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

PROGRAMACIÓN ACTUAL

AÑO 1 • NÚMERO 12

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

SÓLO 995 ptas

WindowsNT

EL PRESENTE Y EL FUTURO

SECCIÓN NIVEL BÁSICO

Control de la CPU en **ENSAMBLADOR, C y C++** en **PROGRAMACIÓN C++**, Hash en CURSO DE PROGRAMACIÓN

SECCIÓN BAJO NIVEL

Mapeado de texturas en **ENTORNOS 3D**, La **SVGA** en **PROGRAMACIÓN GRÁFICA**, El efecto rotozoom en **DEMOSCENE**

SECCIÓN LAS EMPRESAS DEMANDAN

Control de flujo en **VISUAL BASIC 5.0**, Activex en **VISUAL C++ 5.0**, Las DBtools.h en **BASES DE DATOS**, Marcas de agua en **COMERCIO ELECTRÓNICO**

SECCIÓN CAMPUS ACTUAL

El depurador **GDB** en **LINUX ACTUAL**, Multitarea en **PROGRAMACIÓN EN ADA**, **DHCP** en **REDES LOCALES**

JAVASCRIPT AVANZADO

APLICACIONES COMERCIALES Y GNU EN LINUX

EN EL CD ROM:

- Java Runtime Environment 1.1.5
- OpenGL API 1.1
- Utilidades para Internet
- Fuentes de los artículos
- Netscape Communicator y MS Explorer

Y UTILIDADES DE PROGRAMACIÓN:

ActiveContainer • Enigma10 OCX Gold1.0 • Animation GIF ActiveX 2.0 • AnetHelp Tool • Async Toolbox 3.0b • CSplitBarCtrl.OCX • CGI*StarPro • Dynamic SQL Debugger • Goetz's Scrolling Text 1.11 • MSStat Sample Control 1.0 • Printer 1.0a • SH-Register Tool Kit • SText.OCX • TeeChart Pro ActiveX 3.0 • Applet Menu Wizard 1.0 • CasualWriter 1.05 • CodeSMART 2.0. y muchas más.

Prensa Técnica

8 413042 087779 00012

Autor: Santiago Romero

Trabajando en SVGA

En esta entrega se pretende abordar el funcionamiento de los modos SVGA e introducir nuevos conceptos generales para una programación más estructurada y portable, así como comentar las técnicas actuales usadas para trabajar en los modos de alta resolución en juegos y aplicaciones gráficas.

Hasta ahora el curso ha tratado de proporcionar fundamentos básicos de la programación gráfica aplicados al modo 13h por ser sencillo de inicializar y programar. A partir de este momento, el curso sufre un pequeño giro y comienza a ahondar en conceptos arraigados en la programación actual de los programas con acceso masivo a gráficos, tales como los videojuegos, las demos o las aplicaciones de visualización y retoque.

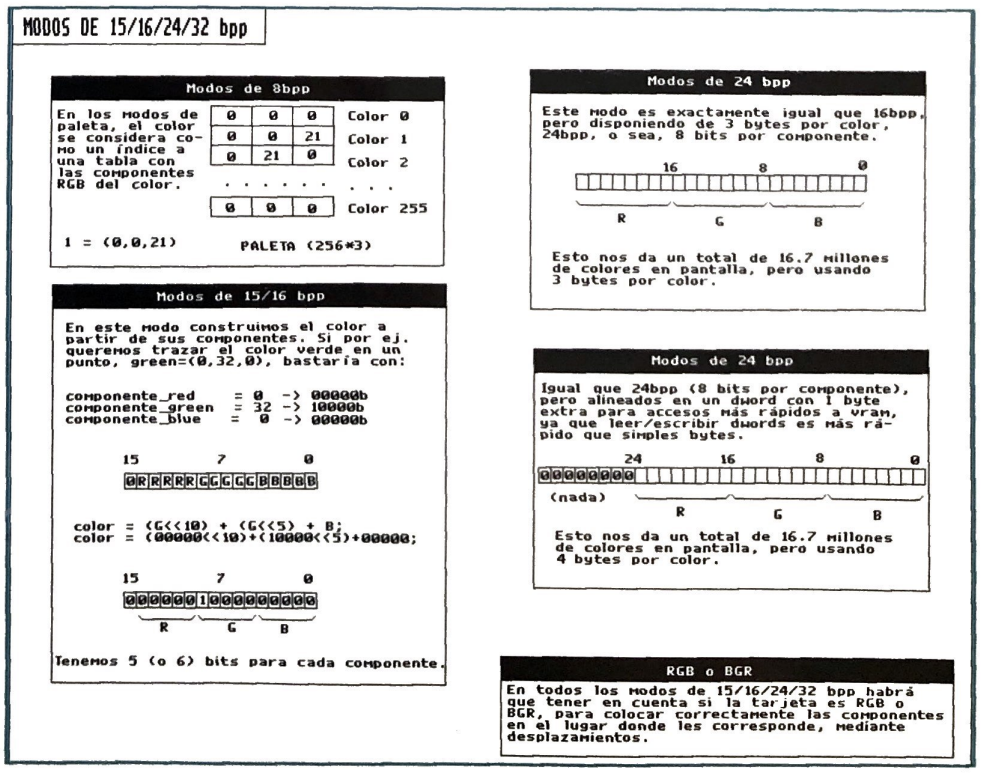
BASE DE LOS PROGRAMAS GRAFICOS

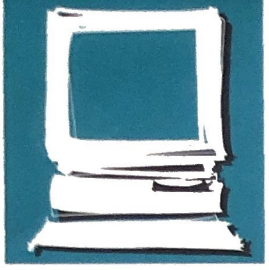
Lo primero que se verá es un pequeño conjunto de reglas que ayudarán a que los programas gráficos ganen en portabilidad, aunque sin perder la filosofía de velocidad que requieren éstos. A partir de ahora trataremos de trabajar en C (o en C++, cuando sea necesario), para adaptarnos a las técnicas actuales de programación gráfica. Esto se hace por diferentes motivos:

- *Compiladores/lenguajes:* Muchos tenemos la costumbre de asociar el ensamblador con velocidad. Hemos de tener muy en cuenta que los compiladores actuales (Watcom/VisualC/etc.) son capaces de traducir el código C a un ensamblador de una manera increíblemente eficiente, incluso mejor de lo que podría hacerlo un humano (a menos que éste conozca todas las interioridades de la máquina), ya que los compiladores conocen el funcionamiento interno de la CPU y tienden a escribir código que utilice las *pipelines* del Pentium para ahorrar muchos ciclos de reloj (el micro Pentium es capaz de ejecutar 2 instrucciones en 1 sólo ciclo de reloj si se agrupan de manera correcta; así, podemos obtener una rutina ejecutándose en, aproximadamente, la mitad de ciclos que una nuestra, gracias al compilador).
- *Legibilidad y portabilidad:* Un código en C/C++ es mucho más legible que un bloque ASM, haciéndolo compilable y portable a otras plataformas con ligeros cambios. Además, resulta más fácil encontrar y corregir bugs y fallos algorítmicos. Si trabajamos en C/C++, nuestro mismo programa puede ser compilado (con un código igualmente optimizado) en otras plataformas, aunque no sepamos cómo funcionan internamente éstas.
- *Tiempo:* En C/C++ tardaremos aproximadamente la mitad de tiempo en realizar el mismo programa. Eso significa más rendimiento y, por tanto, más beneficios (o, si no los hay, un ahorro de tiempo con vista hacia otro proyecto).
- *Algorítmia:* La clave en la programación actual no está en optimizar una y otra vez la misma rutina, sino en diseñar buenos algoritmos que, incluso con una mala implementación, deriven en mejores resultados que los obtenidos con peores maneras de abordar un determinado problema.
- *Versiones finales:* Si hemos finalizado un programa en un lenguaje de alto nivel y pensamos distribuirlo o hacerlo de dominio público o shareware, basta con coger cualquier *profiler* ("cronometrador" de tiempos de ejecución de las distintas rutinas de un programa), tal como el Vtune, el IPeak o incluso el Turbo Profiler, y ejecutarlo para analizar nuestro programa.

Esto permite darse cuenta de cuáles son las rutinas que más tiempo toman de un programa, y las que más se ejecutan, y escribir una versión en ensamblador de las mismas para la versión del programa de una máquina específica. De esta manera, ya que aproximadamente el 90% del tiempo de ejecución reside en el 10% del código, mediante el cambio de estas rutinas ganaremos entre un 10% y 30% de velocidad habiendo utilizado un mínimo tiempo en la creación del mismo.

FIGURA 1. DIFERENTES SOPORTES DE VIDEO.





Por supuesto, si se escribe un programa 100% en asm conociendo los diferentes trucos de *pipelinning* y saltos de cache, etc., se puede obtener un código mucho más rápido, pero en mucho más tiempo y más difícil de mantener.

ENCAPSULACION Y GRAFICOS

Esta palabra, bastante conocida por quienes programen en C++ (o, en general, en OOP), define la técnica de ocultar los datos y funciones asociadas a una determinada tarea con el fin de abstraer al programador de las funciones específicas asociadas a la misma. Veamos qué quiere decir esto con un ejemplo práctico: en este mismo artículo vamos a utilizar una serie de funciones para trabajar en SVGA. La encapsulación de estas funciones consiste en crear un set de funciones (o macros/inlines, ya sean globales o dentro de una clase) que realicen llamadas a la librería específica que usemos.
El objetivo es que dentro del código principal de nuestro programa no haya ninguna llamada a funciones específicas de vídeo. Si nuestra librería es de SVGA VESA 1.2 para MSDOS, no llamamos directamente a estas funciones, sino a nuestra clase/set de funciones que, a su vez, llaman a éstas.

Se pueden cambiar las llamadas en la librería encapsulada para utilizar cualquier otra librería que se desee

¿Qué se consigue con ello? Que el código no sea dependiente de una sola plataforma. Es decir, en el ejemplo anterior, si cambiamos nuestra librería por una de *DirectX*, no habremos de cambiar ni una sola línea de código de nuestro programa principal para que compile (bueno, tan solo las correspondientes al modo de funcionamiento de windows, como el bucle de mensajes, etc). Solamente la clase/set de funciones que hemos creado, que se dice que *encapsulan* las funciones de vídeo. Por ejemplo, crear una función *GFX_FlipScreen()* que contenga una llamada a la *FlipScreen()* de la librería nos libera de utilizar *FlipScreen()* dentro del programa principal.
De no hacerlo de esta manera, si hemos de portar el programa de DOS a WINDOWS, o a LINUX, será necesario ir línea por línea de nuestro programa cambiando las llamadas a la distintas funciones, cambiando los parámetros, etc. Mediante la encapsulación de las funciones de vídeo tan solo hay que modificar ese set de llamadas a otras funciones, proceso tan sencillo como editar el fichero que contiene las definiciones y cambiar *FlipScreen()* por *DDraw: :Blt()*, o similar.

Por eso es recomendable aprender C++ y algo de organización a la hora de programar, para ahorrar luego horas y horas de trabajo, y poder reutilizar parte del código en otro proyecto posterior, comenzando desde la mitad en vez de desde cero.
Los conceptos anteriores pueden parecer de poca utilidad pero, en realidad, nos permiten trabajar de forma mucho más profesional y concentrarnos en la lógica de nuestro programa. Lo importante de un programa no es la manera en que acceda a los gráficos (aunque lo sea en parte), sino la función que realiza el mismo. Si es un juego hemos de centrarnos en hacer el juego, no en las funciones gráficas: el usuario se divierte jugando al juego (o utilizando la aplicación), no en ver cómo se trazan los gráficos en pantalla.

SVGA Y VESA 2.0

Para trabajar en SVGA vamos a utilizar (para los ejemplos) el compilador WATCOM C/C++ (a partir de la versión 10.0a), encapsulando los datos a partir de nuestras próximas aplicaciones. Si queremos portar cualquiera de los programas que hagamos a Windows o Linux, bastará con modificarlo ligeramente para que utilice las *DirectX*, la SVGALIB, el GDI, o la VESA. El extensor de modo protegido es el PMODE/W de Tran.
Nosotros hemos decidido, como siempre, trabajar para los ejemplos bajo DOS usando, a partir de ahora, la VESA 2.0. Hay muchas librerías para trabajar con la VESA 2.0, incluida la que estáis utilizando en la excelente sección de *demoscene*, pero nos hemos decantado por una librería *learnware* que, además de venir en código fuente (no en .LIB, facilitándonos la modificación y actualización), y de ser compatible 1.2, lleva documentación explicativa, ejemplos y, sobre todo, detección y corrección de los errores de programación VESA bajo las tarjetas de S3 y MATROX. Por supuesto, se pueden cambiar las llamadas en la librería encapsulada para utilizar cualquier otra librería que se desee, pero nosotros vamos a manejar ésta. Para ello, vamos a ver un breve resumen de sus funciones, lo que hacen y un pequeño programa de ejemplo.

LIBRERIA DE SVGA

La librería a utilizar es la "*Submissives VESA VBE 2.0 Application Core*". El código fuente de la misma se encuentra en los ficheros VESAVBE.C y VESAVBE.H, que necesitareis linkar e incluir, respectivamente, en el programa para poder utilizar sus funciones. Una vez hecho esto, disponemos de un set de funciones de inicialización y diagnóstico variado.

Esta librería, además, incorpora un par de *#defines* de constantes para permitir o para no permitir (basta con comentar el *#define* entre /* y */) varias cosas: control del bug de la S3 y de las Matrox, posibilidad de enviar cadenas de debug a un fichero de texto para solucionar fallos en nuestro código, etc.

Una vez hecho esto, disponemos de un set de funciones de inicialización y diagnóstico variado

Pero comencemos viendo las principales funciones (para VESA 2.0):

VBE_Init(); debe ser llamada una sola vez al principio del programa y antes de llamar a cualquier otra función de la librería.

VBE_Done(); debe llamarse antes de salir del programa, y libera la memoria asignada (1k) para las estructuras de VESA 2.0.

VBE_Test(); devuelve 1 si la versión de VBEVESA es 2.0 o superior.

VBE_IsModeLinear(Modo); indica si el modo de vídeo especificado es lineal (LFB) o no (por bancos), devolviendo 0 si no lo es.

VBE_FindMode(xres, yres, colores); busca dentro de la tabla de modos disponibles el número de modo (un *short*, de 16 bits) que corresponde a la resolución especificada (ejemplo: (320,240,16)), ya que los números de los modos no son iguales en todas las tarjetas, se hace necesario guardar en una tabla los modos disponibles y buscar en ella el número de modo asociado al mismo.

VBE_GetVideoPtr(mode); esta función, muy importante, devuelve la dirección del LFB de manera que accediendo a ella (ya sea mediante punteros o como un array) podemos acceder a la videomemoria.

El resto de funciones disponibles están comentadas en el fichero VESAVBE.TXT, por ello no las comentaremos aquí. En el listado 1 puede verse un ejemplo de programa en SVGA utilizando esta librería. En él dibujamos un conjunto de rectas verticales, y su funcionamiento es idéntico que el visto hasta ahora para el modo 13h, de manera que, a partir de este momento, comenzaremos a agradecer la andadura que hicimos por el 13h al principio del curso.

WATCOM Y MODO PROTEGIDO

No vamos a explicar cómo se programa en modo protegido pero sí dar unas ligeras referencias bajo el mismo, orientadas a que los lectores no familiarizados con él puedan seguir los ejemplos proporcionados.
En primer lugar, no necesitamos saber cómo se entra y sale del *protected mode*, eso es tarea del

LISTADO 1. SVGA 640X480 CON VESAVBE.C.

```
/*_____________________________________
 Ejemplo de programa en SVGA.
 Se usa la librería VESAVBE.C.
 _____________________________________*/
#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include "vesavbe.h"
#define dword unsigned long

void SVGAPutPixel( dword, dword, char );
char *VIDEO;

/*_____________________________________*/
void main( void )
{
 short Mode, x, y;

 /* inicialización del modo SVGA */
  VBE_Init();
  Mode = VBE_FindMode(640,480,8);
  if ( Mode==-1 )
  {
   printf ("Modo no encontrado.\n");
   return;
  }

  if ( !VBE_IsModeLinear(Mode))
  {
   printf ("El modo actual no tiene LFB.\n");
   return;
  }

  VBE_SetMode (Mode, 1, 1);
  VIDEO = VBE_GetVideoPtr (Mode);

  /* trazamos líneas verticales */
  for( x=0, x<256 ; x++ )
   for( y=0, y<100 ; y++ )
    SVGAPutPixel( x, y, x );

  getch();

  VBE_SetMode (3,0,1);
  VBE_Done();

}

/*_____________________________________*/
void SVGAPutPixel( dword x, dword y, char color )
{
 if( x<640 && y<480 )
  VIDEO[ (y*640)+x ] = color;
}
```

extensor que se utilice (DOS4GW o PMODE, GO32, etc.). Tampoco es necesario que el código C sea distinto del de modo real, de su traducción se encarga el compilador.

Lo que sí es cierto es que al poseer registros de 32 bits (EDI/ESI/EBX) para acceso a memoria, nos olvidamos del tipo *far* de punteros (sólo necesitamos el *near*) y desaparece la segmentación que veíamos hasta ahora con segmento+offset. En vez de esto, hemos de acceder a direcciones absolutas (SEGMENTO*16+OFFSET), cabiendo todo en un solo registro (hasta 4 gigas de memoria lineal):

```
mov edi, 0xA0000
add edi, 330      // pixel (10,1) 320+10
mov al, [color]
mov [edi], al
```

Nótese que hemos usado 0xA0000 y no 0xA000 (un cero más), ya que hemos multiplicado la dirección del segmento por 16 (o añadir un cero en hexadecimal) para obtener la dirección del buffer de vídeo. De la misma manera, el trabajo dentro de C/C++ para acceso a memoria cambia ligeramente (ya no hay segmentación):

```
PutPixel( long x, long y, char color )
{
  pokeb( 0xA0000 + (320*y) + x, color )
}
```

Si no se dispone de solvencia económica para conseguir algún compilador (con su licencia, por supuesto), o las versiones de estudiante de las mismas (consultar precios), podemos optar por el compilador DJGPP, *port* del GCC de Linux al MSDOS. Este compilador genera código 32 bits en modo protegido y es totalmente gratuito, pudiéndose encontrar en Internet fácilmente (*www.delorie.com*), además de estar desarrollándose (si no ya funcionando) una versión para Windows95 del mismo compilador (o bajo la misma licencia GNU).

Aparte de la resolución deseada podemos optar por diferentes profundidades de color *(pitch)*

Así pues, es posible obtener este compilador, leer unas cuantas FAQ's o TXT's sobre el mismo (importante para comprender algunas particularidades de DJGPP), y trabajar con él. Sólo decir que genera buen código y que, a modo de ejemplo, Quake se desarrolló bajo DJGPP, lo cual dice mucho en su favor.

TRABAJANDO CON SVGA

Ya aclarada nuestra nueva manera de trabajar, volvamos la vista hacia la SVGA. Cuando nos dispongamos a utilizar un modo de alta resolución (o de baja, pero usando más de 256 colores), se abren ante nosotros una serie de posibilidades. Aparte de la resolución deseada (640x480, 800x600, 400x300, etc.) podemos optar por diferentes profundidades de color (*pitch*). Esta profundidad se mide en *bits por píxel* (*bpp*), e indica el número de bits necesario para representar un color. En 320x200 (13h) teníamos 256 colores, representables todos ellos mediante un byte (8 bits); es, pues, un modo de 8 bits por píxel.

Otras posibilidades son 15 bits por píxel (1 word, 16 bits, sobrando el bit 15), que da 2 elevado a 15, total 32.768 colores (*32K color*, o *15bpp*), 16 bits por píxel, o 65.536 colores (*64K color* o *high color*), y 24 bpp o 32 bpp, llamado también *truecolor* (16.4 millones de colores).

Nuestro objetivo no es comprender cómo trabajan las funciones VESA (ni de cualquier otro API), sino utilizarlas para representar en pantalla

nuestros algoritmos gráficos. De lo primero se encargan otras secciones de la revista (como la de Demoscene), aunque sí es cierto que habremos de hacer algunas puntualizaciones sobre las profundidades de color antes de continuar.
Tanto si se trabaja usando el estándar VESA como DIRECTX o cualquier otra API, si trabajamos en SVGA habremos de elegir el modo de vídeo y profundidad de color deseados para nuestra aplicación. Si, por ejemplo, elegimos 640x480x8 (8 = 8 bpp = 256 colores), necesitaremos inicializar el modo de vídeo al deseado y disponer de funciones que puedan trabajar con esa profundidad de color. Si, en cambio, elegimos 640x480x16 (16 bpp = 64K color), necesitaremos saber cómo trazar cada color, ya que ahora no disponemos de un sistema indexado por paleta (donde 1 valor = 1 color), sino que hemos de agrupar las componentes RGB del color deseado para poder representarlo en pantalla.

En este modo desaparece la paleta y, directamente, trazamos los colores por medio de sus componentes

A continuación, vamos a comentar los diferentes modos de vídeo según la profundidad de color y cómo trabajar con ellos. Si se siente tentado de saltarse la mayoría de ellos e ir directamente al que le interesa para crear, por ejemplo, un programa bajo Windows 95 en 312x234 (resolución perfectamente aceptable si usamos una ventana de gráficos (*windowed mode*) y no a pantalla completa (*fullscreen mode*), sepa que el usuario puede tener configurado el escritorio a cualquier modo de vídeo (16bpp, 8bpp, etc.). Igual ocurre, por supuesto, con resoluciones estándar, pero en ventana.

Modos de 32 bpp

Este modo es exactamente igual a 24bpp pero se utilizan 4 bytes (un dword, ignorando el byte más significativo) para almacenar valores en memoria. Esto redunda en mayor necesidad de memoria pero un incremento muy notable de velocidad respecto a otros modos, ya que resulta más rápido escribir 4 dword que 1 byte, 1 word o, especialmente 3 bytes. El formato de estos 4 bytes es NADA:R:G:B (0:R:G:B) o NADA:B:G:R (0:B:G:R), de modo que resulta necesario conocer también el tipo de tarjeta para conocer la organización interna del buffer de vídeo.

Esto quiere decir que cuando obtengamos, mediante *DirectDraw*, una *DDsurface* (superficie de *directdraw*), que es un buffer que representa la pantalla de vídeo (la ventana) linealmente (igual que 13h o VESA 2.0, pero un buffer del tamaño de la ventana que tenemos), ese buffer va a tener la misma profundidad de color que el display, y ésta va a depender, a su vez, del gusto del usuario: hay quien puede tenerlo a 640x480x32bpp, y si su rutina de dibujo de píxels, sprites, etc., no sabe dibujar en 32bpp (la resolución no es problema, porque para usted es siempre la de la ventana, pero la profundidad de color cambia), su programa no funcionará (otra solución es usar las funciones de *blitting* de las *Directx*, pero eso, como veremos posteriormente, sólo nos interesa en determinadas ocasiones).
Así que vamos a hacer un breve resumen de los diferentes modos, y en la próxima entrega los comentaremos a fondo. Para ello veamos cómo se organiza la memoria cuando tenemos en el sistema un *Linear Frame Buffer* (LFB), en vez de un sistema de bancos.

MODOS DE 8 BPP

Si el modo de vídeo que hemos elegido (o el presente en el display) es de 8 bpp, nada cambia en cuanto a la filosofía vista hasta ahora. En este modo, cada color es representado por un byte (8 bits), y ese valor representa un índice dentro de una tabla interna de RGB (la paleta), donde la tarjeta irá a buscar las componentes para trazar el color en pantalla.
Como siempre, el offset del píxel (con LFB) se calcula mediante la fórmula de siempre, debido a la linealidad de la videomemoria.

*offset = (y*ancho_pantalla)+x ;*

El ancho de pantalla no se refiere a la resolución sino al ancho en bytes de la videomemoria. Este dato es devuelto por las funciones de inicialización y, a veces, coincide con la resolución horizontal.

MODOS DE 15 BPP Y 16 BPP

Este modo de vídeo difiere mucho en comparación con el anterior. Un gráfico en este formato necesita 2 bytes por cada píxel, por lo que para cada *scanline* necesitaremos el doble de bytes que en 8bpp.
En este modo desaparece la paleta y, directamente, trazamos los colores por medio de sus componentes. Es decir, un número no representa un color concreto dentro de una tabla, sino que construimos el color usando las componentes que queremos que tenga.
Si tenemos 15bpp, tenemos 5 bytes para cada una de las 3 componentes RGB (*5:5:5*, 5x3=15), y el valor máximo de una componente será 2 elevado a 5 = 32. Si queremos escribir el color azul puro RGB=(0,0,32), habremos de componer en un word (16 bits) las 3 componentes mediante *shifts* u operaciones OR, y escribir este byte en videomemoria.

short color = (R<<10)+(G<<5)+(B) ;

Esta configuración se llama *5:5:5*, ya que se usan 5 bits para la R:G:B. Como puede verse, el último bit (MSB, o bit 15) no es usado, y esto es lo que le diferencia de los modos de 16bpp que sí que lo usan y dan por ello acceso a 32768 colores más (64K color). El número de colores en 15bpp es, pues, 32*32*32=32768.

En SVGA habremos de elegir el modo de vídeo y profundidad de color deseados para nuestra aplicación

Pero en este modo hay un ligero problema, ya que algunas tarjetas usan 5:6:5 para RGB, o incluso usan un sistema distinto, BGR, donde las componentes se deben almacenar en el word en sentido inverso.
El modo 5:6:5 se controla cambiando los valores de los desplazamientos (10,15,0) para colocar cada componente en su lugar correcto dentro del word que representa el píxel. Para solucionar el almacenamiento en RGB o BGR tan solo hay que modificar el orden de las componentes en los shifts (<<) para obtener el color correcto.
Por esto no debe haber motivo de preocupación, ya que la información devuelta por las *DirectX* o la inicialización de un modo VESA nos permite saber las máscaras (5:5:5 o 5:6:5) y la organización de las componentes, de manera que compondremos los colores mediante sencillos desplazamientos y operaciones lógicas.
En estos modos se puede modificar un color (un píxel) por otro de cualquier tonalidad sin cambiar la paleta, simplemente construyendo uno nuevo a partir de sus componentes RGB, cosa que lo hace ideal para efectos de luz (sumar valores constantes a todos los píxeles), como los flares (el famoso sprite de luz). Al no haber paleta, la realización, por ejemplo, de un fundido ya no es tal y como veníamos haciendo hasta ahora, sino que tendremos que, para todos los píxeles de la pantalla, decrementar sus componentes RGB en una constante (por ejemplo, 1), disminuyendo así la intensidad de la imagen, es decir, nada nos impide ahora hacer un fundido de un solo

fragmento de la pantalla, pues la modificación de un píxel no implica modificar el resto de píxeles del mismo color.

MODOS DE 24 BPP

Este tipo de modos tiene 24 bits por píxel, lo cual significa 8 bits por componente, o 256*256*256 colores=16.777.216 colores en pantalla, construidos a partir de sus componentes mediante desplazamientos u operaciones lógicas. Este modo (*truecolor*) requiere 3 bytes por color, de manera que un sprite de 80x50 ocuparía en memoria 80x50x3 bytes = 12.000 bytes.

Windows 95 y las DirectX constituyen el soporte estándar del mercado actual

El principal problema de este modo de vídeo es (aparte del tamaño de sus gráficos y de que necesitamos gran cantidad de memoria de vídeo para los modos de alta resolución, hasta 2'5 MB de VideoRAM para 1024x768x24) que al ser 24 bits por píxel, no está alineado en videomemoria y no podemos escribir/leer *dwords* completos con tanta velocidad como en otros modos.

OTROS MODOS

Hay tarjetas que almacenan la información del buffer de vídeo en CYMK, lo que nos obligaría a realizar otra conversión o carga desde disco de los gráficos en este formato. No obstante, la cantidad de tarjetas con este sistema es ínfima y, por el momento, vamos a dedicarnos al otro 99% del mercado: el de RGB.

RUTINAS DE BLITTING

Probablemente nuestro deseo sea trabajar bajo Windows 95 y mediante las *DirectX*, ya que constituye el soporte estándar del mercado actual, o en lenguaje sencillo, "es lo que vende", que, por otra parte, es otra de las máximas del mundo de la programación profesional.
Resultaría muy sencillo inicializar un modo de vídeo (ya sea bajo una ventana o a pantalla completa) y trabajar tan solo con las funciones de dibujo de sprites y volcado de buffers de las *DirectX* (más concretamente de DirectDraw), como *Blt()* o *FastBlt()*. A estas funciones se le especifica el sprite/buffer a volcar en coordenadas de píxeles (no en offsets), y automáticamente la rutina se salta los bits por píxeles necesarios para dibujar correctamente el sprite, es decir, la misma rutina sirve para todos los diferentes modos de profundidad de color.
Esto se cumple, por ejemplo, para el caso de volcar un sprite, si el bitmap está almacenado en memoria con la misma profundidad de color que el buffer de pantalla. Es decir, si el display está en 16bpp, el sprite debe estar almacenado en 16bpp en memoria, y no en 8 bpp ni 24 bpp. Si trabajamos en *fullscreen* no hay problema: inicializamos el modo que deseemos y cargamos desde disco a memoria los gráficos (ya preparados en ese formato), para su uso con Blt(). Pero si trabajamos en *windowed* habremos de detectar el modo de profundidad de color actual y cargar en memoria los gráficos en ese modo para poder usar Blt().
Para ello tenemos dos posibilidades: la primera consiste en tener en disco 5 versiones de los gráficos del programa/juego, cada una en un diferente formato (8bpp, 15bpp, 16bpp, 24bpp, etc.), y cargar la adecuada. Eso es un despilfarro en términos de espacio en disco, por lo que a menos que el programa sea pequeño, muy organizado, y vaya en CDROM's o DVD's no es recomendable este método.

Conociendo la organización interna de estos modos de vídeo podremos trabajar con los gráficos en memoria

La segunda posibilidad es mucho más elegante y ahorrativa: consiste en almacenar en disco los gráficos en 24 (o 32) bpp (la máxima profundidad y calidad de gráficos), y al cargarse el programa, detectar la profundidad de color (*GetPixelFormat()*) y convertir las imágenes de disco de 24bpp a memoria al formato correcto. Esto es muy sencillo, como veremos el mes que viene, y nos permite trabajar con las funciones propias de *DDraw*.
El otro motivo por el que es recomendable saber trabajar en diferentes resoluciones se basa en la aceleración hardware. Si disponemos de ella en el sistema es muy recomendable usar un API como *DirectX* o VESA/AF (*accelerated functions*) porque mediante la aceleración hardware podemos obtener *framerates* (número de fotogramas por segundo) muy elevados. Pero, por otra parte, si la aceleración hardware no está disponible (cualquier tarjeta normal), situación detectable con la información devuelta por la propia API, resulta muy sencillo escribir funciones más rápidas que las propias de *DirectX*. Para ello, podemos efectuar (al instalar el programa en una determinada máquina, o en una página virtual antes de pasar a la ejecución del mismo) un pequeño test para detectar si es más rápida nuestra rutina que la propia de DX en esa determinada máquina (volcar un número determinado de buffers y contar tiempos, etc,).

Mediante la aceleración hardware podemos obtener *framerates* muy elevados

Éstas son las técnicas más comunes (y sencillas) dentro del mundo de las aplicaciones gráficas, que hemos esbozado aquí para que el lector las aplique a sus proyectos particulares. Ha sido un texto fundamentalmente teórico, pero de gran aplicación práctica.

TRABAJO CON GRAFICOS

Ahora que ya hemos hecho la diferenciación entre los distintos modos de vídeo podemos almacenar en memoria todo tipo de gráficos, procesarlos (filtros, transformaciones, etc.) y mediante, las rutinas que nos proporcione el sistema o una API externa (VESA, DX, etc.), trazarlos en pantalla.
Conociendo la organización interna de estos modos de vídeo podremos trabajar con los gráficos en memoria que, al fin y al cabo, es lo que nos interesa por cuestiones de velocidad: usar pantallas virtuales y sólo requerir del API para volcar esto a la página/pantalla visible, pero de los métodos de trazado en lo diferentes modos nos ocuparemos el mes que viene.

La próxima entrega

En el próximo número plantearemos el trazado y conversión de gráficos entre los diferentes soportes de vídeo, así como algunos temas planteados por los lectores: filtros a las imágenes (como los realizados por los programas de dibujo profesionales como Photoshop), programación gráfica estándar aplicada a los juegos actuales, etc.
Hasta entonces, es recomendable que el lector practique trabajando un poco en SVGA en los diferentes modos (tratando de crear algún pequeño programa o efecto), y examinar los ejemplos disponibles en el CD que acompaña a la revista.