LINUX • JAVA • VISUAL BASIC • DEMOSCENE • VISUAL C++ • PROGRAMACIÓN 3D • SQL

PROGRAMACIÓN ACTUAL

AÑO 2 • NÚMERO 13

ARGENTINA 10$ • CHILE 3000$ • PORTUGAL 1250 ESC (CONT)

POWER J 2.0

LA SOLUCIÓN PARA LAS EMPRESAS

SÓLO 995 ptas

SECCIÓN NIVEL BÁSICO

386 en **ENSAMBLADOR**, Clases en **PROGRAMACIÓN C++**, Grafos en **CURSO DE PROGRAMACIÓN**

SECCIÓN BAJO NIVEL

Problemas con mapeado de texturas en **ENTORNOS 3D**, Modos **SVGA** en **PROGRAMACIÓN GRÁFICA**, Texto en modo gráfico **DEMOSCENE**

SECCIÓN LAS EMPRESAS DEMANDAN

POO en **VISUAL BASIC 5.0**, ActiveX en **VISUAL C++ 5.0**, Las DBtools.h en **BASES DE DATOS**, **SET** en **COMERCIO ELECTRÓNICO**, Configuración en **WINDOWS NT**

SECCIÓN CAMPUS ACTUAL

Control de versiones en **LINUX ACTUAL**, Utilidades **TCP/IP** en **REDES LOCALES**, Modelo de objetos en Javascript en **WWW**

ACTIVEX Y VISUAL C++

EN EL CD ROM

SOFTWARE GNU

- Compilador GNU C v2.7.2.1 para DOS.
- Compilador GNU C v2.80 para Windows 95.
- Compilador GNU GCC v2.80 para Unix.

DEMOS

- Norton Antivirus para Windows 95, NT 4.0
- Norton CrashGuard.
- Visual Cafe 1.0.
- DIV GAMES STUDIO

Y UTILIDADES DE PROGRAMACIÓN

• ActiveFile 1.2 • AddSoft 1.12 • CGI Expert 3.03b • CodeSMART 2.2 • CooScale 1.1 • eAuthor Help 2.05 preview 5 • EZ Macros 3.0 • Form To VB Export Wizard 1.3 • HexDecCharEditor 1.02 • Lf Directory Tree 1.2 • MultiLanguage Pack 2.01 • PiXCL Tools 4.1 • Project Analyzer 4.1.05 • Setup Specialist 97b • Stub Generator 1.1 • XShell 1.0 • y muchas más.

MODELO DE OBJETOS EN JAVASCRIPT AVANZADO

Prensa Técnica

8 413042 087779 00013

CURSO DE PROGRAMACION GRAFICA

Nivel: Avanzado

Capítulo 13º

Autor: Santiago Romero

Trabajando en SVGA (II)

Como vimos en la anterior entrega, existen dos posibilidades a la hora de trabajar con gráficos en PC. La primera de ellas consiste en utilizar un modo de vídeo específico bajo unas condiciones predeterminadas al desarrollar nuestra aplicación o juego. Esto es lo que ocurre cuando se decide utilizar el modo 13h, o modos SVGA de 8 bits por píxel. La segunda consiste en adaptarse a las características del sistema durante el proceso de carga de la aplicación.

APLICACIONES ESPECIFICAS

Supongamos que elegimos un modo de 8bpp (p. ej. 640x480x256) y pantalla completa para la realización de nuestra aplicación/juego. En estas condiciones se tiene prediseñado todo el código y gráficos de antemano, sabiendo que nada va a cambiar durante la ejecución de la aplicación (ejemplo: juegos MSDOS 256 colores y aplicaciones DX full screen en 8bpp).

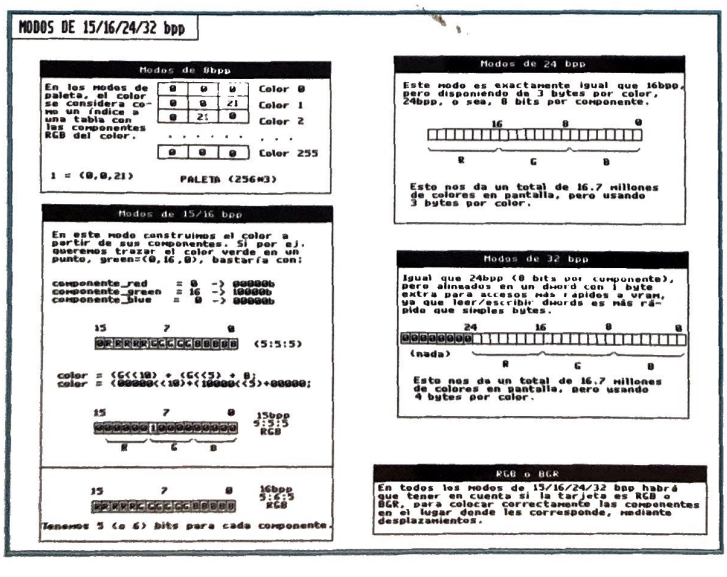
Mediante este método de trabajo limitamos la posibilidad de utilizar aplicaciones *windowed*. Bajo Windows95, si diseñamos una aplicación GDI (que use el *Interface de Dispositivo Gráfico* estándar de Windows), el mismo GDI es capaz de hacer conversión automática al realizar el volcado del bitmap si la profundidad de color (bpp) del DC (contexto de dispositivo) destino es diferente de la origen (mediante BitBlt()). Por supuesto, esto resultará más lento que realizar una sola conversión durante el inicio de la aplicación, como se comentó en la anterior entrega.

ADAPTACION AL DISPLAY

Una segunda posibilidad representa el desarrollar un programa independiente del dispositivo o susceptible de cambios en el display durante la ejecución del mismo.

En diferentes tarjetas los modos *HiColor* (15/16bpp) pueden definirse mediante 5 bits para la componente rojo (R), 5 para la verde (G) y 5 para la azul (B) (5:5:5 = 15 bits), aunque también es muy común encontrar tarjetas que trabajen con un sistema de 5:6:5 (6 bits para la componente verde). Además, la tarjeta puede trabajar con RGB (atendiendo a cómo

FIGURA 1. DIFERENTES FORMATOS DE PIXEL.



Vimos en la anterior entrega los diferentes modos SVGA, así como sus fundamentos a nivel de la propia tarjeta. Este mes desgranamos su tratamiento como mapa de bits, trabajando con las distintas profundidades de color, y realizando conversiones de imágenes entre ellas.

están situadas las componentes dentro del word o dword) o BGR, lo cual obliga a definir diferentes modos de trabajo según las características de la misma, así como saber convertir los gráficos entre estos formatos. Para comprender los diferentes modos vamos a ver cómo trabaja la primitiva PutPixel() dentro de ellos.

MODOS DE 8BPP

Los modos de 8bpp han sido ya comentados durante el curso. Cada elemento del buffer de trabajo (cada byte) constituye un índice (apuntador) dentro de una tabla de componentes RGB (la paleta) que el dispositivo gráfico utiliza para representar cada byte del mapa de bits. Estos modos son rápidos y sencillos, requieren poca memoria (al ser cada elemento de tipo byte) y han sido los más utilizados hasta ahora. Su organización interna puede observarse en las figuras 1 y 2.

```
// tamaño de la pantalla/buffer
#define  ANCHO  640
#define  ALTO   480

byte buffer[ANCHO*ALTO] ;
void PutPixel_8bpp( x, y, byte color )
{
  buffer[(y*ANCHO)+x] = color ;
}
```

La lectura es igual de sencilla; para el acceso a un elemento (x,y) basta con aplicar una conversión directa de ambas coordenadas sabiendo la anchura lógica de cada *scanline* del buffer/pantalla.

MODOS DE 16BPP 5:5:5

El modo 5:5:5 no es exactamente de 16 bits por píxel, sino de 15 (5 bits para RED, 5 para GREEN y 5 para BLUE suman un total de 15 píxeles). El bit más significativo puede ignorarse a la hora de trazar o leer píxeles (*modo de 15bpp* o *32kColour*). En las figuras 1 y 2 podemos ver un esquema interno de este modo. Atendiendo al mismo:

```
// Colocación de cada componente:
#define MAKERGB_15bpp(r,g,b) (word) ((r<<10)|(g<<5)|(b))

word buffer[ANCHO*ALTO] ;
```

```
void PutPixel_15bpp( x, y, int r, int g, int b )
{
  buffer[((y*ANCHO)+x)] = MAKERGB_15bpp(r, g, b) ;
}
```

Lo primero que notamos en el listado anterior es que el buffer se ha declarado usando componentes de tipo *short* (2 bytes), ya que en 15bpp cada píxel es representado por 2 bytes (16/8=2). Siguiendo con el ejemplo, puede observarse que se ha definido una macro *MAKERGB* que acepta las 3 componentes (con valores entre 0 y 31 cada una (5 bits)), y construye un *word* (color/píxel) mediante desplazamientos y operaciones lógicas OR (|). Vamos a examinar detenidamente dicha macro:
Supongamos que queremos trazar un píxel de valores RGB (16,9,1). Con esos datos, ejecutamos una llamada a la función PutPixel:

```
PutPixel_15bpp( x, y, 16, 9, 1 );
```

Para la llamada a la macro MAKERGB se tiene que:

```
r = 16 = 10000b
g =  9 = 01001b
b =  1 = 00001b
```

El contenido final del byte a escribir (5:5:5) ha de quedar de la siguiente manera (debido a la organización interna del modo):

```
(555RGB): bit 15 ->  0rrrrrgggggbbbbb  <- bit 0
```

Aprovechando los desplazamientos (<<) de bits y la operación OR (que permite introducir valores sin modificar los bits existentes aparte de los activados ya en la variable), se deduce que:

```
color = (r<<10) | (g<<5) | (b) ;
```

Los desplazamientos dejan cada componente en el lugar que le corresponde dentro del *word* final, y las operaciones OR realizan la composición de todas las componentes en un solo elemento de memoria.

Cada elemento del buffer de trabajo (cada byte) constituye un índice (apuntador) dentro de una tabla de componentes RGB

El resultado final es, pues, 0100000100100001b, que es el parámetro pasado a la función PutPixel. Esta función se encarga simplemente de almacenar este valor en el elemento correspondiente del array de tipo *word*. Nótese que las funciones que se están analizando son todos en formato RGB. Para trazar el mismo píxel en una tarjeta BGR basta con cambiar los valores de los *shifts* a *((b<<16)|(g<<8)|(b))*, aunque luego veremos funciones de conversión para este tipo de modos.

MODOS DE 16BPP 5:6:5

El modo 5:6:5 constituye el modo *HiColour/64KColour*, y es exactamente igual al anterior (figura 1), con la única excepción de que ahora se utilizan 5 bits para RED, 6 para GREEN (rango 0-63) y 5 para BLUE. Basta pues con cambiar los valores de los desplazamientos:

```
// Colocación de cada componente:
#define MAKERGB_16bpp(r,g,b) (word) ((r<<11)|(g<<5)|(b))
```

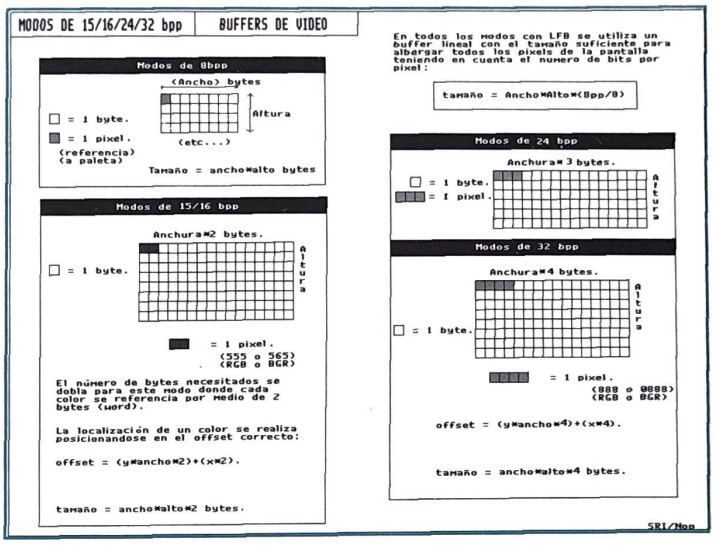


FIGURA 2. TAMAÑOS DEL BUFFER DE VIDEO.

La rutina PutPixel() realiza el mismo trabajo que la anterior, almacenar el *word* resultante en el buffer de trabajo/vram.

MODOS DE 24BPP 8:8:8

En las figuras 1 y 2 puede observarse la organización del modo de 24bpp, en el que se aprecia la utilización de 8 bits por componente, lo que requiere (24bpp/8=3) 3 bytes para describir un color. Al no existir ningún tipo de datos de 3 bytes para almacenar un color puede optarse por usarse un *long o int* (4 bytes) o 3 bytes individuales.

a) Tratado individual de las componentes:

```
#define  NUMBPP  3

char buffer[ANCHO*ALTO*NUMBPP] ;

void PutPixel_24bpp_1( x, y, byte r, byte g, byte b )
{
  long offset = ((y*ANCHO)+x)*NUMBPP;
  buffer[offset]   = r;
  buffer[offset+1] = g;
  buffer[offset+2] = b;
}
```

Se define un array con el tamaño necesario para todos los elementos (ancho*alto*3bytesporpixel), se calcula el offset perteneciente a dicho píxel ((y*ancho+x)*3), y se almacenan en memoria las 3 componentes del color especificado.

b) Aglutinación de las componentes:

```
char buffer[ANCHO*ALTO*NUMBPP] ;

// Colocación de cada componente:
#define MAKERGB(r,g,b) (dword) ((r<<16)|(g<<8)|(b))

void PutPixel_24bpp_2( x, y, dword color )
{
  long offset = ((y*ANCHO)+x)*NUMBPP;
  buffer[offset]   = (byte) (r>>16) & 255;
  buffer[offset+1] = (byte) (g>>8)  & 255;
```

```
  buffer[offset+2] = (byte) (b & 255);
}
```

En esta rutina pasamos el color ya compuesto como parámetro para descomponerlo en los bytes individuales. Para ello desplazamos la componente deseada a los 8 últimos bits del *word* y hacemos un AND con 255, para poner a cero todos los bits del long excepto los 8 últimos (255=11111111b), que convertimos a byte para escribir en el buffer (aunque el *typecast* ya estaría implícito en la escritura).

VENTAJAS Y DESVENTAJAS

Los modos de 256 colores tienen grandes desventajas, como una mayor dificultad de creación de sombreados (efectos de luces/sombras), para los cuales será necesario definir una paleta preparada para diferentes tonalidades (por ejemplo, 64x4 tonalidades), o utilizar elaboradas *look-up-tables* (*LUT*, o tablas precalculadas), donde podamos encontrar equivalentes brillantes/oscuros de los 256 colores. Esto lo veremos en la próxima entrega, junto con rutinas de "*best match*".
También nos encontramos con el típico problema de paletización de bitmaps. Imaginemos 2 sprites cada uno de ellos con su paleta propia y se desea visualizarlos juntos en pantalla. Al no tener idéntica paleta deberemos proceder a construir una paleta óptima para ambos y *mapear* los colores de los sprites a esta paleta (esto es engorroso, por ejemplo, para un juego aunque no lo hagamos en tiempo real).

Escribir un dword en memoria puede no ser tan rápido si no lo hacemos adecuadamente

Precisamente lo contrario ofrecen los modos de 16bpp o más. A cambio de requerir una mayor cantidad de videomemoria (y, por lo tanto, menor número de fps al haber más datos, más pérdidas de caché, etc), ofrecen la individualidad de cada píxel en pantalla, lo que nos permitirá modificar la intensidad/color de un píxel sin modificar los colores similares en la imagen (es decir, ya no es posible modificar la paleta puesto que no existe). Esto da la posibilidad de realizar efectos de luces/sombras añadiendo/restando valores constantes a todos los píxeles de un sprite en pantalla, como los conocidos *flares* que pueden verse en muchas demos/juegos.

OPTIMIZACIONES SIMPLES

La mejor optimización que puede realizarse para acelerar el trazado de gráficos es, obviamente, no utilizar llamadas a la rutina PutPixel para la realización de gráficos complejos (sino funciones específicas). En la realización de estos, siempre que sea posible, se deberá de *tratar de aglutinar colores en dwords:*

- Si sabemos que todos los bitmaps/sprites de nuestra aplicación son de una anchura múltiplo de 4, al trazarlos se leerán 4 bytes (un *dword*) y se escribirá éste en el buffer destino.
- En el caso de 15/16bpp, tratar de escribir 2 píxeles cada vez utilizando para ello un *dword* (2 píxeles x 2 bytes = 4 bytes).
- En los modos de 24bpp, escribir (si es posible) 4 píxeles cada vez, utilizando para ello 3 *dwords* (4 píxeles x 3 bytes = 12 bytes=3dw).

De esta manera, además de hacer accesos a memoria más rápidos, se estará desenrrollando el bucle a 1/4, 1/2 o 1/3 de las iteraciones.

Modos de 32bpp 0:8:8:8

Este modo es similar al anterior: posee 8 bits para cada componente, pero cada color se representa por 4 bytes (un dword), ignorando el byte superior de los 4 (ORGB). Esto se hizo así para aprovechar que el bus de los PC's escribe de manera más rápida y eficiente un DWORD que un BYTE (aunque pueda parecer que debería tardar más tiempo), debido a su arquitectura interna.

```
dword buffer[ANCHO*ALTO] ;

// Colocación de cada componente:
#define MAKERGB(r,g,b) (dword) ((r<<16)|(g<<8)|(b))

void PutPixel_32bpp( x, y, dword color )
{
  buffer[((y*ANCHO)+x)] = color;
}
```

Es el modo más sencillo de trabajar (aparte del de 8bpp), el más completo (16,7 millones de colores) y el que más memoria necesita (hasta 4 veces más).

OPTIMIZACIONES AVANZADAS

a) Alineamiento de datos en memoria.
Escribir un *dword* en memoria puede no ser tan rápido si no lo hacemos adecuadamente. Si la escritura/lectura no se realiza en una posición de memoria múltiplo de 4 (0, 4, 8...) es decir, si no está alineado con el tamaño del *dword* (por ejemplo, escribir en Offset=1/2/3), sufriremos una penalización de 3 ciclos de reloj por acceso. Al trazar un sprite por medio de instrucciones MOVSD (o por medio de un bucle for() con aritmética de punteros a long); si la posición del sprite es, por ej., (1,0) (desalineado), se nos penalizará con 3 ciclos de reloj *por cada elemento que escribamos a memoria* (por eso existe en los compiladores una opción de alineación de datos a 1/2/4 bytes).
Una forma de solucionar esto es tener 4 rutinas diferentes ((DrawSpr0 a DrawSpr3) que dependiendo del resto de la coordenada X destino del sprite (0,1,2,3)), escriban los píxeles desalineados como bytes y el resto como *dwords*. Se debe llamar a una u otra en función de la coordenada x del sprite (esto, es, *NumRutina = SpriteX & 3*, o *NumRutina = SpriteX % 4 ;* (el resto de dividir por 4)).
La primera de ellas, DrawSpr0, debe ser llamada cuando la coordenada. X es múltiplo de 4 (spriteX & 3 da 0 cuando ocurre esto). Ésta se encarga de volcar *dwords* sin preocuparse, pues el destino está alineado a 4.
A modo de ejemplo, supongamos que trazamos el sprite en (3,0). La rutina llamada será DrawSpr3, que deberá escribir 1 byte del sprite en memoria y, en ese momento, el siguiente píxel ya estará en SpriteX = 4 (alineado), por lo que podemos volcar el resto del sprite con *dwords* (cuidado con los últimos bytes del mismo). De similar manera habremos de crear DrawSpr1 (escribir primero 3 bytes para alinear, y luego *dwords*) y DrawSpr2 (escribir 2 bytes y luego *dwords*).

b) Uso de FPU o MMX.
Puede usarse tanto la FPU (coprocesador matemático) como las instrucciones MMX para copiar 64 bytes de golpe (más rápido que el equivalente en *movsd*) de una posición de memoria a otra. Veamos un ejemplo de copia con FPU:

```
fild qword ptr [eax]
fild qword ptr [eax+8]
fild qword ptr [eax+16]
fild qword ptr [eax+24]
fild qword ptr [eax+32]
fild qword ptr [eax+40]
fild qword ptr [eax+48]
fild qword ptr [eax+56]
fxch st(1)
fistp qword ptr [ebx+48]
fistp qword ptr [ebx+56]
```

```
fistp qword ptr [ebx+40]
fistp qword ptr [ebx+32]
fistp qword ptr [ebx+24]
fistp qword ptr [ebx+16]
fistp qword ptr [ebx+8]
```

El anterior fragmento de código copia 64 bytes desde la dirección apuntada por EAX a EBX. Las instrucciones *FILD* (*Fpu Integer LoaD*) almacenan 8 bytes cada una en la pila del coprocesador, y las instrucciones *FISTP* (*Fpu Integer Store and Pop*) saca el último valor de la pila de la fpu y lo almacena (como un *integer*) en la posición indicada. La primera instrucción de almacenamiento (*fistp*) se ha intercambiado por la segunda (+56 por +48), para no producir 2 accesos seguidos a la misma posición de la pila y, por tanto, no obtener ninguna penalización. Este tipo de copia es muy recomendable con buffers de volcado grandes (640x480x16bpp); tan solo hay que utilizar un bucle en incrementemos EBX y EAX en 64 hasta realizar el volcado.

LISTADO 1. FUNCIONES DE CONVERSION 32BPP->XBPP

```
//-------------------------------------------------
short Convert32To15( long color )
{
   long result, r, g, b;

   r = (color>>16) & 255;
   g = (color>>8)  & 255;
   b = color & 255;

   r = r>>3;
   g = g>>3;
   b = b>>3;

   result = (r<<10) | (g<<5) | b;
   return( (short) result );
}

//-------------------------------------------------
short Convert32To16( long color )
{
   long result, r, g, b;

   r = (color>>16) & 255;
   g = (color>>8)  & 255;
   b = color & 255;

   r = r>>3;
   g = g>>2;
   b = b>>3;

   result = (r<<11) | (g<<5) | b;
   return( (short) result );
}

//-------------------------------------------------
long Convert32To24( long color )
{
  return( color & 0x00FFFFFF ) ;
}
```

CONVERSION ENTRE LOS DIFERENTES FORMATOS

Para convertir todos los formatos de profundidad de color entre sí tenemos 30 posibilidades distintas (8->15, 8->16, etc.), así que una forma muy práctica de hacer las conversiones es convertir el color desde cualquier formato a 32bpp y después de 32bpp al formato deseado. De esta manera, sólo necesitamos 7 funciones de conversión, 4 de ellas desde Xbpp a 32bpp, y las otras 3 desde 32bpp a Xbpp. No vamos a convertir de 32bpp a 8bpp porque eso requiere un algoritmo de cuantización de color (tenemos que crear una paleta para el bitmap, y sólo dispondremos de 256 colores para ello).

CONVERSION 32BPP->XBPP

Puesto que son las más sencillas, vamos a ver primero las conversiones desde 32bpp a cualquier otro formato. Como puede verse en el listado 1, se basa simplemente en desplazar cada componente perdiendo en el >> los bits que le sobran. Es decir, si queremos pasar un RED de 8 bits a 5 bits, bastará con desplazarla >>(8-5) = >>3.

CONVERSION XBPP->32BPP

En el listado 2 pueden observarse los pasos necesarios para cada una de estas conversiones. Para pasar de 8bpp a 32bpp no hay ninguna dificultad, tan sólo hay que pasarle las componentes (de 8 bits, no de 6) y colocarlas en su lugar adecuado en el *dword*. Se ha optado por pasarlas compuestas dentro de un *long* (similar a 24bpp).
Un sencillo problema surge en las conversiones de 5 o 6 bits a 8. Como puede verse en *Convert15bppTo32bpp()*, no basta con desplazar la componente 3 bits (8-5=3), porque haciendo esto dejamos 3 bits a cero en la parte inferior del byte :

11111 red << 3 = 11111000

Esos 3 ceros son incorrectos y los hemos de cubrir con la parte superior de la componente (antes de desplazarla), de esta manera (5-3=2):

Red = (red<<3) | (red>>2) ;

Y de la misma manera con el resto de componentes y conversiones.

LISTADO 2. FUNCIONES DE CONVERSION XBPP->32BPP

```
//-------------------------------------------------
long Convert8To32bppRGB( long colorRGB )
{
   long result, r, g, b;

   r = (colorRGB>>16) & 255;
   g = (colorRGB>>8)  & 255;
   b = colorRGB & 255;
   result = (r<<16) | (g<<8) | b;

   return( result );
}

//-------------------------------------------------
long Convert15To32bppRGB( short color )
{
   long result, r, g, b;

   r = (color>>10) & 31;
   r = (r<<3) | (r>>2);
   g = (color>>5) & 31;
   g = (g<<3) | (g>>2);
   b = color & 31;
   b = (b<<3) | (b>>2);

   result = (r<<16) | (g<<8) | b;
   return( result );
}

//-------------------------------------------------
long Convert16To32bpp( short color )
{
   long result, r, g, b;

   r = (color>>11) & 31;
   r = (r<<3) | (r>>2);
   g = (color>>6) & 63;
   g = (g<<2) | (g>>4);
   b = color & 31;
   b = (b<<3) | (b>>2);

   result = (r<<16) | (g<<8) | b;
   return( result );
}

//-------------------------------------------------
long Convert24To32bpp( long color )
{
   return( color & 0x00FFFFFF ) ;
}
```

CONVERSION RGB<->BGR

Como ya se ha comentado, la conversión entre ambos tipos de formato equivale a cambiar el orden de los desplazamientos dentro del *dword* final. El listado 3 refleja las dos funciones de conversión.

LISTADO 3. CONVERSION RGB<->BGR

```
//-------------------------------------
long Convert32bppRGBToBGR( long color )
{
   long result, r, g, b;

   r = (color>>16) & 255;
   g = (color>>8)  & 255;
   b = color & 255;
   result = (b<<16) | (g<<8) | r;
}

//-------------------------------------
long Convert32bppRGBToBGR( long color )
{
   long result, r, g, b;

   r = color & 255;
   g = (color>>8)  & 255;
   b = (color>>16) & 255;
   result = (r<<16) | (g<<8) | b;

}
```

AVERIGUAR EL ESTADO DEL DISPLAY

Para averiguar el formato del buffer de vídeo, y convertir nuestras imágenes al mismo, podemos solicitar al sistema que nos indique las máscaras (patrones de bits) de las distintas componentes de color, además del estado del display (8/16/24/32bpp).
Para averiguar el número de bits por píxel en que se está trabajando bajo Windows (bajo DOS lo seleccionamos nosotros mismos) simplemente hemos de obtener la información del contexto de dispositivo mediante:

```
numbpp = GetDeviceCaps( hdc, BITSPIXEL );
```

Respecto al tipo de modo (555, 565, RGB, etc.), para Vesa/MSDOS, en el *ModeInfoBlock* (de cada modo de vídeo), disponemos de las siguientes variables para realizar la identificación del modo de display:

```
char RedMaskSize, RedFieldPosition;
(e igual con Green y Blue)
```

La primera indica el número de bits para esa componente (por ejemplo, *RedMaskSize* es 5 para el modo 5:5:5), y la segunda el lugar donde está situada esa componente (para el shift). Para *RedFieldPosition* en 5:6:5 obtenemos 11 para RGB y 0 para BGR. De esta manera, podemos crear fácilmente una sóla rutina de conversión con estos datos.
En el caso de DirectX, como resultado de solicitar una descripción de la *DDSURFACE*, obtendremos 4 máscaras de 32 bits (RedMask, GreenMask, etc), que nos permitirán identificar el modo de display activo.
Por ejemplo, para 565RGB se devuelve en la redmask el valor 0xf800. Este valor pasado a binario es 1111100000000000b que, como puede verse, corresponde a la posición de la componente RED en el color. Greenmask (también devuelta) tomaría el valor 0000011111100000b. De esta manera, podemos identificar el tipo de buffer de vídeo a partir de las posiciones (*xxSize*) y de los valores (*xxPosition*).
Con toda esta información, para convertir la imagen, por ejemplo, de 15bpp a 24bpp, podemos utilizar un bucle como el siguiente :

```
for( y=0; y<altura; y++ )
 for( x=0; x<anchura; x++ )
 {
  color = GetPixel(x, y, origen);
  color = 32To24bpp( 15To32bpp( color ) ) ;
  PutPixel(x, y, color, destino) ;
 }
```

CONVERSION ELEGANTE

Un método de conversión más elegante que el de disponer de tan elevado número de rutinas es aprovechar directamente los datos de las máscaras devueltos por el sistema y contar uno mismo (con un bucle de conteo de bits) las posiciones de los mismos, realizando una sóla rutina de conversión de imágenes a la que se le pasen como parámetros las máscaras origen, destino y el color a convertir, pero el método expuesto en este artículo ha de servir para lo que estaba pensado: mostrar de manera sencilla las conversiones. Para ello escaneamos desde el final (bit 0) hasta encontrar un bit distinto de 0, haciendo lo mismo desde el principio (último bit). La diferencia entre estos 2 valores más 1 (bit más alto - bit menos alto encontrados != 0 + 1) constituye el número de bits para esa componente.

Podemos identificar el tipo de buffer de vídeo a partir de las posiciones xxSize y de los valores xxPosition

Así, para un *redmask* de 0x7c00 (0111110000000000), encontramos que el bit menor a 1 es el número 10, y el más alto es el 14. Así, el número de bits para esta componente (xxSize) es 14-10+1=5, y la posición base (xxBase) para el desplazamiento << es el la del bit menor (10).
Con este sistema podemos, de una manera general, controlar cualquier tipo de organización de videomemoria, haciendo conversiones con estos datos. Al obtenerse RedSize y RedBase en DirectX nos encontramos en el mismo caso que en Vesa 2.0 (donde teníamos *RedMaskSize* y *RedFieldPosition*, con exactamente el mismo significado), de manera que podemos realizar el mismo tipo de conversión abstracta.

LOCK(), UNLOCK() Y DIRECTX

Las rutinas de dibujo que proporciona DirectX para el trazado (blt()) de gráficos resultan muy rápidas si existe aceleración hardware en el sistema. Si no es así, resulta muy sencillo escribir nuestras propias rutinas de volcado de bmps/sprs, tal y como hemos explicado hasta ahora. Para ello, al dibujar en una Surface (similar a un buffer lineal), tan solo tenemos que bloquearla (*lock()*) para escribir en ella como en cualquier otro buffer. Tras el trazado, la desbloqueamos (*UnLock()*) para devolver su control a DX.

Para averiguar el número de bits por píxel bajo Windows hemos de obtener la información del contexto de dispositivo

Al escribir en la dirección devuelta por el Lock() (**lpSurface*), no podemos utilizar el ancho lógico de pantalla, sino que hay que usar el valor especificado por DirectX como ancho de la Surface (ya que no es obligatoriamente el ancho que le pedimos):

```
// usar previamente Surface->lock()
PutPixelDX( x, y, color )
{
  char *Vram = (char *) surface->lpSurface ;
  dword pitch = surface->lPitch ;

  Vram[ (y*pitch)+x ] = color ;
}
```

Próxima entrega

Todo lo visto en esta entrega nos permite cerrar la explicación sobre trabajo en SVGA. Aprovechamos para agradecer a Russ Williams (r.g.p) su ayuda prestada en este tipo de conversiones.
En la próxima entrega se comentará la rutina de BestMatch() para cuantización de color, así como la aplicación de filtros a imágenes (Softens, Blurs...). La rutina de BestMatch servirá también para crear efectos de luces con LookUpTables bajo 8bpp.