().
  + count: Número de elementos en el array indices.
  + type: GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT, GL_UNSIGNED_INT.
  + indices: Array con los indices de los elementos a pintar.
  ```

  De este modo no solo podemos indicar un orden aleatorio, para pintar vértices contenidos en el array de vértices, sino que podemos repetir varias veces un mismo elemento que pudiera estar compartido por varios polígonos.

- glDrawRangeElements():

Una función que fue añadida en la versión 1.2 (por lo que posiblemente haya que usar wglGetProcAddress() para obtenerla), fue esta versión más avanzada de la función que acabamos de ver glDrawElements():

```
void glDrawRangeElements (GLenum mode, GLuint start, GLuint end,
                          GLsizei count, GLenum type,
                          const GLvoid * indices);
```

¿Qué hace esta función? Pues lo mismo que glDrawElements salvo con una excepción, y es que le indicaremos con start y end, cual es el índice mínimo y cual el índice máximo en el array de vértices. De este modo solo se procesarán los índices que estén contenidos en el rango de elementos que acabamos de especificar.

- glArrayElement():

Y la última función que hay relacionada con el procesado de arrays de vértices es:

```
void glArrayElement (GLint index);
```

Que manda a procesar un solo vértice, y no es el tipo de función que haya que usar a menudo, ya que es bastante poco eficiente, en comparación con las otras.

+ Arrays de vértices y el multitexturizado:

Es ahora que ya sabemos todo lo que necesitamos de arrays de vértices, cuando vamos a ver aquella función del multitexturizado que no vimos:

```
void glClientActiveTextureARB (GLenum textUnit);
+ textUnit = GL_TEXTURE#_ARB (# = 0 .. maxTexUnits-1)
```

El funcionamiento de esta función es relativamente sencillo, al estar el multitexturizado soportado, internamente OpenGL tendrá soporte para varios arrays de coordenadas de texturas internamente. Así que con esta función le indicamos cual está activo:

```
glClientActiveTextureARB(GL_TEXTURE0_ARB);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
glTexCoordPointer(2, GL_FLOAT, 0, (GLvoid *) texUnit0Vertices);

glClientActiveTextureARB(GL_TEXTURE1_ARB);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
glTexCoordPointer(2, GL_FLOAT, 0, (GLvoid *) texUnit1Vertices);
```

Para desactivar los arrays de coordenadas de texturas, hay que hacerlo al revés de como se a activado, para evitar problemillas en la máquina de estados OpenGL.

+ Bloqueando un array:

Existe una extensión llamada GL_EXT_compiled_vertex_array, que está compuesta de las dos siguientes funciones.

```
void glLockArraysEXT   (GLint first, GLsizei count);
void glUnlockArraysEXT (void);
```

La primera función bloquea una serie de arrays que van de first a (first + cont - 1). La segunda desbloquea los arrays previamente bloqueados. Esto sirve para que internamente OpenGL haga operaciones más rápidas, a cambio de no poder modificar el

contenido de los arrays, ya que el resultado de la primera vez que procesa el array lo pasa a la caché de OpenGL, y las siguientes veces pilla el resultado de la caché, al suponerse que el contenido de los arrays no ha sido modificado.

+ Ejemplo de arrays de vértices:

(Páginas de la 335 a la 338).

Capítulo 11: Mostrando texto.

1) Fuentes bitmap:

Y en este capítulo aprenderemos a mostrar texto en la pantalla de una forma relativamente sencilla, para ello usaremos wglUseFontBitmaps(), para generar los bitmaps que serán necesarios a la hora de renderizar texto. Pero primero hay que hacer un par de cosillas que resumiremos en una función tal que:

```
unsigned int CreateBitmapFont (char * fontname, int fontsize)
{
    HFONT hFont;
    unsigned int base;

    if(stricmp(fontname, "symbol") == 0)
    {
        hFont = CreateFont (fontsize, 0, 0, 0, FW_BOLD, FALSE, FALSE,
                            FALSE, SYMBOL_CHARSET, OUT_TT_PRECIS,
                            CLIP_DEFAULT_PRECIS, ANTIALIASED_QUALITY,
                            FF_DONTCARE | DEFAULT_PITCH, fontname);
    }
    else
    {
        hFont = CreateFont (fontsize, 0, 0, 0, FW_BOLD, FALSE, FALSE,
                            FALSE, ANSI_CHARSET, OUT_TT_PRECIS,
                            CLIP_DEFAULT_PRECIS, ANTIALIASED_QUALITY,
                            FF_DONTCARE | DEFAULT_PITCH, fontname);
    }

    if(!hFont) return 0;

    base = glGenLists(96);
    SelectObject(g_HDC, hFont);
    wglUseFontBitmaps(g_HDC, 32, 96, base);

    return base;
}
```

Con una función como esa se crearían una serie de listas con las letras de la 32 a la (96 + 32 - 1), almacenadas en unos bitmaps OpenGL. Pero analicemos las funciones a las que hemos llamado, primero CreateFont():

```
HFONT CreateFont (int nHeight, int nWidth, int nEscapement,
                  int nOrientation, int fnWeight, DWORD fdwItalic,
                  DWORD fdwUnderline, DWORD fdwStrikeOut,
                  DWORD fdwCharSet, DWORD fdwOutputPrecision,
                  DWORD fdwClipPrecision, DWORD fdwQuality,
```

```
                    DWORD fdwPitchAndFamily, LPCTSTR lpszFace);
+ nHeight: Altura lógica de la fuente.
+ nWidth: Ancho lógico medio ponderado.
+ nEscapement: Ángulo de escape.
+ nOrientation: Ángulo de orientación de la linea base.
+ fnWeight: Ancho de la fuente (para la negrita).
+ fdwItalic: Flag para poner la fuente en itálica.
+ fdwUnderline: Flag para subrayar la fuente.
+ fdwStrikeOut: Flag para tachar la fuente.
+ fdwCharSet: Tipo de conjunto de caracteres.
+ fdwOutputPrecision: Precisión de salida.
+ fdwClipPrecision: Precisión de recorte.
+ fdwQuality: Calidad de salida.
+ fdwPitchAndFamily: Familia y modificación.
+ lpszFace: Nombre de la fuente.
```

Una vez obtenida la fuente, que en caso de devolver 0 es que no ha conseguido obtener ninguna, tendremos que llamar a glGenLists() para crear las listas donde almacenar las imagenes. Luego seleccionamos la fuente para la que la api del windows la pueda procesar en la siguiente función:

```
BOOL wglUseFontBitmaps (HDC hdc, DWORD first, DWORD count,
                        DWORD listBase);
+ hdc: Dispositivo de contexto donde será la fuente usada.
+ first: Posición base para crear la lista de letras.
+ count: Número de letras que vamos a meter en la lista.
+ listBase: Nombre base de todas las listas.
```

Así que tras lograr crear la lista de bitmaps, que contienen las letras que vamos a usar, de la fuente seleccionada, tendremos que saber usarla de algún modo:

```
void PrintString (unsigned int base, char * str)
{
    if((base == 0) || (str == NULL)) return;

    glPushAttrib(GL_LIST_BIT);
        glListBase(base - 32);
        glCallLists(strlen(str), GL_UNSIGNED_BYTE, str);
    glPopAttrib();
}

void ClearFont (unsigned int base)
{
    if(base != 0)
        glDeleteLists(base, 96);
}
```

Con la primera función encontramos un ejemplo sencillo para pintar una cadena de texto en pantalla, y la segunda es para liberar de la memoria de la tarjeta el espacio ocupado por la fuente. Además la forma de usar esto más o menos sería:

```
void Initialize (void)
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);

    glShadeModel(GL_SMOOTH);
    glEnable(GL_DEPTH_TEST);

    listBase = CreateBitmapFont("Comic Sans MS", 48);
}
```

```
void Render (void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    glTranslatef(0.0f, 0.0f, -1.0f);
    glColor3f(1.0f, 1.0f, 1.0f);

    glRasterPos2f(-0.35f, 0.0f);
    PrintString(listBase, "OpenGL Bitmap Fonts!");

    glFlush();
    SwapBuffers(g_HDC);
}
```

Y con esto ya tenemos para pintar cualquier tipo de texto en pantalla, con algunas limitaciones como que no podremos ponerle ninguna textura al texto.

2) Fuentes tridimensionales:

Para crear una fuente con tres dimensiones hay que seguir más o menos los mismos pasos. Uno de los principales cambios es que la lista a crear será de 256 elementos, y además habrá que crear un array de 256 elementos del siguiente tipo:

```
typedef struct _GLYPHMETRICSFLOAT {
    FLOAT      gmfBlackBoxX;
    FLOAT      gmfBlackBoxY;
    POINTFLOAT gmfptGlyphOrigin;
    FLOAT      gmfCellIncX;
    FLOAT      gmfCellIncY;
} GLYPHMETRICSFLOAT;
```

Array que será rellenado con la función wglUseFontOutlines(), que será la que usaremos en vez de wglUseFontBitmaps():

```
BOOL wglUseFontOutlines (HDC hdc, DWORD first, DWORD count,
                         DWORD listBase, FLOAT deviation,
                         FLOAT extrusion, int format,
                         LPGLYPHMETRICSFLOAT lpgmf);
+ hdc: Dispositivo de contexto donde será la fuente usada.
+ first: Posición base para crear la lista de letras.
+ count: Número de letras que vamos a meter en la lista.
+ listBase: Nombre base de todas las listas.
+ deviation: Indica la desviación máxima de la fuente.
+ extrusion: Profundidad hacia el eje -z de la fuente.
+ format: Indica si las listas serán lineas o polígonos.
+ lpgmf: Información sobre las medidas de la fuente.
```

Así que la función sería un pelín diferente para crear una fuente nueva, solo habría que tener en cuenta que la lista ahora será de 256 elementos y que usaremos una función distinta para crear las listas de caracteres:

```
unsigned int CreateBitmapFont (char * fontname, int fontsize, float depth)
{
    HFONT hFont;
    unsigned int base;
```

```
    if(stricmp(fontname, "symbol") == 0)
    {
        hFont = CreateFont (fontsize, 0, 0, 0, FW_BOLD, FALSE, FALSE,
                            FALSE, SYMBOL_CHARSET, OUT_TT_PRECIS,
                            CLIP_DEFAULT_PRECIS, ANTIALIASED_QUALITY,
                            FF_DONTCARE | DEFAULT_PITCH, fontname);
    }
    else
    {
        hFont = CreateFont (fontsize, 0, 0, 0, FW_BOLD, FALSE, FALSE,
                            FALSE, ANSI_CHARSET, OUT_TT_PRECIS,
                            CLIP_DEFAULT_PRECIS, ANTIALIASED_QUALITY,
                            FF_DONTCARE | DEFAULT_PITCH, fontname);
    }

    if(!hFont) return 0;

    base = glGenLists(256);
    SelectObject(g_HDC, hFont);
    wglUseFontOutlines(g_HDC, 0, 255, base, 0.0f, depth,
                       WGL_FONT_POLYGONS, gmf);

    return base;
}
```

Las ventajas de tener el array de medidas es que podemos pintar el texto centrado si queremos, u otra clase de efectos basados en las medidas:

```
void PrintStringCenter (unsigned int base, char * str)
{
    if((base == 0) || (str == NULL)) return;

    float length = 0;
    int idx;

    for(idx = 0; idx < strlen(str); idx++)
    {
        length += gmf[str[idx]].gmfCellIncX;
    }
    glTranslatef(-length/2.0f, 0.0f, 0.0f);

    glPushAttrib(GL_LIST_BIT);
        glListBase(base);
        glCallLists(strlen(str), GL_UNSIGNED_BYTE, str);
    glPopAttrib();
}
```

El resto de las funciones, la de borrar la fuente sería igual salvo que habría que borrar unas 256 listas. Y al renderizar ya no tendríamos que usar glRasterPos2f(), pues lo que vamos a poner son polígonos, y no bitmaps en pantalla.

3) Fuentes con texturas mapeadas:

Para poner una textura a la fuente que querramos pintar, hemos de cargar la imagen, pedir un nombre para esta, crear la textura, configurarla, cargarla con o sin mipmap, y configurar la generación automática de coordenadas para la textura que usaremos en el texto, con el siguiente código:

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
```

¿Te suena, eh? Sí es parecido a lo que hacíamos con environment mapping, solo que para aquella situación usabamos GL_SPHERE_MAP, que le venía a decir que la textura era para realizar un environment map. Con GL_OBJECT_LINEAR le decimos que la textura está fija a un objeto. Y existe otro valor llamado GL_EYE_LINEAR, que indica que la textura está fija en una posición de la pantalla, con lo que si el objeto se mueve, la textura es movida para dar la impresión de que no se mueve con respecto al fondo.

Así que con ese código, la textura tendrá una posición fija a la hora de ser mapeada en el texto que queremos renderizar. El resto del programa es igual que el de la fuente de tres dimensiones. La única reseña que hay que hacer, es que en caso de usar más objetos en la escena, no hay que olvidar que antes de pintar la cadena, hay que configurar algunos parametros, como invocar la textura, o el código ahí expuesto para poner bien las coordenadas. Y cuando hayamos terminado de poner el texto, hay que desconfigurar las cosas que no vayamos a usar para el siguiente objeto.

Capítulo 12: Buffers de OpenGL.

1) ¿Qué es un buffer de OpenGL?:

Un buffer es principalmente una zona de memoria que almacena una determinada información de los pixels que vemos en la pantalla. Por ejemplo en el buffer de color se almacenará un valor RGBA. Existen cuatro tipos de buffers en OpenGL, de color, de profundidad, de stencil y de acumulación. Todos ellos juntos forman lo que se denomina el frame buffer, que es sobre lo que operamos al tocar cualquiera de ellos.

+ Configurando el formato de pixel:

En el primer tema vimos como había que hacer para crear una ventana, sobre al que pintar el frame buffer de OpenGL. Uno de los pasos era configurar el formato del pixel con la estrutura PIXELFORMATDESCRIPTOR. Bien esa estructura tenía algunos campos que se comentó para que servían, aunque muy por encima. Uno de ellos dwFlags, vimos que lo configurabamos para que pintara sobre la ventana, con OpenGL y un doble buffer. Pero hay unas cuantas opciones más:

+ PFD_DRAW_TO_WINDOW = El buffer puede dibujar sobre una ventana.
+ PFD_DRAW_TO_BITMAP = El buffer puede dibujar sobre un bitmap en memoria.
+ PFD_SUPPORT_GDI = El buffer soporta dibujado con GDI.
+ PFD_SUPPORT_OPENGL = El buffer soporta dibujado con OpenGL.
+ PFD_DOUBLEBUFFER = El buffer tiene doble buffer.
+ PFD_STEREO = El buffer es estereoscópico.
+ PFD_DEPTH_DONTCARE = Es formato puede tener o no buffer de profundidad.
+ PFD_DOUBLEBUFFER_DONTCARE = Es formato puede tener o no doble buffer.
+ PFD_STEREO_DONTCARE = Es formato puede tener o no buffer estereoscópico.

Luego tenemos cColorBits, cDepthBits, cAccumBits y CStencilBits, para indicar el tamaño de los cuatro buffers soportados en OpenGL. Si se le pasa 0 como valor, el buffer será desactivado.

Luego hay un par de campos que tienen varias opciones, pero que no se suelen usar como es el caso de iPixelType, que normalmente valdrá PFD_TYPE_RGBA, aunque también tiene como valor posible PFD_TYPE_COLORINDEX. El otro campo es iLayerType, que normalmente valdrá PFD_MAIN_PLANE, aunque también puede valer PFD_OVERLAY_PLANE o PFD_UNDERLAY_PLANE.

+ Borrando el contenido de los buffers:

Pero antes de entrar en detalle con cada tipo de buffer, vamos a aprender como se borran. Lo primero que hay que hacer es seleccionar el valor con el que vamos a borrar cada tipo de buffer:

```
void glClearColor   (GLclampf red, GLclampf green, GLclampf blue,
                     GLclampf alpha);
void glClearDepth   (GLclampf depth);
void glClearStencil (GLint s);
void glClearAccum   (GLfloat red, GLfloat green, GLfloat blue,
                     GLfloat alpha);
```

El tipo GLclampf se supone que tiene que contener un valor entre 0.0 y 1.0. Por defecto el valor que tienen cada parametro es de 0, salvo depth que vale 1. Configurado los valores de borrado para poder borrar los buffers usaremos la función:

```
void glClear (GLbitfield mask);
+ mask:
   - GL_COLOR_BUFFER_BIT   = Borra el buffer de color.
   - GL_DEPTH_BUFFER_BIT   = Borra el buffer de profundidad.
   - GL_STENCIL_BUFFER_BIT = Borra el buffer de stencil.
   - GL_ACCUM_BUFFER_BIT   = Borra el buffer de acumulación.
```

Para poder borrar varios en una sola llamada solo tenemos que encadenarlos con el operador or a nivel de bit, como se ha visto en algún que otro ejemplo.

2) El buffer de color:

Este buffer almacena la información referente al color de la escena actual. Podemos indicar que utilice un doble buffer, y también que pueda ser un buffer estereoscópico.

+ El sistema de doble-buffer:

Al usar la técnica del double buffer, OpenGL tiene un buffer trasero y uno delantero, que intercambia a la hora de pintar en pantalla el resultado final. Por defecto el buffer en el que se renderiza la escena, antes de actualizar la pantalla, es en el buffer trasero. Pero hay una función para modificar ese detalle:

```
void glDrawBuffer (GLenum mode);
+ mode: GL_FRONT, GL_BACK, GL_FRONT_AND_BACK, GL_NONE.
```

Si le pasamos GL_FRONT, actuará como si solo tuviera un solo buffer. Las ventajas de esta función no parecen ser muchas, pero pasándole GL_NONE, ocurre que aquello que mandemos a pintar no será reflejado en el buffer de color, cosa que sería bastante inútil de no ser por el hecho de que aunque no haya cambios en el buffer de color, si tenemos

activados los otros buffers, sí serán afectados, llegando a poder ser modificados, si se dan las condiciones para ello.

Otra función que también es útil, es la que le dice a OpenGL de qué buffer tiene que leer los pixels que se pueden manipular con las funciones glReadPixels(), glCopyPixels(), glCopyTexImage*() y glCopyTexSubImage*():

```
void glReadBuffer (GLenum mode);
+ mode: GL_FRONT, GL_BACK, GL_FRONT_AND_BACK, GL_NONE.
```

+ Buffer estereoscópico:

El buffer estereoscópico sirve para tener dos buffers de renderizado, uno derecho y otro izquierdo, uno para cada ojo. No es algo muy usado en PC, si no se tiene unas gafas que tenga dos pantallas e inventos así. De este modo al activar este sistema el parametro mode de las dos funciones del punto anterior también podrían valer: GL_LEFT, GL_RIGHT, GL_FRONT_LEFT, GL_FRONT_RIGHT, GL_BACK_LEFT, GL_BACK_RIGHT.

3) El buffer de profundidad:

La misión de este buffer es de llevar un control de la profundidad a la que está cada pixel en la pantalla, normalmente se busca que esta distancia sea la del pixel que más cerca está a la pantalla. Así podemos saber al pintar polígonos si salen o no en pantalla, por lo que si hemos pintado varios objetos que tapan grandes areas de la pantalla, todo aquello que pintemos detrás y que esté oculto por los objetos más cercanos a la cámara, no serán pintados.

+ Las funciones de comparación de la profundidad:

Así que a la hora de pintar las cosas, la coordenada z del objeto es comparada con los valores que hay en el buffer de profundidad, y en base a este valor y el resultado de la comparación, serán pintados algunos pixeles, todos o ninguno, actualizando el valor de los pixeles que hayan sido pintados en el buffer de profundidad. Para decidir que función de comparación vamos a usar tenemos:

```
void glDepthFunc (GLenum func);
+ func: Pintar el objeto si la z de este es...
   - GL_LESS = Menor que el valor del buffer de profundidad.
   - GL_EQUAL = Igual que el valor del buffer de profundidad.
   - GL_LEQUAL = Menor o igual que el valor del buffer de profundidad.
   - GL_GREATER = Mayor que el valor del buffer de profundidad.
   - GL_NOTEQUAL = Distinto que el valor del buffer de profundidad.
   - GL_GEQUAL = Mayor o igual que el valor del buffer de profundidad.
   - GL_NEVER = Nunca pintarlo.
   - GL_ALWAYS = Pintarlo siempre.
```

Por defecto la función del buffer de profundidad será GL_LESS, ya que normalmente se suele querer que solo se pinten los píxeles que no están tapados por otros objetos.

+ Utilizando el buffer de profundidad:

En las páginas de la 370 a la 382 hay dos ejemplos. El primero muestra la diferencia entre la función GL_LESS y GL_ALWAYS, que radica en que con la segunda función las

cosas se irán pintando siempre encima de las que hubiera antes, con lo que tendríamos que encargarnos nosotros de pintar en orden las cosas para evitar fallos visuales.

Y el segundo ejemplo muestra como usar de una forma útil glDrawBuffer(), de tal forma que aunque le mandemos que no dibuje en ningún buffer los polígonos (GL_NONE), el contenido del buffer de profundidad será modificado aunque lo que hayamos "pintado" no sea visible porque el buffer de color no haya sido actualizado.

Un último detalle sobre el buffer de profundidad, es que el número de bits que se le suele normalmente dar es de 16bits o 32bits. A mayor número de bits mayor se supone que será la precisión que podrá alcanzar el buffer para realizar las comparaciones.

4) El stencil buffer:

El buffer de stencil es una de esas cosas "raras" que requiere bastante esfuerzo entender. De primeras, ¿qué es lo que se almacena en el buffer este? Pues sabemos que son enteros (que normalmente configuraremos de 16bits en la propiedad cStencilBits del descriptor del formato de pixel), cuya información no representa nada en particular, sino lo que querramos nosotros representar. Muchas veces este buffer es usado para delimitar las zonas donde vamos a pintar.

Una vez creado el contexto de renderizado de OpenGL, con el formato de pixel adecuado, cada vez que querramos usar el buffer de stencil tendremos que activarlo con glEnable() y el parametro GL_STENCIL_TEST. De este modo se realizarán las operaciones que querramos indicar sobre este buffer. Para determinar ese punto se usa:

```
void glStencilFunc (GLenum func, GLint ref, GLuint mask);
+ func: Función de comparación.
   - GL_NEVER = Siempre devuelve false la función, que siempre falla.
   - GL_ALWAYS = Siempre da true la función, que siempre pasa.
   - GL_LESS = Pasa si ref es menor que el valor del buffer.
   - GL_LEQUAL = Pasa si ref es menor o igual que el valor del buffer.
   - GL_GREATER = Pasa si ref es mayor que el valor del buffer.
   - GL_GEQUAL = Pasa si ref es mayor o igual que el valor del buffer.
   - GL_EQUAL = Pasa si ref es igual que el valor del buffer.
   - GL_NOTEQUAL = Pasa si ref es distinto que el valor del buffer.
+ ref: Es el valor de referencia con el que comparar el valor del buffer.
+ mask: Es una máscara que se aplica con un and a nivel de bit tanto al
   valor de referencia, como al valor que hay en el buffer de stencil.

En resumen:
+ GL_NEVER    -> Siempre falla.
+ GL_ALWAYS   -> Siempre pasa.
+ GL_LESS     -> Si (ref & mask) <  (stencil & mask) entonces pasa.
+ GL_LEQUAL   -> Si (ref & mask) <= (stencil & mask) entonces pasa.
+ GL_GREATER  -> Si (ref & mask) >  (stencil & mask) entonces pasa.
+ GL_GEQUAL   -> Si (ref & mask) >= (stencil & mask) entonces pasa.
+ GL_EQUAL    -> Si (ref & mask) == (stencil & mask) entonces pasa.
+ GL_NOTEQUAL -> Si (ref & mask) != (stencil & mask) entonces pasa.
```

Pero además de determinar la función, hemos de indicar que tiene que hacer la función en caso de fallar o pasar, con la siguiente función:

```
void glStencilOp (GLenum fail, GLenum zfail, GLenum zpass);
+ fail: Qué hacer cuando la función de evaluación devuelve fallo.
+ zfail: La función devuelve exito, pero la de profundidad falla.
```

```
+ zpass: La función devuelve exito, y la de profundidad también.
Valores para fail, zfail y zpass:
- GL_KEEP    = Conserva el valor actual del buffer.
- GL_ZERO    = Asigna al valor actual del buffer un cero.
- GL_REPLACE = Asigna al valor actual del buffer el valor de ref.
- GL_INCR    = Incrementa el valor actual del buffer.
- GL_DECR    = Decrementa el valor actual del buffer.
- GL_INVERT  = Invierte bit a bit el valor actual del buffer.
```

En caso de querer cambiar solo la máscara que se usa en la evaluación del buffer de stencil, existe una función específica para ello:

```
void glStencilMask (GLuint mask);
```

En el libro hay un ejemplo en las páginas de la 385 a la 392, sobre como usar el buffer de stencil para crear efectos como el del reflejo de un suelo bien encerado. Lo más destacable de dicho algoritmo es el siguiente fragmento esquemático de código:

```
// Configuramos algunos datos de la cámara, y el ángulo del donut.

// Borramos los buffers de color, profundidad y stencil.
// Borramos la matriz de modelview.

// Configuramos la posición y dirección de la camara.

glDisable(GL_DEPTH_TEST);

// Desactivamos la modificación de los colores en el buffer de color,
// y activamos el manejo del buffer de stencil.
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
glEnable(GL_STENCIL_TEST);

// Configuramos la función de evaluación de tal modo, que cualquier pixel
// que vayamos a modificar, se le asigne el valor 1.
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
glStencilFunc(GL_ALWAYS, 1, 1);

PintarSuelo();

glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);

// Cambiamos la función de evaluación de tal modo, que no se cambie el
// contenido del buffer de stencil, y que no se pinte en pantalla, más
// que en la zona que tenga los pixels valor de 1.
glEnable(GL_STENCIL_TEST);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
glStencilFunc(GL_EQUAL, 1, 1);

// Pintamos el reflejo del donut.
glPushMatrix();
    glScalef(1.0f, -1.0f, 1.0f);
    glCullFace(GL_FRONT);
    PintarDonut();
    glCullFace(GL_BACK);
glPopMatrix();

glDisable(GL_STENCIL_TEST);

// Pintamos el suelo con transparencia.
glEnable(GL_BLEND);
```

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glColor4f(1.0f, 1.0f, 1.0f, 0.4f);
PintarSuelo();
glDisable(GL_BLEND);

glEnable(GL_DEPTH_TEST);
PintarDonut();

glFlush();
SwapBuffers(g_HDC);
```

Y eso es básicamente todo lo que hay en la función de renderizado, teniendo en cuenta que para este caso básicamente hemos definido un area, que estaría ocupada por el suelo, que no se pinta gracias a la función glColorMask(r, g, b, a). Luego pintamos el reflejo, en esa zona marcada, desactivamos el uso del buffer de stencil, y pintamos el suelo con transparencia, y luego el objeto reflejado.

5) El buffer de acumulado:

Este buffer es parecido al buffer de color, solo que se usa para realizar algunos efectos la mar de curiosos como son el motion blur, efectos de profundidad de visión, antialiasing de la escena, o sombras suavizadas entre otros. ¿Como funciona el buffer de acumulación? Básicamente la idea es pintar varias veces el buffer de color, y cada vez que lo pintemos cogeremos el contenido y lo meteremos en el buffer de acumulación. Terminada la escena cogeremos el contenido final del buffer de acumulación, y lo enviamos a la pantalla.

Para manejar el buffer tendremos básicamente una sola función, que hará todo lo que necesitemos hacer con este buffer:

```
void glAccum (GLenum op, GLfloat value);
+ op: Operación que realizaremos sobre el buffer.
   - GL_ACCUM = Obtiene los valores RGBA del buffer actual de color
      seleccionado para leerlos y acumularlos en el buffer.
   - GL_LOAD = Parecido a GL_ACCUM, solo que en este caso borra todos los
      valores que había almacenados antes en el buffer.
   - GL_ADD = Suma cada pixel en el buffer de acumulación a value.
   - GL_MULT = Lo mismo que la anterior pero multiplicando.
   - GL_RETURN = Multiplica cada pixel del buffer de acumulación por value
      y manda el resultado de dicha operación al buffer de color.
+ value: Valor que será usado para realizar la operación.
```

De este modo para crear un motion blur sencillo habría que hacer lo siguiente:

```
// Primero pintamos el objeto entero y lo añadimos.
DrawObject();
glAccum(GL_LOAD, 0.7f);

// Y pintamos 5 veces más el objeto, añadiéndolo con menos intensidad.
for(int i = 1; i < 5; ++i)
{
    glRotatef(objangle - (float) i, 0.0f, 1.0f, 0.0f);
    DrawObject();
    glAccum(GL_ACCUM, 0.2f);
}

// Mandamos el resultado final al buffer de color.
glAccum(GL_RETURN, 1.0);
```

Un ejemplo bastante sencillo, aunque en un videojuego de verdad, renderizar el mundo varias veces, podría ser mortal, así que es posible que cada vez que se renderiza la pantalla, se guarde una "foto" de dicho render, y usando las anteriores se realice la acumulación para realizar el efecto final.

Capítulo 13: Cuádricas de OpenGL.

1) Introducción a las cuádricas en OpenGL:

Las cuádricas son objetos que contienen información sobre formas esféricas, que podemos pintar con OpenGL. Para crear una hay que usar:

```
GLUquadricObj * gluNewQuadric (void);
```

Crea el objeto, que en caso de no poder devuelve un NULL. Una vez creado el objeto tendrá una serie de estados que tenemos que tener en cuenta:

+ Estilo de pintado.
+ Las normales.
+ La orientación.
+ Y las coordenadas de las textura.

Aunque es posible pintar diversas formas con el mismo objeto, y compartir los estados entre varias figuras, no siempre es recomendable. Ya que los cálculos relacionados con estos temas no son precisamente gratuitos.

+ Estilo de pintado:

Este estado es para indicar si los polígonos son pintados con relleno, como lineas, o como puntos, y para ello se usa la función:

```
void gluQuadricDrawStyle (GLUquadricObj * qobj, GLenum style);
```

El parametro style puede valer GLU_FILL (que es el valor por defecto), GLU_LINE, GLU_POINT, o GLU_SILHOUETTE.

+ Las normales:

Este estado indica el modo del que son las normales generadas:

```
void gluQuadricNormals (GLUquadricObj * qobj, GLenum normalMode);
```

El parametro normalMode puede valer GLU_NONE, GLU_FLAT, GLU_SMOOTH (que es el valor por defecto para este estado).

+ La orientación:

Este estado indica la orientación de las normales del objeto:

```
void gluQuadricOrientation (GLUquadricObj * qobj, GLenum orientation);
```

El parametro orientation puede valer GLU_OUTSIDE (que es el valor por defecto) o puede valer GLU_INSIDE.

+ Las coordenadas de las textura:

Este estado indica si se van a usar coordenadas para la textura del objeto:

```
void gluQuadricTexture (GLUquadricObj * qobj, GLboolean useTexCoords);
```

El parametro useTexCoords puede valer GL_TRUE o GL_FALSE.

+ Eliminando una cuádrica:

Una vez hayamos terminado de usar el objeto, podemos borrarlo con:

```
void gluDeleteQuadric (GLUquadricObj * qobj);
```

De este modo la memoria que ocupaba será liberada.

2) Discos:

Una vez tenemos la base, podemos empezar a crear formas, y la primera de ellas será crear un disco plano, que esté en el punto cero del plano z:

```
void gluDisk (GLUquadricObj * qobj, GLdouble innerRadius,
              GLdouble outerRadius, GLint slices, GLint loops);
```

En esta función a parte de pasarle el objeto, tenemos que indicarle el radio interno y el externo, por si queremos crear un disco con el centro hueco. Los dos últimos parametros sirven para indicar la cantidad de polígonos que forman el objeto, slices para indicar el número de divisiones (cual si fuera una tarta), tiene el disco, y loops para indicar el número de "anillos" que tiene el disco. Además existe otra función, por si queremos pintar solo una parte del disco:

```
void gluPartialDisk (GLUquadricObj * qobj, GLdouble innerRadius,
                     GLdouble outerRadius, GLint slices, GLint loops
                     GLdouble startAngle, GLdouble sweepAngle);
```

La función es igual, solo que se le añaden dos parametros, el ángulo de inicio, y el ángulo del disco en sí. Por lo que empezaría en startAngle y terminaría en (startAngle + sweepAngle).

3) Cilindros:

La siguiente forma que se puede crear son los cilindros:

```
void gluCylinder (GLUquadricObj * qobj, GLdouble baseRadius,
                  GLdouble topRadius, GLdouble height,
                  GLint slices, GLint stacks);
```

Para definir la forma hemos de indicarle el radio de la base, el radio del techo, y la altura. En cuanto a la cantidad de polígonos, slices y stacks indican la cantidad de divisiones que tienen en lo ancho y lo alto.

4) Esferas:

Y la última es crear esferas, que es parecida a los cilindros:

```
void gluSphere (GLUquadricObj * qobj, GLdouble radius,
                GLint slices, GLint stacks);
```

Para ello tendremos que indicarle el radio, y la cantidad de polígonos, del mismo modo que lo hacíamos con el cilindro.

5) Un ejemplo con cuádricas:

(Páginas de la 404 a la 407).

Capítulo 14: Curvas y superficies.

1) Representación de curvas y superficies:

Hasta ahora la mayoría de las cosas representadas, eran poligonales, pero OpenGL también tiene la capacidad de representar superficies curvas. Normalmente para definir una curva se utilizan ecuaciones, y más o menos en esa dirección vamos a seguir.

+ Ecuaciones paramétricas:

Son ecuaciones que no dependen simplemente de x, sino que tanto para x, como para y, existen variables de las que dependen. En el caso de las curvas, los parametros serán s y t, para las ecuaciones que se manejan en OpenGL.

+ Puntos de control y continuidad:

La cuestión es que para definir curvas, no vamos a meter una ecuación, sino que iremos metiendo puntos, con los que seguramente se realizará una interpolación, que nos dará el resto de los puntos de la curva. Cuantos más puntos definamos, normalmente más "perfecta" será la curva.

Pero un detalle importante es definir cual es la forma de relacionarse entre los puntos de control, que forman la curva. Hay diversas formas que son:

+ De ninguna forma, se forman curvas, pero sin ser una curva continua.
+ De forma posicional, cuando algunas curvas acaban compartiendo, que el punto final de una es el inicial de la otra.
+ De forma tangencial, de tal modo que se traza una tangente, que encaje en la continuidad de las diversas curvas.
+ De forma curvacional, que es parecido a la tangencial pero más suave.

2) Evaluadores:

Así que visto un poco por encima lo anterior, es hora de aprender a indicar puntos de control, para trazar curvas y superficies Bézier en OpenGL. Para ello hay que activar su uso y configurarlo, con esta función:

```
void glMap1f (GLenum target, GLfloat u1, GLfloat u2, GLint stride,
              GLint order, const float * points);
+ target: Indica que es lo que representan los puntos de control.
   - GL_MAP1_VERTEX_3        = Coordenadas de puntos (x, y, z).
   - GL_MAP1_VERTEX_4        = Coordenadas de puntos (x, y, z, w).
   - GL_MAP1_INDEX           = Colores en modo paleta.
   - GL_MAP1_COLOR_4         = Colores RGBA.
   - GL_MAP1_NORMAL          = Coordenadas de las normales.
   - GL_MAP1_TEXTURE_COORD_1 = Coordenadas de la textura (s).
   - GL_MAP1_TEXTURE_COORD_2 = Coordenadas de la textura (s, t).
   - GL_MAP1_TEXTURE_COORD_3 = Coordenadas de la textura (s, t, r).
   - GL_MAP1_TEXTURE_COORD_4 = Coordenadas de la textura (s, t, r, q).
+ u1:     Indica el valor mínimo del parametro u.
+ u2:     Indica el valor máximo del parametro u.
+ stride: Es la distancia entre un punto y el siguiente en el array.
+ order:  Número de puntos de control dentro de points.
+ points: Array con los puntos de control de la curva.
```

Con esta función se crea la ecuación que se usará para dibujar la curva en nuestro programa. Dependiendo del target que indiquemos, ese mismo tendremos que activarlo con glEnable(), del siguiente modo:

```
glMap1f(GL_MAP1_VERTEX_3, 0.0f, 1.0f, 3, 4, points);
glEnable(GL_MAP1_VERTEX_3);

glColor(1.0f, 1.0f, 1.0f);

glBegin(GL_LINE_STRIP);
    for(int i = 0; i <= 30; ++i)
        glEvalCoord1f((float) i / 30.0f);
glEnd();
```

Configurada la ecuación, lo siguiente sería pintar la curva trazando lineas de un punto a otro, y ello se puede realizar con la función:

```
void glEvalCoord1f (GLfloat u);
```

Que usando la ecuación, calcula el punto intermedio u que le indiquemos, y lo manda como un vértice a la máquina OpenGL.

3) La representación de una rejilla:

Muchas veces resulta útil dibujar una rejilla en el suelo, con el firme propósito de hacer un editor en el que situar objetos y demás. Para crear dicha rejilla tenemos:

```
void glMapGrid1f (GLint n, GLfloat u1, GLfloat u2);
```

Donde n representa el número de divisiones de la rejilla, y los parametros u1 y u2, el rango de valores que podremos usar. Definido ese detalle, para pintarla tenemos que llamar a:

```
void glEvalMesh1 (GLenum mode, GLint p1, GLint p2);
+ mode: GL_POINT, GL_LINE.
+ p1, p2: Rango de cuadrículas que calcular.
```

De este modo solo tendríamos que pintar la rejilla llamando a tan solo dos funciones:

```
glColor(1.0f, 1.0f, 1.0f);
glMapGrid1f(100, 0.0f, 100.0f);
glEvalMesh1(GL_LINE, 0, 100);
```

4) Superficies:

Pero todo esto sería mucho más práctico y útil poder usarlo también para crear superficies curvas, con las que dar cierta calidad a nuestra escena renderizada:

```
glMap2f(GL_MAP2_VERTEX_3, 0.0f, 10.0f, 3, 3, 0.0f, 10.0f, 9, 3, points);
glEnable(GL_MAP2_VERTEX_3);

glMapGrid2f(10, 0.0f, 10.0f, 10, 0.0f, 10.0f);
glEvalMesh2(GL_LINE, 0, 10, 0, 10);
```

Analizando un poco mejor este ejemplo nos damos cuenta de que para configurar la ecuación usamos la función glMap2f(), tras la cual tenemos que activar con glEnable() el target de tipo GL_MAP2_* que le hemos configurado. Luego con glMapGrid2f() creamos la rejilla de la superficie, y la dibujamos con glEvalMesh2(). Así pues, analicemos glMap2f():

```
void glMap2f (GLenum target, GLfloat u1, GLfloat u2, GLint ustride,
              GLint uorder, GLfloat v1, GLfloat v2, GLint vstride,
              GLint vorder, const GLfloat * points);
+ target:  Indica que es lo que representan los puntos de control.
   - GL_MAP2_VERTEX_3        = Coordenadas de puntos (x, y, z).
   - GL_MAP2_VERTEX_4        = Coordenadas de puntos (x, y, z, w).
   - GL_MAP2_INDEX           = Colores en modo paleta.
   - GL_MAP2_COLOR_4         = Colores RGBA.
   - GL_MAP2_NORMAL          = Coordenadas de las normales.
   - GL_MAP2_TEXTURE_COORD_1 = Coordenadas de la textura (s).
   - GL_MAP2_TEXTURE_COORD_2 = Coordenadas de la textura (s, t).
   - GL_MAP2_TEXTURE_COORD_3 = Coordenadas de la textura (s, t, r).
   - GL_MAP2_TEXTURE_COORD_4 = Coordenadas de la textura (s, t, r, q).
+ u1:      Indica el valor mínimo del parametro u.
+ u2:      Indica el valor máximo del parametro u.
+ ustride: Es la distancia entre un punto y el siguiente en el array.
+ uorder:  Número de puntos de control dentro de points en el eje u.
+ v1:      Indica el valor mínimo del parametro v.
+ v2:      Indica el valor máximo del parametro v.
+ vstride: Es la distancia entre un punto y el siguiente en el array.
+ vorder:  Número de puntos de control dentro de points en el eje v.
+ points:  Array con los puntos de control de la curva.
```

Hay que tener en cuenta, que el array de puntos tendría 3 dimensiones, la dimensión u, la dimensión v, y la dimensión del punto en sí. Luego para definir la rejilla tenemos la función:

```
void glMapGrid2f (GLint nu, GLfloat u1, GLfloat u2,
                  GLint nv, GLfloat v1, GLfloat v2);
```

Los parametros nu y nv indican la cantidad de divisiones que existen en la rejilla. Y el resto son para indicar los rangos que podemos usar en cada eje, en la siguiente función:

```
void glEvalMesh2 (GLenum mode, GLint i1, GLint i2, GLint j1, GLint j2);
+ mode: GL_POINT, GL_LINE, GL_FILL.
```

```
+ i1, i2: Rango de cuadrículas que pintar del eje u.
+ j1, j2: Rango de cuadrículas que pintar del eje v.
```

Y tras llamar a esta función se pintará en pantalla la superficie, en los intervalos que le hayamos indicado en los respectivos rangos de valores. Además de todo esto si queremos que se pinte la superficie con relleno y con las normales bien configuradas para una iluminación suave tendríamos que hacer:

```
glMap2f(GL_MAP2_VERTEX_3, 0.0f, 10.0f, 3, 3, 0.0f, 10.0f, 9, 3, points);
glEnable(GL_MAP2_VERTEX_3);

glEnable(GL_AUTO_NORMAL);

glMapGrid2f(10, 0.0f, 10.0f, 10, 0.0f, 10.0f);
glEvalMesh2(GL_FILL, 0, 10, 0, 10);
```

De este modo las normales de la superficie se calcularán automáticamente, ahorrandonos el trauma de hacerlo nosotros a pelo.

5) Aplicando texturas a las superficies:

Entre las páginas 421 y 426 hay un ejemplo de poner una superficie con una textura. Los pasos son sencillos, una vez tengamos creada y cargada en memoria la textura. Tan solo tendremos que llamar a la textura antes de pintar la superficie, y poner algo como esto:

```
glBindTexture(GL_TEXTURE_2D, texid);

glMap2f(GL_MAP2_VERTEX_3, 0.0f, 1.0f, 3, 3, 0.0f, 1.0f, 9, 3, points);
glMap2f(GL_MAP2_TEXTURE_COORD_2, 0f, 1f, 2, 2, 0f, 1f, 4, 2, texcoords);

glEnable(GL_MAP2_VERTEX_3);
glEnable(GL_MAP2_TEXTURE_COORD_2);

glMapGrid2f(10, 0.0f, 1.0f, 10, 0.0f, 1.0f);
glEvalMesh2(GL_FILL, 0, 10, 0, 10);
```

Y de este modo se pintaría una superficie con textura. El array de coordenadas de la textura tendrían tres dimensiones (u, v, (s, t)), de dos posiciones para la u, dos para la v, y las 2 para los valores (s, t).

6) NURBS:

El único problema de las curvas Bézier, es que cuantos más puntos le añadimos para definir la curva, esperando que esta quede más suavizada, más problemas encuentra para poder llevar a cabo su misión. Por ello existe otro sistema para crear curvas y superficies llamadas de tipo NURBS. Lo primero para manejar una nurbs es crear un objeto de tipo nurbs con la función:

```
GLUnurbsObj * gluNewNurbsRenderer (void);
```

Una vez creado el objeto, podemos configurar algunas de sus propiedades con la función:

```
void gluNurbsProperty (GLUnurbsObj * obj, GLenum property, GLfloat value);
+ property:
```

```
   - GLU_SAMPLING_TOLERANCE = Por defecto vale 50.0f, e indica en pixels
     el tamaño máximo que puede manejar al pintar la superficie.
   - GLU_DISPLAY_MODE = GLU_FILL (pinta la superficie como un conjunto de
     polígonos normales, es la opción por defecto), GLU_OUTLINE_POLYGON
     (pinta la superficie como un conjunto de polígonos generados por
     tesselación), GLU_OUTLINE_PATCH (pinta la superficie como un
     conjunto de polígonos parcheados).
+ value: Valor de la propiedad para asignar.
```

Una vez configurado el objeto, tendremos que pintarlo en la escena, usando las tres siguientes funciones básicas:

```
void gluBeginSurface (GLUnurbsObj * obj);

void gluNurbsSurface (GLUnurbsObj * obj, GLint uknot_count,
                      GLfloat * uknot, GLint vknot_count, GLfloat * vknot,
                      GLint u_stride, GLint v_stride, GLfloat * ctlarray,
                      GLint uorder, GLint vorder, GLenum type);
+ uknot_count: Número de knots para el eje u por cada punto.
+ uknot:       Knots para el eje u por cada punto.
+ vknot_count: Número de knots para el eje v por cada punto.
+ vknot:       Knots para el eje u por cada punto.
+ u_stride:    Distancia entre cada fila en el array.
+ v_stride:    Distancia entre cada punto en el array.
+ ctlarray:    Array de puntos de control.
+ uorder:      Tamaño de la dimensión u del array.
+ vorder:      Tamaño de la dimensión v del array.
+ type:        Equivale al parametro target de glMap2f().

void gluEndSurface (GLUnurbsObj * obj);

Ejemplo:

    float knots[8] = {...};
    float points[4][4][3] = {...};

    gluBeginSurface(nobj);
        gluNurbsSurface(nobj, 8, knots, 8, knots, 4*3, 3, points, 4, 4,
                        GL_MAP2_VERTEX_3);
    gluEndSurface(nobj);
```

No tan sencillo como pintar un triángulo, pero que se le va a hacer. Hay que comentar que por cada punto de control tiene que haber dos valores knots, que tienen que ser iguales a cualquier rango aceptado por el dominio paramétrico de u o de v. Y finalmente:

```
void gluDeleteNurbsRenderer (GLUnurbsObj * obj);
```

Con esta función podremos eliminar un objeto nurbs que hemos creado.

Capítulo 15: Efectos especiales.

1) Tableros o billboarding:

El primer "efecto" especia que vamos a echar un vistazo, es lo que llamamos "tableros". Se trata de poner una imagen en 2D en la pantalla, mirando siempre para el observador, sin importar desde donde se la mire. Efectivamente es lo mismo que pasaba con los

enemigos del DooM. ¿Y para qué podría servir esto que parece una chapuza? Bien, aunque parezca increible es bastante útil para hacer sistemas de partículas, que sí sirven para crear efectos especiales como el fuego, la nieve, la lluvia y demás. Lo primero para llevar a cabo la técnica es obtener la matriz actual de modelado con:

```
GLfloat view[16];

glGetFloatv(GL_MODELVIEW_MATRIX, view);
```

La idea principal, es multiplicar la matriz por la inversa, para invertir los efectos que la matriz aplica a todos los objetos que pintamos. Pero en este caso no haremos ninguna multiplicación, sino que cogeremos algunos datos que necesitamos para calcular las coordenadas, que simulen esa multiplicación.

Los datos que necesitaremos será, la primera fila y la segunda de la matriz tras dejarla en modo traspuesta. La primera fila la llamaremos la derecha y la segunda la arriba. Y las almacenaremos en un par de estructuras:

```
GLfloat right[3] = {view[0], view[4], view[8]};
GLfloat up[3]    = {view[1], view[5], view[9]};
```

Así que los cálculos que habría que hacer imaginando que queremos poner un cuadrado en pantalla, tendremos (px, py, pz) como punto central del cuadrado que pensamos pintar, y tendríamos que calcular lo siguiente:

```
GLfloat blc[3]; // Esquina inferior izquierda.
   blc[0] = px - right[0] * sizew - up[0] * sizeh;
   blc[1] = py - right[1] * sizew - up[1] * sizeh;
   blc[2] = pz - right[2] * sizew - up[2] * sizeh;

GLfloat brc[3]; // Esquina inferior derecha.
   brc[0] = px + right[0] * sizew - up[0] * sizeh;
   brc[1] = py + right[1] * sizew - up[1] * sizeh;
   brc[2] = pz + right[2] * sizew - up[2] * sizeh;

GLfloat urc[3]; // Esquina superior derecha.
   urc[0] = px + right[0] * sizew + up[0] * sizeh;
   urc[1] = py + right[1] * sizew + up[1] * sizeh;
   urc[2] = pz + right[2] * sizew + up[2] * sizeh;

GLfloat ulc[3]; // Esquina superior izquierda.
   ulc[0] = px - right[0] * sizew + up[0] * sizeh;
   ulc[1] = py - right[1] * sizew + up[1] * sizeh;
   ulc[2] = pz - right[2] * sizew + up[2] * sizeh;
```

Y entre las páginas 436 y 438 se encuentra un ejemplo sobre esta técnica, donde pintar una imagen de un cactus múltiples veces, para recrear un intento de desierto.

2) Sistemas de partículas:

¿Qué es un sistema de partículas? Es un mecanismo compuesto por una gran cantidad de entidades, de un tamaño visual bastante pequeño, que juntas llegan a formar efectos como el del fuego, o una tormenta de nieve, etcétera. Por ejemplo, en la tormenta de nieve, cada copo que cae sería una partícula, que tendría una serie de propiedades tales como la posición, la velocidad, cada cuanto se regeneran, y demás.

+ Las partículas:

Así que veamos cuales son los principales atributos de una partícula cualquiera, aunque hay que recordar que esto podría variar dependiendo de que es lo que queramos, y también hay que tener en cuenta, que algunas de las propiedades siguientes, podrían ser comunes a todas las partículas, por lo que no sería necesario que pertenecieran a la clase partícula:

+ Posición: Dado que vamos a representar un mundo 3D, necesitamos saber la posición en dicho espacio para cada partícula, con el fin de poder pintarla en pantalla. El único factor que altera esta propiedad es la velocidad.

+ Velocidad: La velocidad básicamente sirve para mover la partícula por el espacio que estamos representando, y puede verse afectado por factores como el viento o la gravedad, que pueden llegar a acelerar las partículas.

+ Duración: La duración o life span en inglés, indica el tiempo que le queda de vida a la partícula en el mundo que estamos representando.

+ Tamaño: Otra cosa que necesitamos saber, es el tamaño de la partícula, para poder pintarla correctamente en la pantalla.

+ Peso: El peso simplemente es usado para los cálculos de la física de la partícula, para ver con que intensidad le afectan diversas fuerzas.

+ Representación: También debemos tener en cuenta como vamos a representar la partícula, si va a ser con un punto, con una linea, o con un cuadrilátero que tenga una textura mapeada sobre este.

+ Color: En el caso de pintar puntos y lineas, necesitaremos saber de que color los tendremos que pintar, para cualquier partícula o para cada una.

+ Dueño: Y por último, necesitaremos saber quien es el dueño, el que ha creado o generado, dicha partícula, con el fin de poder controlar el sistema mejor.

Además de todo esto es bastante probable, que cada partícula tenga una serie de métodos para complementar su funcionalidad, como por ejemplo podría ser el actualizar el estado de cada una de ellas.

+ Los sistemas de partículas:

Visto lo que sería una partícula, el siguiente paso es ver el sistema, que controla a un conjunto de partículas. Para ello se requiere conocer:

+ Lista de partículas: Lo primero que se tiene que tener en cualquier sistema de partículas es una lista con todas ellas, para almacenar los datos de cada una y poder manejarlos y controlarlos de un modo ordenado.

+ Posición: La posición hace referencia al punto o area rectangular, desde la que se emiten las partículas. Básicamente es indicar cual es el punto o area de origen.

+ Ratio de emisión: Sirve para inciar con cuanta frecuencia se van creando partículas. También es necesario tener en cuenta hace cuanto fue creada la última partícula.

+ Fuerzas: También es útil tener en cuenta que fuerzas son aplicadas a las partículas, para poder calcular su movimiento de modo más realista.

+ Atributos por defecto: Al inicializar cada nueva partícula generada, hay que indicarle una serie de valores para todas sus propiedades, y muchas de ellas pueden ser valores por defecto, que tenemos que conocer desde el sistema.

+ Estado actual: Normalmente el estado consiste en saber si el sistema está activado o no, para saber si se tienen que emitir más partículas o no. Pero esto está sujeto a lo complejo que querramos realizar nuestro sistema.

+ Blending: Otro dato que tendremos que tener en cuenta es si las partículas tienen transparencia o no, configurándolo correctamente en OpenGL.

+ Representación: Finalmente como se vio en las partículas, tendremos que saber la forma en la que son representadas las partículas, que en el caso de ser todas iguales, pues será un dato que esté en el sistema.

Además como métodos necesitaremos por ejemplo:

+ Inicializar: Obviamente hará falta una función que inicialice el sistema entero, para poder controlar las partículas.

+ Actualizar: Función que actualizará el estado de todas las partículas.

+ Renderizar: Función para representar en pantalla todas las partículas.

+ Mover: Función para mover el sistema, moviendo así el punto de emisión.

+ Cambiar el estado: Función para cambiar el estado interno del sistema.

+ Get/Set de la fuerza: Funciones para obtener y modificar las fuerzas del sistema.

Aunque claro está esto es relativo a lo que querramos tener montado, y por ello nos podremos ver necesitados de añadir o eliminar funciones para hacer a nuestra medida, el sistema que querramos crear.

+ El gestor del sistema de partículas:

Podría llegarse a necesitar, si nuestra aplicación se complicara lo suyo, un gestor de sistemas de partículas. Cuya misión sería controlar todos los sistemas de partículas que tuviera asignados en su interior. Esto viene bien para tener ordenado en algunos casos este tema, en casos de objetos que necesiten de ello.

+ Implementación:

Entre las páginas de la 446 a la 450, hay un ejemplo de implementación de un sistema de particulas bastante sencillo, donde se monta una clase abstracta CParticleSystem, que será quien gestione un sistema de partículas.

+ Efectos con sistemas de partículas:

Hay muchisimos tipos de efectos que se pueden recrear con esta técnica, pero tiene mucho de experimentación todo ello, a la hora de crear efectos determinados:

+ Meditación: Hay que meditar bien sobre nuestro sistema, pues en la mayoría de los casos, sus propiedades representan aspéctos físicos en el mundo real. Por ejemplo, tenemos el humo que puede verse afectado por la fuerza del viento, o que su color cambia a medida que pasa su tiempo y se extingue.

+ La física: En todo esto, la física tiene un papel relativamente importante, a la hora de realizar un sistema realista. Pero es un arma de doble filo, ya que el modelo físico que querramos usar, tampoco puede ser muy pesado, para no saturar la CPU.

+ Mira lo que ya han hecho otros: Ya son muchos años de gente realizando programación gráfica, y muchos de ellos han logrado hacer cantidad de efectos, que no hay porque reinventar la rueda.

+ Experimenta con ello: Además también ocurre, que esos efectos ya creados por otros, pueden venir bien como base de experimentación, a la hora de crear otros efectos.

+ Ejemplo de tormenta de nieve:

(Páginas de la 451 a la 456).

3) Niebla:

La niebla es uno de esos efectos que ya venía incluido en OpenGL, aunque luego se encontaron formas más realistas de implementar este efecto ambiental. Como recurso es útil si no podemos renderizar escenarios muy grandes, tapando la nada con niebla, tal como ocurre en Silent Hill.

+ Niebla en OpenGL:

Para aplicar la niebla, OpenGL realiza una fusión mediante blending, en base a la distancia de cada pixel en la pantalla. Y para poder usar la niebla de OpenGL hemos primero de activar el parametro en la máquina de estados con:

```
glEnable(GL_FOG);
```

Y lo segundo es configurar las propiedades de la niebla con las funciones:

```
void glFogf  (GLenum pname, GLfloat   param);
void glFogi  (GLenum pname, GLint     param);
void glFogfv (GLenum pname, GLfloat * params);
void glFogiv (GLenum pname, GLint *   param);
+ pname: Propiedad a configurar.
   - GL_FOG_MODE = Se le puede pasar los valores GL_LINEAR, GL_EXP o
       GL_EXP2, que sirven para indicar cual es la ecuación que se
       utilizará para calcular el blending. Por defecto vale GL_EXP.
   - GL_FOG_DENSITY = Indica la densidad de la niebla, y es utilizado
       este parametro para las tres ecuaciones anteriores. Por defecto
       el valor de la densidad es de 1.0f.
   - GL_FOG_START = Indica con un solo valor (no un vector), cual es la
       distancia en la que empieza la niebla a surtir efecto.
   - GL_FOG_END = Indica cual es la distancia final de la niebla.
   - GL_FOG_INDEX = Color de la niebla indicado con un índice de la
       paleta de colores actualmente usada en el programa.
   - GL_FOG_COLOR = Color RGBA de la niebla. Por defecto es el negro.
```

Los tres tipos de ecuaciones que se pueden utilizar para la niebla son:

+ Lineal: BlendFactor = (end-z) / (end-start)
+ Exponencial: BlendFactor = e^(-densidad * profundidad)
+ Exponencial al cuadrado: BlendFactor = e^((-densidad * profundidad)^2)

Y así de simple es añadir la niebla nativa de OpenGL, que aprendiendo a controlarla bien, puede quedar el efecto bastante apañado. Aunque a día de hoy hay mejores técnicas para realizar este efecto.

+ Niebla volumétrica:

¿Cual es el principal defecto de la niebla nativa de OpenGL? Su principal defecto es que se basa en la distancia, con lo que solo podemos generar nieblas que nos rodean en todas las direcciones. Pero en la realidad las cosas no funcionan así rara vez, de hecho es bastante poco realista. En el mundo real, la niebla tiene puntos donde es más densa que en otros, quedando así la situación de estar un objeto cercano, más sumido en la niebla que otro más lejano.

Así que para poder hacer algo más realista, hemos de tener en cuenta que nos encontramos con diferentes volúmenes de niebla, esparcidos por nuestro mundo. Separados o pegados, pero ahí situados en el mundo con sus propiedades. Y para representar dichos volúmenes hay muchas formas de hacerlo tales como usar esferas, partículas, o mapas de nieblas (parecidos a los lightmaps). Aunque no es algo que se pinte directamente sobre la pantalla, y requiere complejas técnicas.

4) Reflejos:

En los capítulos anteriores se pudo ver como simular el reflejo de un suelo encerado, o el "reflejo" del entorno sobre una superficie (mediante trucos de texturas con el enviroment mapping). En el caso del reflejo sobre una superficie plana, como un suelo, la técnica es ligeramente sencilla, y sería esquemáticamente hablando lo siguiente:

```
DibujarSuelo();
DibujarObjeto();
glScale(1.0f, -1.0f, 1.0f);
DibujarObjeto();
```

Pero hay bastantes aspectos que no se vieron entonces, y que veremos ahora.

+ Reflejos de luces:

Por ejemplo las luces, hay que configurarlas bien para poder realizar correctamente el reflejo de un objeto. De ahí que haya que reposicionar las luces a la hora de pintar el objeto reflejado, que más o menos sería algo tal que:

```
// Pintar el objeto reflejado.
glPushMatrix();
    glScale(1.0f, -1.0f, 1.0f);
    PonerLuces();
    DibujarObjeto();
glPopMatrix();
```

```
// Pintar la escena normal.
PonerLuces();
DibujarSuelo();
DibujarObjeto();
```

+ Manejando el buffer de profundidad:

Un problema que suele ocurrir a menudo, es que necesitamos desactivar el buffer de profundidad, con el fin de poder pintar transparencias, como será en este caso. Y no solo transparencias, sino a la hora de querer hacer efectos raros con el buffer de stencil. Por ello no será raro que tengamos que invocar a glDepthMask(GL_FALSE), para desactivar la actualización del buffer de profundidad, y pasarle GL_TRUE luego para reactivarlo.

+ Manejando planos finitos con el buffer de stencil:

El único problema de hacer las cosas a lo bruto, como ocurre en el esquema que se ha mostrado, es que al pintar el objeto reflejado, nos ocurre que "sobresale" por debajo del plano del suelo. Y no es la clase de cosas que querríamos hacer, o que queden bien. De ahí que se utilice el buffer de stencil para realizar técnicas de clipping avanzadas al pintar nuestra escena:

```
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
glDepthMask(GL_FALSE);
glEnable(GL_STENCIL_TEST);

glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
glStencilFunc(GL_ALWAYS, 1, 0xFFFFFFFF);

DibujarSuelo();

glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
glDepthMask(GL_TRUE);

glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
glStencilFunc(GL_EQUAL, 1, 0xFFFFFFFF);

glPushMatrix();
    glScalef(1.0f, -1.0f, 1.0f);
    PonerLuces();
    glCullFace(GL_FRONT);
    DibujarObjeto();
    glCullFace(GL_BACK);
glPopMatrix();

glDisable(GL_STENCIL_TEST);

PonerLuces();
DibujarSuelo();
DibujarObjeto();
```

Un ejemplo muy parecido al que había en el capítulo sobre el stencil buffer, que simplemente pintabamos el suelo, para actualizar el contenido del buffer de stencil, para luego indicar que solo vamos a pintar sobre el area modificada. Además al pintar las cosas reflejadas probablemente hemos de cambiar la cara a la que se le hará el culling, para evitar que no se pinte el objeto.

+ Superficies con reflejos irregulares:

Hecho así el reflejo quedaría bastante nítido, increiblemente nítido realmente. Y muchas veces el reflejo de una superficie, es bastante distorsionado, por lo que podría ser interesante a la hora de realizar el blending del suelo, buscar posibles combinaciones alternativas, al blending normal, eso o recurrir a otras técnicas.

+ Manejando planos orientados arbitrariamente:

Finalmente todo esto que hemos visto, solo sirve para cuando el plano en cuestión está perpendicular a un eje, y eso no suele ser tan frecuente, a no ser que lo capemos nosotros obligando al diseñador que haga las superficies reflectantes perpendiculares a un eje de forma obligada. Pero en caso de que el plano esté orientado de forma arbitraria, tendremos que hacer más o menos:

+ Calcular como se tendría que manipular dicho plano para que quedara paralelo con el plano xz (o el que mejor nos venga), ya que sería un plano sencillo de usar.
+ Aplicando los cálculos anteriores para orientar el objeto correctamente hacia el plano, calcular cual sería la posición del objeto del objeto reflejado al otro lado del plano.
+ Y sobre esos datos aplicar el paso inverso al primero.

Un poco más claro sería por ejemplo que tenemos una pared, que si giramos un poco en el sentido de las agujas del reloj (negativo para OpenGL), se pondría en el plano yz. Tomamos la distancia del objeto desde su valor x hasta el origen de coordenadas, y lo multiplicamos por dos. Teniendo el angulo y la distancia, aplicamos el angulo a la inversa, posicionamos el objeto y lo escalamos a la inversa en el eje x para este caso, para finalmente pintarlo.

Y con esto sería "sencillo" realizar dichos reflejos, claro que lo que no es tan sencillo es realizar todos esos calculos, que básicamente consiste en encontrar varios angulos que tendría que girar el plano, y el objeto. Para luego calcular la posición reflejada y volver para atrás el giro. Haciendo lo que ya de por si era bastante pesado, más pesado aún.

5) Sombras:

Finalmente un tema que siempre da muchos dolores de cabeza es el de las sombras. Para poder hacer una representación mínimamente real, hay que al menos mostrar unas sombras que pudieran llegar a parecer mínimamente reales. Pero claro, esto no es trivial, de primeras porque en el mundo pueden haber múltiples fuentes de luz actuando en la escena, luego están todos los objetos que proyectan sombras sobre otros. Y mismamente hay sombras suaves y sombras fuertes. Las fuertes tienen una misma tonalidad, y las suaves serían en degradado.

+ Sombras estáticas:

Esta técnica es la menos costosa de cara a hacer cálculos, pero es la más pobre. Básicamente consiste en precalcular la iluminación de la escena, y poner lightmaps como texturas añadidas a los objetos, para simular su iluminación.

Para una escena sin movimiento, hasta podría quedar bien, pero el mayor problema de esto a la hora de moverse, es que no se puede "mover". Nada altera la forma de la sombra, con lo que no deberíamos mover las "luces" de la escena, porque aunque movieramos las texturas de iluminación, los objetos con los que nos fueramos encontrando por el camino, no interactuarían bien lanzando sombras.

+ Sombras proyectadas:

Este método se basa en la idea de proyectar un objeto, que está siendo iluminado, contra un plano de forma parecida a la que se haría al lanzar la la imagen que renderizamos contra la pantalla. Así que lo primero es conseguir la matriz que necesitamos para calcular la proyección de cada punto del objeto:

```
GLfloat myplane = {0.0f, 1.0f, 0.0f, 0.0f};

void SetShadowMatrix (GLfloat mat[16], GLfloat lightpos[4],
                      GLfloat plane[4])
{
    // El producto escalar entre un plano y la posición de la luz.
    GLfloat dot = plane[0] * lightpos[0] +
                  plane[1] * lightpos[1] +
                  plane[2] * lightpos[2] +
                  plane[3] * lightpos[3];

    // Primera columna de la matriz:
    mat[0]  = dot  - lightpos[0] * plane[0];
    mat[4]  = 1.0f - lightpos[0] * plane[1];
    mat[8]  = 1.0f - lightpos[0] * plane[2];
    mat[12] = 1.0f - lightpos[0] * plane[3];

    // Segunda columna de la matriz:
    mat[1]  = 1.0f - lightpos[1] * plane[0];
    mat[5]  = dot  - lightpos[1] * plane[1];
    mat[9]  = 1.0f - lightpos[1] * plane[2];
    mat[13] = 1.0f - lightpos[1] * plane[3];

    // Tercera columna de la matriz:
    mat[2]  = 1.0f - lightpos[2] * plane[0];
    mat[6]  = 1.0f - lightpos[2] * plane[1];
    mat[10] = dot  - lightpos[2] * plane[2];
    mat[14] = 1.0f - lightpos[2] * plane[3];

    // Cuarta columna de la matriz:
    mat[3]  = 1.0f - lightpos[3] * plane[0];
    mat[7]  = 1.0f - lightpos[3] * plane[1];
    mat[11] = 1.0f - lightpos[3] * plane[2];
    mat[15] = dot  - lightpos[3] * plane[3];
}
```

La variable plane, está relacionada con una ecuadión tal que ax + by + cz + d = 0, donde los valores a, b, c y d corresponden a los valores de la normal del plano, que es lo que está almacenado en plane. De este modo obtenida la matriz, ya solo hay que multiplicar cada punto que vayamos a pintar, pintando el objeto con el color negro para obtener la sombra.

Aunque esto conlleva diversos problemas con el buffer de profundidad, ya que los calculos pueden hacer salir valores que estén por detrás o por delante del suelo, provocando ciertos fallos visuales bastante molestos. Que pueden obligarnos a intentar pintar primero las sombras y luego el suelo, o desactivar el buffer de profundidad. Aunque también sería una buena forma usar la función glDepthFunc(GL_ALWAYS); al pintar la sombra, si tenemos cuidado de pintarla cuando todavía no esté oculta tras algún objeto. Así que un problema bastante serio que tener en cuenta.

Luego tenemos que las superficies sobre las que vamos a proyectar las sombras no son infinitas, por lo que tendremos posiblemente que usar el stencil buffer para realizar recortes, y controlar donde pintamos. Además al pintar la sombra tenemos que mirar si queremos aplicar blending, y desde luego hemos de desactivar las luces y las texturas (y posiblemente el buffer de profundidad), para evitar que la sombra se pinte mal. Quedando todo ello del siguiente modo:

+ Usar el stencil buffer para limitar el area de dibujado.
+ Crear la matriz de proyección de la sombra, y multiplicarla por la de modelview.
+ Desactivar la iluminación y el mapeado de texturas, y cambiar el color a negro.
+ Desactivar la actualización del buffer de profundidad.
+ Activar el blending.
+ Pintar la figura que proyectará la sombra.

¿Pero qué ocurre si tenemos varias fuentes de luz? Pues que por cada una de ellas tendremos que repetir el proceso, con cada objeto de la escena, pudiendo llegar a ser bastante costoso en tiempo de ejecución. Por lo que como de costumbre, cuanto más realismo, menos gratuito se vuelve el efecto.

A pesar de todas las posibles bondades de las sombras proyectadas, existen todavía algunos fallos bien severos, como la dificultad que tendrían para proyectar sombras objetos concavos sobre si mismos, por ejemplo. Además este efecto solo produce sombras fuertes, por lo que para poder producir las suaves, hay que usar blending y realizar la proyección moviendo la luz ligeramente, para difuminar el resultado. Tercero, al estar fusionando mediante blending la sombra con la superficie, cualquier brillo fuerte de una luz especular, seguirá estándo allí de forma visible, lo cual es un poco irracional. A pesar de todo, es un buen remedio si técnicas más potentes nos impiden llegar a una buena tasa de frames por segundo.

+ Sombras volumétricas con el stencil buffer:

Por último uno de los algoritmos que más llamaron la atención en el DooM 3, fue el de las sombras volumétricas. Es un algoritmo que ya se conocía desde hacía mucho tiempo (incluso antes del Quake 2). Pero por lo costoso que era, resultaba casi imposible usarlo en tiempo real. Las sombras volumétricas nos salvan de algunas imperfecciones del método anterior, por ejemplo los objetos podrán proyectar sombras sobre si mismos, y ya no habrá que preocuparse de que los puntos de brillo de la luz especular, aparezcan en medio de una sombra.

La idea principal de las sombras volumétricas, es que las fuentes de luz lancen lineas a través de los bordes de los objetos, definiendo así volumenes normalmente infinitos donde las sombras habitan, detrás de los objetos que han sido rociados con la luz. Después de definir todos los volúmenes, es cuando hay que definir que objetos se encuentran en el interior de estas sombras. Quedando en ese momento dos tareas: como encontrar los volúmenes y como determinar que puntos están en las sombras y cuales no lo están.

Así que el principal coste de este algoritmo es encontrar los volúmenes con las herramientas que tenemos. Para ello es necesario encontrar la "silueta" de cada objeto de la escena, que se forma tras recibir una luz. Para lograrlo hay que ir comprobando todos los polígonos de la escena, uno por uno. Y luego hay que ir mirando que puntos están o no en los volúmenes ya formados. Para ello nos pondremos desde el observador, contando cada vez que se entra o se sale de un volumen.

Y tras comentar que es teóricamente lo que se hace en el algoritmo, ahora explicaremos con un poco más de aproximación, como es la técnica usando los stencil buffers:

1) Renderizamos la escena de forma normal usando la luz emisiva y ambiental.
2) Realizamos una segunda pasada, en la que vamos a renderizar solo los volúmenes de sombras. Dado que no querremos ver los volúmenes como objetos pintados en la escena, desactivaremos la escritura en el buffer de profundidad y en el de color, y pondremos el de stencil en modo escritura. En esta ocasión, dado que vamos a contar cuantas veces los volúmenes son invadidos, aplicaremos el culling a las caras traseras de los polígonos (es decir pintaremos las caras frontales), y le indicaremos al buffer de stencil que se incremente cada vez que el test de profundidad tenga exito.
3) Realizamos una tercera pasada, similar a la segunda, pero en esta ocasión aplicaremos el culling a las caras delanteras (es decir pintaresmos las caras traseras), y decrementaremos el buffer de stencil cada vez que el test de profundidad tenga exito.
4) Por último se hará otra pasada con la geometría, pero esta vez renderizando con las luces difusas y especulares, pero solo en aquellas zonas donde el buffer de stencil sea igual a cero en su valor.

Y así es más o menos como se logra el "milagro" de las sombras volumétricas, aunque no es algo gratuito ya que hay que mandar a renderizar la geometría 4 veces, pero obteniendo unos resultados bastante buenos... Aunque nada es perfecto e incluso esta técnica tiene sus fallos, que principalmente son que si el observador se encontrara en el interior de uno de los volúmenes, el algoritmo se iría al carajo. De ahí que el propio John Carmack creara una variante conocida como "Carmack's Reverse", que en se basa en el fallo del buffer de profundidad, aunque requiere que se pinte la geometría bastantes más veces, con lo que se hace más lento el renderizado.

+ Ejemplo de reflejos y sombras:

Entre las páginas de la 469 a la 473, hay un ejemplo sobre como reflejar y proyectar sombras. Sombras que utilizan el método de sombras proyectadas.

Parte III: Construyendo un juego.

Capítulo 16: Usando DirectX: DirectInput.

1) Los mensajes de windows:

La forma de interactuar con una aplicación en windows, es enviándole mensajes el propio sistema operativo, con los horribles actos perpetrados por el usuario. Pero lógicamente estos siguen unos patrones y hay diferentes tipos de mensajes para recibir en la función de manejar eventos de windows, entre los que se encuentran:

+ WM_CHAR: Cada vez que una tecla es pulsada, se envía este mensaje con el valor del caracter asociado a la tecla, almacenado normalmente en wParam.
+ WM_KEYDOWN: Evento que se lanza al pulsar una tecla el usuario.
+ WM_KEYUP: Evento que se lanza al soltar una tecla el usuario.
+ WM_LBUTTONDOWN: El usuario ha pulsado el botón izquierdo del ratón.
+ WM_MBUTTONDOWN: El usuario ha pulsado el botón central del ratón.
+ WM_LBUTTONDOWN: El usuario ha pulsado el botón derecho del ratón.
+ WM_LBUTTONUP: El usuario ha soltado el botón izquierdo del ratón.
+ WM_MBUTTONUP: El usuario ha soltado el botón central del ratón.
+ WM_LBUTTONUP: El usuario ha soltado el botón derecho del ratón.
+ WM_MOUSEMOVE: El usuario ha movido el ratón de posición.
+ WM_MOUSEWHEEL: El usuario ha movido la rueda del ratón.

Normalmente esto en un juego no es muy usado, y lejos de explicarlo extensivamente, es más recomendable echar un vistazo a la ayuda del MSDN, donde viene la mayoría de la información relacionada con la api del windows.

2) Win32:

Aunque hay algunas funciones auxiliares, para obtener datos de una forma un poco más independiente, como por ejemplo para el teclado con:

```
SHORT GetAsyncKeyState (int vKey);

#define KEY_DOWN(vKey) (GetAsyncKeyState(vKey) & 0x8000)
#define KEY_UP(vKey)   !(GetAsyncKeyState(vKey) & 0x8000)
```

Con esta función sabremos el estado de la tecla que le pasemos, independientemente de si está actualizada en el estado interno de la aplicación, estado que modifican los mensajes. Si quisieramos saber el estado de las teclas, en base a lo que nos ha llegado por los mensajes, podríamos usar GetKeyState(). Luego para por ejemplo manejar un joystick tenemos la siguiente función:

```
MMRESULT joyGetPos (UINT joyID, LPJOYINFO pji);

typedef struct {
    UINT wXpos;
    UINT wYpos;
    UINT wZpos;
    UINT wButtons;
} JOYINFO;
```

Función a la que le indicamos el joystick que queremos comprobar, y obtenemos su estado posicional y los botones que tiene pulsados (JOY_BUTTON#). Aunque en caso de necesitar algo más existe joyGetPosEx(), que tiene mayor funcionalidad, pero que no vamos a entrar en detalle ya que con DirectInput las cosas son mucho mejores.

3) DirectInput:

Pero gracias a dios, existen mejores alternativas, como por ejemplo DirectX, con la que no tendremos que preocuparnos de muchos detalles del hardware que usemos, y podremos acceder a este con mayor velocidad.

Para poder usar esta librería tendremos que incluir la librería dinput.h, e importar los ficheros dinput8.lib y dxguid.lib a nuestro proyecto para que sean enlazados junto con nuestra aplicación. Además antes de hacer el include tenemos que indicar la versión:

```
#define DIRECTINPUT_VERSION  0x0800
#include <dinput.h>
```

4) Inicializando DirectInput:

Para poder usar DirectInput, hay que inicializarlo, pero no de la forma normal en la que se hace con los objetos, ya que DirectX al igual que muchas librerías de Microsoft está basada en el modelo COM, y para ello se usan funciones para inicializar los objetos, que nos devolverán estas mismas funciones el objeto creado e inicializado. En este caso es:

```
HRESULT WINAPI DirectInput8Create (
    HINSTANCE hInstance, // Instancia de la aplicación.
    DWORD dwVersion,     // Versión del DirectInput.
    REFIID riidltf,      // Identificador de la interfaz.
    LPVOID * ppvOut,     // Puntero a la variable que apunta a la clase.
    LPUNKNOWN punkOuter);

Ejemplo:

LPDIRECTINPUT8 lpDInput; // Objeto DirectInput.

if(FAILED(DirectInput8Create(GetModuleHandle(NULL), DIRECTINPUT_VERSION,
                             IID_IDirectInput8, (void**) &lpDInput, NULL)))
{
     // No se ha podido crear el objeto...
}
```

Así si todo va bien obtenemos el objeto principal con el que manejar el DirectInput, que nos permitirá crear otros objetos para acceder a los dispositivos y actualizar con ellos el estado de la entrada. Por último, el FAILED() es una macro que hace que si una función devuelve un error, la condición que devuelve sea true.

5) Añadiendo dispositivos de entrada:

Una vez obtenido el objeto DirectInput, lo siguiente es listar los dispostivivos conectados, crear el dispositivo, comprobar la configuración de este, listar los objetos de este dispositivo, configurar el formato de los datos, configurar el nivel de cooperación, modificar

algunas propiedades y adquirir el foco del dispositivo. Aunque hay algunos de estos pasos que son opcionales y resultan poco útiles para manejar por ejemplo el teclado y el ratón.

Pero ya que normalmente lo que más se usa en PC es el teclado y el ratón, lo que normalmente se hará para crear estos dispositivos será:

```
LPDIRECTINPUTDEVICE8 lpKeyb;
LPDIRECTINPUTDEVICE8 lpMouse;

DWORD flags = DISCL_BACKGROUND | DISCL_NONEXCLUSIVE; // | DISCL_NOWINKEY;

// Creación del dispositivo del teclado.
if(FAILED(lpDInput->CreateDevice(GUID_SysKeyboard, &lpKeyb, NULL)))
    return ERROR_CREATEDEVICE_KEYBOARD;

// Le indicamos la forma de guardar los datos
if(FAILED(lpKeyb->SetDataFormat(&c_dfDIKeyboard)))
    return ERROR_SETDATAFORMAT_KEYBOARD;

// Establecemos el nivel coperativo:
//   DISCL_FOREGROUND   -> El dispositivo solo funciona con la aplicacion
//                         activa.
//   DISCL_BACKGROUND   -> El dispositivo funciona aun sin estar la
//                         aplicacion activa.
//   DISCL_EXCLUSIVE    -> El dispositivo es exclusivo a la aplicacion.
//   DISCL_NONEXCLUSIVE -> El dispositivo no es exclusivo a la aplicacion.
if(FAILED(lpKeyb->SetCooperativeLevel(hWnd, flags)))
    return ERROR_SETCOOPERATIVELEVEL_KEYBOARD;

// Activamos el teclado.
if(FAILED(lpKeyb->Acquire()))
    return ERROR_ACQUIRE_KEYBOARD;


// Creación del dispositivo del ratón.
if(FAILED(lpDInput->CreateDevice(GUID_SysMouse, &lpMouse, NULL)))
    return ERROR_CREATEDEVICE_MOUSE;

// Le indicamos la forma de guardar los datos.
if(FAILED(lpMouse->SetDataFormat(&c_dfDIMouse)))
    return ERROR_SETDATAFORMAT_MOUSE;

// Establecemos el nivel coperativo:
//   DISCL_FOREGROUND   -> El dispositivo solo funciona con la aplicacion
//                         activa.
//   DISCL_BACKGROUND   -> El dispositivo funciona aun sin estar la
//                         aplicacion activa.
//   DISCL_EXCLUSIVE    -> El dispositivo es exclusivo a la aplicacion.
//   DISCL_NONEXCLUSIVE -> El dispositivo no es exclusivo a la aplicacion.
if(FAILED(lpMouse->SetCooperativeLevel(hWnd, flags)))
    return ERROR_SETCOOPERATIVELEVEL_MOUSE;

// Activamos el ratón.
if(FAILED(lpMouse->Acquire()))
    return ERROR_ACQUIRE_MOUSE;
```

Así ya tendríamos listos para usar el teclado y el ratón. Lo único reseñable es que al indicar la forma de guardar los datos tenemos c_dfDIKeyboard y c_dfDIMouse, que son variables de DirectX que definen como se tiene que almacenar los datos sobre dicho dispositivo. En el caso de dfDIKeyboard será un array de 256 caracteres, del que cada

posición representará una tecla (que están definidas como constantes, que siguen el patrón DIK_key). Y luego tenemos c_dfDIMouse que está asociado a la estructura:

```
typedef struct DIMOUSESTATE {
    LONG lX;
    LONG lY;
    LONG lZ;
    BYTE rgbButtons[4];
} DIMOUSESTATE, *LPDIMOUSESTATE;
```

Que contiene los movimientos relativos del ratón y el estado de los botones del ratón. Por cierto que el eje z es el relativo al movimiento de la rueda del ratón.

6) Obteniendo la entrada:

Para poder obtener los datos de los dispositivos tenemos varias funciones en la interfaz IDirectInputDevice8, como son GetDeviceState() para obtenerlos inmediatamente, o GetDeviceData() si son datos que se van almacenando en un buffer, como ocurriría si hicieramos un juego en el que queremos saber si el jugador ha pulsado una combinación de teclas. Un ejemplo para el teclado y el ratón sería:

```
if (FAILED(lpKeyb->GetDeviceState(sizeof(UCHAR[256]),
                                  (LPVOID) KeybState)))
     return ERROR_GETDEVICESTATE_KEYBOARD;

if (FAILED(lpMouse->GetDeviceState(sizeof(DIMOUSESTATE),
                                   (LPVOID) &MouseState)))
     return ERROR_GETDEVICESTATE_MOUSE;
```

Aunque para el teclado y el ratón no es necesario existe el método Poll(), que sirve para actualizar el estado de los datos que hay almacenados internamente en DirectX para gestionar los dispositivos, ya que algunos al cambiar su estado, no notifican a DirectX de estos sucesos, y la información quedaría desincronizada.

7) Apagando DirectInput:

Para dejar de usar cualquier dispostivo adquirido con DirectX tenemos que liberarlo usando las dos siguientes funciones, por ejemplo para el teclado:

```
lpKeyb->Unacquire();
lpKeyb->Release();
lpKeyb = NULL;
```

Y para el objeto DirectInput solo tenemos que llamar a Release():

```
lpDInput->Release();
lpDInput = NULL;
```

Se recomienda igualar a NULL los punteros de los objetos que hemos liberado, para poder controlar mejor que los hemos dejado de usar.

8) Ejemplo de un subsistema de entrada:

(Páginas de la 502 a la 513).

Capítulo 17: Usando DirectX: Audio.

1) ¿Qué es DirectX Audio?

DirectX Audio es la parte de esta api que trata los temas relacionados con el sonido, y para ello tiene dos subapartados que son DirectSound y DirectMusic, de los cuales el más usado es DirectSound, ya que tiene soporte para sonido 3D, algo muy importante a día de hoy en muchos juegos (sobre todo en los de primera persona). Juntas permiten que realicemos en nuestras aplicaciones cosas como lo siguiente:

+ Cargar y reproducir sonidos desde ficheros midi o wav.
+ Reproducir múltiples sonidos simultaneamente.
+ Planificar la reproducción de los sonidos con precisión.
+ Usar efectos de sonido 3D.
+ Aplicar efectos al sonido como la reverberación, la distorsión y otros.
+ Capturar de un micrófono sonido en formato wav.
+ Etcétera...

2) Cargando y reproduciendo audio con DirectMusic:

DirectMusic es puramente COM, y por ello no es necesario enlazar con ninguna librería, tan solo llamar a los includes dmusicc.h y dmusici.h, para poder utilizarla en nuestro programa. Una vez tenemos accesible la librería, para poder reproducir un sonido hay que seguir los siguientes pasos:

+ Inicializar el sistema COM llamando a CoInitialize().
+ Crear e inicializar el objeto Performance.
+ Crear el cargador.
+ Cargar un segmento.
+ Y reproducir el segmento.

Además hay algunas otras operaciones bastante útiles como saber si está siendo reproducido un segmento, o parar la reproducción de estos.

+ Inicializar el COM:

Para inicializar el sistema COM en nuestra aplicación será tan fácil como poner:

```
CoInitialize(NULL);
```

+ Crear e inicializar el Performance:

Para poder crear e inicializar el objeto principal, que realizará el trabajo para gestionar la reproducción con DirectMusic, tendremos que:

```
IDirectMusicPerformance8 * lpDMP    = NULL;
IDirectMusic8            * lpDMusic = NULL;
IDirectSound8            * lpDSound = NULL;
```

```
if(FAILED(CoCreateInstance(CLSID_DirectMusicPerformance, NULL,
                           CLSCTX_INPROC, IID_IDirectMusicPerformance8,
                           (void**) &lpDMP)))
   return ERROR_COCREATEINSTANCE1;

if(FAILED(lpDMP->InitAudio((IDirectMusic **) &lpDMusic,
                           (IDirectSound **) &lpDSound,
                           hWnd, DMUS_APATH_SHARED_STEREOPLUSREVERB, 64,
                           DMUS_AUDIOF_ALL, NULL)))
   return ERROR_INITAUDIO;

lpDSound->SetCooperativeLevel(hWnd, DSSCL_PRIORITY);
```

Con CoCreateInstance() creamos el objeto Performance, que nos servirá para inicializar con InitAudio() los sistema DirectMusic y DirectSound, a 64 canales en este ejemplo.

+ Crear el cargador:

Luego con la función CoCreateInstance(), también crearemos el cargador de ficheros de sonido de DirectMusic:

```
IDirectMusicLoader8 * lpDML;

if(FAILED(CoCreateInstance(CLSID_DirectMusicLoader, NULL, CLSCTX_INPROC,
                           IID_IDirectMusicLoader8, (void**) &lpDML)))
   return ERROR_COCREATEINSTANCE2;
```

Este objeto nos servirá para poder cargar los segmentos que vamos a reproducir.

+ Cargar un segmento:

Para poder cargar un segmento primero hay que configurar el directorio de busqueda, donde se supone que estarán los ficheros que vamos a reproducir:

```
char searchPath[MAX_PATH];
WCHAR wSearchPath[MAX_PATH];

GetCurrentDirectory(MAX_PATH, searchPath);
MultiByteToWideChar(CP_ACP, 0, searchPath, -1, wSearchPath, MAX_PATH);

lpDML->SetSearchDirectory(GUID_DirectMusicAllTypes, wSearchPath, FALSE);
```

Una vez tenemos el directorio, hemos de cargar el segmento con:

```
IDirectMusicSegment8 * lpSegment;
WCHAR wSongPath[MAX_PATH];

if(FAILED(lpDML->LoadObjectFromFile(CLSID_DirectMusicSegment,
                                    IID_IDirectMusicSegment8,
                                    wSongPath, (void**) &lpSegment)))
   return ERROR_LOADOBJECTFROMFILE;

if(FAILED(lpSegment->Download(lpDMP)))
   return ERROR_DOWNLOAD;
```

La función Download() lo que hace es descarga la banda del segmento al sintetizador.

+ Reproducir el segmento:

Para mandar a reproducir un segmento es tan fácil como:

```
if(FAILED(lpDMP->PlaySegmentEx(lpSegment, NULL, NULL, 0, 0,
                               NULL, NULL, NULL)))
   return ERROR_PLAYSEGMENTEX;
```

+ Parando el segmento:

Y parar la reproducción es con:

```
if(FAILED(lpDMP->StopEx(lpSegment, 0, 0)))
   return ERROR_STOPEX;
```

+ Saber si se está reproduciendo el segmento:

Para saber si un segmento está siendo reproducido tendremos que:

```
if(lpDMP->IsPlaying(lpSegment, NULL) == S_OK)
{
   // El sonido está siendo reproducido...
}
```

+ Controlar los segmentos en bucle:

Si queremos reproducir infinitamente en bucle un segmento tenemos que:

```
if(loop)
   hr = lpSegment->SetRepeats(DMUS_SEG_REPEAT_INFINITE);
else
   hr = lpSegment->SetRepeats(0);
```

También hay una función llamada SetLoopPoints(), que indica cual es el punto de inicio y cual el de fin, en un segmento, para el bucle de reproducción.

```
HRESULT SetLoopPoints (MUSIC_TIME mtStart, MUSIC_TIME mtEnd);
```

+ Eliminándolo todo:

Por último, para eliminar todo esto solo tenemos que usar:

```
lpSegment->Release();
lpSegment = NULL;

lpDML->Release();
lpDML = NULL;

lpDMP->CloseDown();
lpDMP->Release();
lpDMP = NULL;

lpDMusic->Release();
lpDMusic = NULL;

lpDSound->Release();
lpDSound = NULL;
```

Y liberando todos los objetos usados con Release, eliminamos todo lo que habíamos montado con DirectMusic. Además para el Performance tenemos que llamar a la función CloseDown(), que manda a parar todos los segmentos que están siendo reproducidos.

+ Ejemplo simple:

(Páginas de la 534 a la 548).

+ Los audiopaths:

Entre las páginas de la 548 a la 554 se explica lo que son los audiopaths, que sirven básicamente para controlar la reproducción del sonido bajo DirectMusic de una forma un poco más completa y compleja.

3) Sonido 3D:

Para mejorar la inmersión en nuestra aplicación o videojuego, podemos añadirle sonido 3D, que simula de forma bastante eficiente, aunque no perfecta, como escucharíamos sonidos que estén localizados en diferentes localizaciones del espacio.

+ Las coordenadas del sonido 3D:

Para representar una coordenada en el espacio 3D de DirectX tenemos:

```
typedef struct {
    float x;
    float y;
    float z;
} D3DVECTOR;
```

Que funciona exactamente igual que OpenGL salvo para el eje Z, que está invertido con respecto al de OpenGL.

+ El buffer 3D en DirectSound:

Cualquier sonido 3D en nuestra simulación se almacenaría en IDirectSound3DBuffer8, y estos sonidos tienen que ser de un solo canal, es decir mono, y no estereo. Para crear un sonido 3D con DirectMusic se tendría que hacer:

```
IDirectMusicAudioPath8 * AudioPath;
IDirectSound3DBuffer8  * SoundBuffer;

if(FAILED(lpDMP->CreateStandardAudioPath(DMUS_APATH_DYNAMIC_3D, 64,
                                         TRUE, &AudioPath)))
   return ERROR_CREATEAUDIOPATH;

if(FAILED(AudioPath->GetObjectInPath(DMUS_PCHANNEL_ALL,
          DMUS_PATH_BUFFER, 0, GUID_NULL, 0, IID_IDirectSound3DBuffer8,
          (void **) &SoundBuffer)))
   return ERROR_GET3DBUFFER;
```

+ Configurando los parametros 3D:

Además cada buffer tiene una serie de propiedades que pueden ser obtenidas o modificadas con las funciones Get/Set del objeto. Estas son:

+ MaxDistance: Distancia máxima a la que se escuchará el sonido.
+ MinDistance: Distancia mínima para que se empiece a escuchar el sonido.
+ Mode: Modo para ser procesado de forma normal (la posición es absoluta al mundo), relativa a la cabeza (la posición es relativa a la cabeza del escuchante), desactivado (el sonido deja de ser 3d y parece como si saliera siempre del centro de la cabeza).
+ Position: Posición del sonido.
+ ConeAngles: Ángulo del cono de emisión del sonido.
+ ConeOrientation: Orientación del cono de emisión del sonido.
+ ConeOutsideVolume: Volumen fuera del cono de emisión del sonido.
+ Velocity: Velocidad que tiene el objeto que emite el sonido.

Además con la función SetAllParameters() podemos cambiar todas las propiedades a la vez, o con GetAllParameters() obtenerlas todas de golpe.

+ El "escuchante" en DirectSound:

Además existe un objeto llamado Listener, que hace referencia a las propiedades del sujeto que está escuchando los sonidos. Para obtenerlo tenemos que:

```
IDirectSound3DListener8 * Listener;

if(FAILED(AudioPath->GetObjectInPath(DMUS_PCHANNEL_ALL,
          DMUS_PATH_PRIMARY_BUFFER, 0, GUID_NULL, 0,
          IID_IDirectSound3DListener8, (void **) &Listener)))
   return ERROR_GETLISTENER;
```

Además el listener tiene una serie de propiedades que pueden ser obtenidas o modificadas con las funciones Get/Set del objeto. Estas son:

+ DistanceFactor: Factor distancia del listener.
+ DopplerFactor: Factor de doppler del listener.
+ Orientation: Orientación del listener.
+ Position: Posición del listener.
+ RolloffFactor: Factor de roll-off del listener.
+ Velocity: Velocidad a la que se está moviendo el listener.

Además con la función SetAllParameters() podemos cambiar todas las propiedades a la vez, o con GetAllParameters() obtenerlas todas de golpe.

+ Un ejemplo 3D:

(Páginas de la 562 a la 578).