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

PROGRAMACIÓN ACTUAL

AÑO 1 • NÚMERO 6

ARGENTINA 10$ • CHILE 3000$ • PORTUGAL 1500$

EUSKAL PARTY 5

LA FIESTA DE LA PROGRAMACIÓN

SÓLO 995 ptas

SECCIÓN NIVEL BÁSICO
Gestión de memoria en **CURSO DE C**, operaciones sobre bits en **ENSAMBLADOR**, virus informáticos en **ANALISIS ACTUAL**

SECCIÓN LAS EMPRESAS DEMANDAN
Controles estándar en **VISUAL BASIC 5.0**, funciones de columnas en **BASES DE DATOS**, curso de **VISUAL C++ 5.0**

SECCIÓN BAJO NIVEL
Bitmaps y sprites en **PROGRAMACIÓN GRÁFICA**, polígnos posteriores en **ENTORNOS 3D**, bios **VESA 2.0** en **DEMOSCENE**

SECCIÓN CAMPUS
Concentradores **ATM** en **REDES LOCALES**, tipos y punteros en **ADA**, el sistema de ventanas XWindow en **LINUX**

PROGRAMACIÓN PRÁCTICA: INSTALACIÓN DE SERVIDORES

EN NUESTRO CD ROM

TCL/TK 8.0 BETA 2
MAPLE V SECURE SHELL PARA UNIX

MANUALES DE PROGRAMACIÓN MICROSOFT

Ensamblador Nasm para Linux, Untgz para MS-DOS, Emacs 19.34 para Windows 95 y mucho más...

LA COMPARATIVA FINAL: JAVAOS Vs. INFERNO

Prensa Técnica

8 413042 087779 00006

Autor: Miguel Cubas y Santiago Romero

Los fuentes de letra

Este mes retomamos el tema de las fuentes de letras en el modo gráfico para luego continuar con el tratamiento de sprites y la utilización de pantallas virtuales para conseguir un mayor rendimiento en nuestros programas y aplicaciones. Mientras tanto, los ejemplos que acompañan al artículo y algunas utilidades nos ayudarán a dominar el dibujo de fuentes en modos gráficos.

Las fuentes del MS-DOS son un ejemplo de como el Sistema Operativo utiliza fuentes monocolor, que con un tamaño de 8x16 (4096 bytes) están guardadas en memoria (en la ROM) y es de donde la BIOS accede para representar en pantalla (en esas celdillas que la forman) los distintos caracteres. Estas letras pueden ser cambiadas mediante una subfunción de la INT 10h, destinada a controlar los modos de funcionamiento de pantalla y escritura de caracteres gráficos.

LAS FUENTES DEL DOS

Mediante la subfunción 1100h de la INT 10h es posible cambiar la dirección del nuevo buffer de caracteres de 8x16. De esta forma, el MS-DOS puede adquirir una nueva imagen distinta de la habitual (como realiza el programa FDOS incluido en el directorio UTILS).

LISTADO 2.A.
DIBUJO DE CARACTERES OPTIMIZADO CON ASM

```
/*-----------------------------------------
-
WriteChar(); Dibujo de
caracteres en ASM.

Pseudocódigo:
 DS:SI = OFFSET DEL CARACTER
 ES:DI = OFFSET DE LA
MEMORIA DESTINO
 ES:DI = (y*320) + x
  BUCLE1 FOR 0 TO ALTO
   BUCLE2  FOR 0 TO ANCHO
    ES:[DI] = DS:[SI]
    DI++; SI++
   END BUCLE2
   DI += 320-ANCHO
  END BUCLE1

-----------------------------------------
*/

void WriteChar ( int x, int y, char
ncar, char masc, int ancho, int
alto, char far *fuente, int
segmento)
{
int offset_car;

offset_car = ncar * (ancho*alto);
asm {
   lds si , [fuente]   /*DS:SI =
fuente*/
   add si , [offset_car]
   mov ax , [segmento]
   mov es , ax
   mov ax , [y]
   mov bx , ax
   shl ax , 8
   shl bx , 6
   add ax , bx
   add ax , [x]
   mov di , ax

   mov bx , [alto]
}
bucleY:;
asm mov cx, [ancho]
bucleX:;
asm {
   mov al , ds:[si]
   cmp al , [masc]
   je NoPinta
   mov es:[di] , al
  }
NoPinta:
asm {
   inc di
   inc si
   dec cx
   jnz bucleX

   sub di , cx
   add di , 320
   dec bx
   jnz bucleY
  }
}
```

IMPRESIÓN DE UNA FRASE COMPLETA

Escribir una frase consiste, simplemente, en escribir todos los caracteres que la componen. El mecanismo es muy simple, basta con imprimir todos los caracteres del array o frase a imprimir desde el número 0 hasta que se encuentre el indicador de fin de cadena que es el carácter 0 (o '\0'). Un ejemplo de esto podemos verlo en los ejemplos y librerías de código del CD que acompaña a la revista.

ORDEN DE ALMACENAJE DE LAS FUENTES

Hasta ahora hemos estado guardando las fuentes en el orden lógico en el que las visualizamos en pantalla (es decir, por scanlines horizontales). Por esto mismo, si es necesario en alguna aplicación dibujar las letras por líneas verticales en lugar de horizontales será más cómodo guardarlas en el orden en el que vayamos a sacarlas.

La figura 2 ilustra como es guardado el mismo carácter de una forma y de otra. De modo que si, por ejemplo, deseamos realizar un scroll de texto en modo gráfico, donde los caracteres tienen que salir por el borde de la pantalla visualizando líneas verticales, será más cómodo y rápido el guardar las fuentes en el orden en el que salen.

Para demostrar las posibilidades de las fuentes de texto y como ejemplo de almacenaje de las fuentes en sentido vertical disponemos de dos ejemplos en el CD que acompaña a la revista que realizan un scroll horizontal por software, tan utilizado en el mundo de las demos.

BITMAPS, SPRITES Y PANTALLAS VIRTUALES

Si los pixels son la primitiva básica de los gráficos por ordenador, la asociación de estos proporciona otra unidad mínima de las composiciones gráficas. Esta asociación constituye el mapa de bits o bitmap, donde se agrupan estos pixels para formar una figura gráfica mayor, un conjunto de pixels gráficos.

En principio es normal confundir los sprites con los bitmaps, pero no son exactamente lo mismo. Un sprite (duende, en castellano) es un bitmap con una característica que lo identifica: el movimiento. Para iniciar la comprensión de estas estructuras lo primero es ver algunas definiciones generales básicas.

BITMAPS Y SPRITES

Un bitmap (mapa de bits, en castellano) es la representación de una imagen o bloque gráfico dentro de la videomemoria de la tarjeta gráfica (o, en general, dentro de la memoria). La imagen visual que se suele tener al iniciarse en este tema es la de ver el bitmap como un "bloque cuadrado" que contiene todos los puntos que definen una imagen con una anchura y altura determinadas, tal y como puede apreciarse en la figura 1.

Ese pequeño recuadro (con la anchura y altura que sea necesaria para contener la imagen completa) que encierra la figura está constituido por ScanLines o líneas de escaneado, es decir, está constituido por una serie de líneas horizontales (o verticales, preferentemente en modos unchained o modos x) que toman forma a la hora de dibujar el bitmap, como veremos

LISTADO 3.A. DIBUJO DE CARACTERES MONOCOLOR EN C.

```
/* ----------------------------------------
   PutChar8x16( ); Dibujo de letras monocolor.
---------------------------------------- */
void PutChar8x16 (int x,int y,unsigned char color,unsigned char
caracter,char *fuente)
{
int offset_inicio, by;
unsigned char getbyte;

offset_inicio = caracter * 16;

 for (by=0; by<16; by++ , offset_inicio++)
  {
   getbyte = fuente[offset_inicio];
   if((getbyte & 128)!= 0) PutPixel( x , y + by, color);
   if((getbyte &  64)!= 0) PutPixel( x+1 , y + by, color);
   if((getbyte &  32)!= 0) PutPixel( x+2 , y + by, color);
   if((getbyte &  16)!= 0) PutPixel( x+3 , y + by, color);
   if((getbyte &   8)!= 0) PutPixel( x+4 , y + by, color);
   if((getbyte &   4)!= 0) PutPixel( x+5 , y + by, color);
   if((getbyte &   2)!= 0) PutPixel( x+6 , y + by, color);
   if((getbyte &   1)!= 0) PutPixel( x+7 , y + by, color);
   }
}
```

LISTADO 4.A. DIBUJO DE CARACTERES MONOCOLOR CON TEXTURA

```
/* ----------------------------------------
---
PutChar_MonoTextura8x16( );
Dibujo de letras monocolor con
textura
----------------------------------------
-- */
void PutChar_MonoTextura8x16 (int
x,int y,unsigned char caracter,char
*fuente, char *textura )
{
int offset_inicio, by;
unsigned char getbyte;
char offset_textura;

offset_inicio = caracter * 16;
offset_textura = 0;

for (by=0; by<16; by++,
offset_inicio++)
 {
  getbyte = fuente[offset_inicio];
  if((getbyte & 128)!=0) PutPixel( x,
y+by, textura[offset_textura]);
  offset_textura++;
  if((getbyte & 64)!=0) PutPixel( x+1,
y+by, textura[offset_textura]);
  offset_textura++;
  if((getbyte & 32)!=0) PutPixel( x+2,
y+by, textura[offset_textura]);
  offset_textura++;
  if((getbyte & 16)!=0) PutPixel( x+3,
y+by, textura[offset_textura]);
  offset_textura++;
  if((getbyte & 8)!=0) PutPixel( x+4,
y+by, textura[offset_textura]);
  offset_textura++;
  if((getbyte & 4)!=0) PutPixel( x+5,
y+by, textura[offset_textura]);
  offset_textura++;
  if((getbyte & 2)!=0) PutPixel( x+6,
y+by, textura[offset_textura]);
  offset_textura++;
  if((getbyte & 1)!=0) PutPixel( x+7,
y+by, textura[offset_textura]);
  offset_textura++;
 }
}
```

más adelante.
Un sprite, por contra, es un bitmap dotado de estado, posición, movimiento, etc., que necesita ser controlado, es decir, es un mapa de bits con características propias, como puede verse también en la figura 1.
Habrá que decidir qué velocidad tiene cada sprite, su posición inicial, qué tipo de eventos hacen modificar su estado, etc. Son, por lo tanto, bitmaps con una serie de variables que determinan su existencia, como pudiera ser el siguiente ejemplo de definición de Sprite:

```
struct Sprite
{
 char estado;
 int x, y;
 int ancho, alto;
 long energia;
 char *bitmap;
};
```

Su anchura y altura son el número de pixels de ancho y alto del bitmap que lleva asociado. Las variables (x,y), en este ejemplo, designarían su posición en pantalla, sus coordenadas. La variable energia se ha definido para dotarle de una característica distintiva de otros sprites (es tan sólo a modo de ejemplo), y estado es un flag o indicador cuyo valor puede usarse en el código para indicar el estado del sprite, como por ejemplo 0=inactivo, 1=activo, 2=móvil, etc.

Los sprites pueden tener cualquier tamaño, ya que nada nos impide utilizar sprites más grandes que la pantalla en sí, pero debe quedar claro que a mayor tamaño, mayor número de pixels y, por tanto, menor velocidad de representación.
De una manera más intuitiva, un bitmap es cualquier gráfico fijo de una aplicación o juego que no permite interacción con otros elementos del programa ni con el usuario, mientras que un sprite es un botón (que puede ser pulsado, estar activo, inactivo, etc.) o el personaje de cualquier juego (con sus coordenadas propias y sus interacciones con el usuario y el resto de sprites del juego).
Lo que vamos a ver en el presente artículo es la manera de guardarlos en memoria, de imprimirlos en pantalla, de gestionar su estado y movimiento y la manera de almacenarlos y cargarlos desde archivos de disco, además de la utilización de pantallas virtuales.

LISTADO 5.A. IMPRESION DE TEXTO CON SOMBRA

```
/*----------------------------------------------------------
   Print8x16( ); Imprime una cadena de texto con sombra
---------------------------------------------------------- */

void Print8x16 (int x ,int y ,unsigned char color ,unsigned char colsombra ,char *frase,char
*fuente )
{
int loop = 0;

while (frase[loop] != 0)
 {
  PutChar8x16 ( x+1 , y+1 , colsombra , frase[loop] , fuente );
  PutChar8x16 ( x , y , color , frase[loop] , fuente );

  x = x + 8 + 1;        /* separación entre carácteres */
  loop++;
 }
}
```

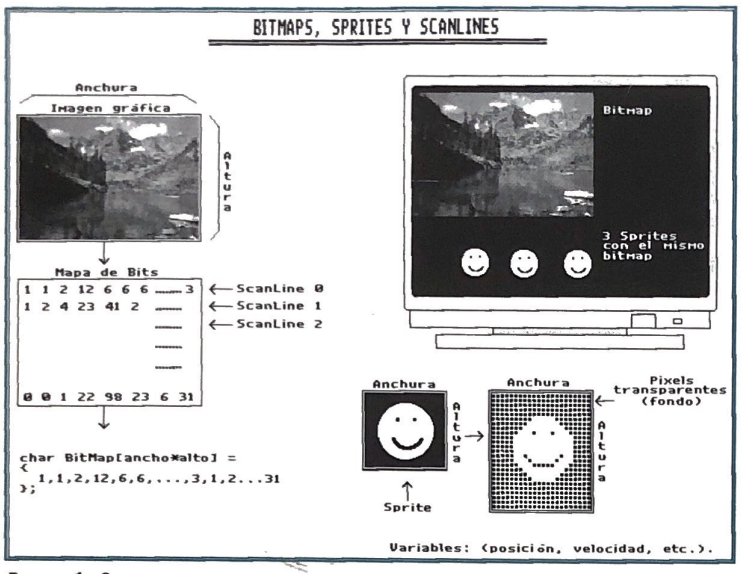


FIGURA 1. SPRITES, BITMAPS Y SCANLINES.

DEFINICION DE UN BITMAP

Al igual que hacíamos con las fuentes, podemos guardar los sprites en memoria en arrays conteniendo los diferentes puntos. Si estamos en un modo de 2 colores, nos basta con utilizar un bit por cada pixel (como en las fuentes monocolor), indicando 1 color (bit=0) o el otro (bit=1). El caso más sencillo son los modos de 1 byte por pixel (como el modo 13h), donde al representarse cada pixel con un byte, para almacenar un sprite de 16*16 pixels, por ejemplo, bastaría con un array tipo char de 16*16=256 elementos (similar a como hacíamos en las fuentes multicolor). Únicamente hemos de guardar en este array todos los puntos que lo definen, de manera consecutiva.

```
char Sprite[5*4] =
{
    3,1,1,1,3,
    2,0,0,0,2,
    2,0,0,0,2,
    3,1,1,1,3
};
```

Dado que los sprites son, simplemente, variables de estado, posición, etc., con un bitmap asociado es el dibujo de estos mapas de bits lo que nos interesa en primera instancia.

DIBUJO DE BITMAPS

En sistemas como el Amiga, por ejemplo, el hardware proporciona funciones para la impresión de mapas de bits con un tamaño prefijado, pero (mala suerte la nuestra) el PC se hizo, exclusivamente, "para trabajar" así que hemos de crear nuestras propias rutinas de impresión de bitmaps por software. Una forma de hacer esto consiste en utilizar la primitiva gráfica <pixel> para, utilizando bucles anidados, dibujar cada uno de los pixels que conforman el bloque gráfico.

```
void DrawSprite( int x, int y, int ancho, int alto )
{
int vx, vy;

 for( vy=0; vy<alto; vy++ )
  for( vx=0; vx<ancho; vx++ )
   PutPixel(x+vx, y+vy, Sprite[(vy*ancho) + vx] );
}
```

Esta forma de dibujar un mapa de bits es muy general y sirve para cualquier modo gráfico del que dispongamos de su función PutPixel().

DIBUJO POR SCANLINES

Dibujar los mapas de bits por ScanLines consiste en aprovecharse de la linealidad de modos como el 13h o los modos SVGA VBE 2.0 para dibujar los sprites como un conjunto de líneas horizontales. Simplemente nos colocamos en la posición correcta de pantalla (seg:offs) y comenzamos a dibujar pixels horizontalmente, que es como los tenemos organizados en el array y como mejor podemos traspasarlos a la VideoMemoria. Para pasar al siguiente punto de pantalla bastaría con incrementar el puntero de memoria (generalmente ES:DI). Al llegar al final de cada ScanLine incrementamos el puntero de memoria en <AnchoPantalla> unidades (para pasar a la línea siguiente) y le restamos el ancho del sprite para posicionarnos en el pixel correcto (para colocar el puntero de escritura en el primer pixel de la siguiente línea). Este proceso queda reflejado en el listado 1, con el siguiente pseudocódigo:

```
DS:SI = Sprite
ES:DI = (x,y) de pantalla
Repetir ALTO veces
  Copiar ANCHO bytes desde DS:SI a ES:DI
  DI += (AnchoPantalla-ANCHO)
Fin Repetir
```

Es, con este sistema, con el que se consigue una mayor velocidad de representación, ya que el bucle se repite sólo ALTO veces (y no ANCHO*ALTO como en el otro método), y además nos permite aprovechar la linealidad de la VRAM.

LOS PIXELS TRANSPARENTES

Al definir los bitmaps como bloques rectangulares y dibujarlos en pantalla tratándolos como sprites nos encontramos con el problema de que el relleno rectangular (el clásico fondo negro de los bitmaps) es también dibujado sobre la pantalla, sobreescribiendo el fondo. Esto significa que si dibujamos un sprite sin tener en cuenta qué pixels son transparentes (no deben ser dibujados) y cuáles no, en pantalla aparecerá el sprite con su recuadro negro en vez de ver el sprite recortado sobre el fondo (figura 2).
Hay 2 formas de evitar esto. La primera de ellas es la más rápida y cómoda en términos de memoria y es, por ello, la más sencilla de aplicar.

1) Utilizar un color máscara. Esto significa designar un color que no se use en el sprite en sí y con el que rellenamos las partes del bitmap que no han de dibujarse en pantalla, el fondo del bitmap. En la rutina de dibujado comprobamos el color de cada pixel del bitmap. Sólo en caso de que el pixel leído sea distinto del color máscara, lo dibujamos en pantalla. El color más utilizado en estos casos es el color 0 (generalmente negro), con el que se rellenan las partes del sprite que deseamos que sean transparentes. Dentro del bucle de dibujado con una simple condición podemos saber si el pixel que estamos tratando debe ser dibujado o no:

```
for( vy=0; vy<alto; vy++ )
```

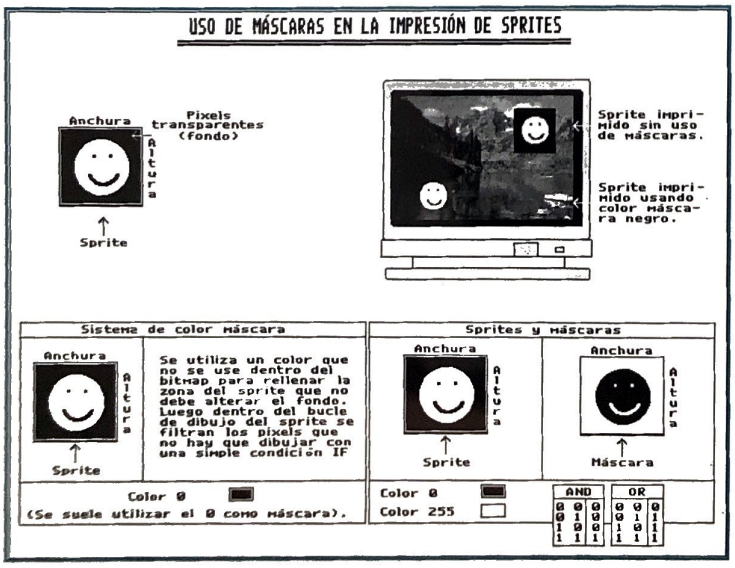
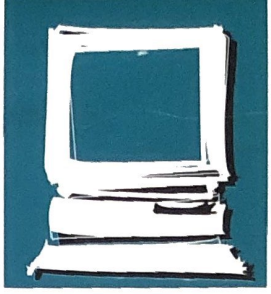


FIGURA 2. CONTROL DE LOS PIXELS DE FONDO.



```
for( vx=0; vx<ancho; vx++ )
 if( Sprite[(vy*ancho) + vx] != 0 )
  PutPixel(x+vx, y+vy, Sprite[(vy*ancho) + vx] );
```

2) Utilizar bitmaps y máscaras. Consiste en utilizar un bitmap invertido o máscara (como puede verse en la figura 2) con lo cual, mediante operaciones lógicas OR y AND, puede evitarse la escritura de los pixels transparentes sobre la pantalla.
Para crear las máscaras y los sprites hemos de rellenar la parte transparente del sprite (el fondo) del color negro en cualquier programa de dibujo. A continuación, copiamos el sprite al lado del anterior y rellenamos su fondo con el color 255. Estamos creando la máscara del sprite; tan sólo falta vaciar el interior de la máscara (lo que queremos que aparezca en pantalla) con el color negro. En resumen, necesitamos el dibujo original con fondo negro (0) y el mismo dibujo vaciado de negro con fondo blanco (255).
El proceso consiste en tener para cada bitmap una máscara de manera que podamos realizar los siguientes pasos (para cada pixel del sprite):

- Se lee el pixel sobre el que vamos a dibujar en pantalla.
- Se realiza un AND lógico (operador & en C y and en assembler) entre el byte leído y la máscara del dibujo.
- Con el byte resultante se hace un OR lógico (operador |) con el byte correspondiente al bitmap original.
- El resultado de la última operación se almacena en VideoMemoria.

Con un par de pruebas (de operaciones lógicas AND y OR), y observando la figura 2, el uso de máscaras resulta muy sencillo pero costoso en términos de memoria, pues para cada bitmap hemos de almacenar además su máscara. Por ello resulta mucho más cómodo, rápido y sencillo el primer método, realizable con una simple comprobación.

AHORRO DE MEMORIA

Cada Sprite, en realidad, no es más que un conjunto de variables de estado con un bitmap asociado. Esto quiere decir que con un sólo bitmap podemos crear multitud de sprites, no siendo necesario almacenar el cuerpo gráfico o imagen para cada uno de ellos. Esto puede verse en uno de los ejemplos incluidos en el CD que acompaña a la revista, donde se utiliza el mismo bitmap para 10 sprites diferentes que rebotan por la pantalla.
Los sprites pueden agruparse para dar lugar a un sprite mayor, algo que suele hacerse para reutilizar bitmaps. Si al crear un juego disponemos de un sprite cuyo mapa de bits sea el de una pequeña nave, por ejemplo, añadiéndole a los lados de esta 2 pequeños sprites que representen 2 escudos tendremos un sprite mayor y distinto del anterior, que nos ahorra el almacenamiento en memoria de esta mayor estructura.

CONTROL DE LOS SPRITES

Una vez declaradas las variables para cada sprite, el siguiente paso consiste en dotarles de un valor inicial y preparar la interacción que este sprite va a tener con el usuario y con el resto de sprites del programa.
En el caso, por ejemplo, del sprite de un juego habrá que detectar colisiones con otros sprites (cuando cualquiera de los puntos del sprite toma contacto con un punto de otro sprite), de controlar su movimiento con el joystick, ratón o teclado, o si no es controlable por el usuario deberemos tener precalculada una tabla de movimientos de manera que el sprite posea trayectoria propia.
Hay muchos ejemplos de lo que se acaba de nombrar. Si el usuario pulsa, por ejemplo, la tecla <DERECHA>, tras algunas comprobaciones habrá que incrementar la coordenada X del sprite en tantas unidades como velocidad tenga en esa dirección. Si colisiona con otro sprite, habrá que tener una función capaz de discernir qué ha de ocurrir tras ese choque (el personaje muere, recogemos un objeto del juego, el cursor está sobre el botón, etc).
Las tablas de movimientos, por otra parte, son arrays con incrementos del tipo (+1, 0, -1) que se añaden a las coordenadas (x,y) del sprite y están precalculados de manera que los sprites sigan una trayectoria determinada. Un ejemplo de esto son las trayectorias de los enemigos en un juego de naves (que suelen realizar complejas curvas u oscilaciones), y que permite a la 2ª ó 3ª partida saber por qué lado van a aparecer los enemigos y disparar allí antes de que entren en pantalla. Se puede hacer que sigan trayectorias circulares, lineales, curvas, etc.
Por otra parte, una vez almacenados en memoria los bitmaps, al representarlos en pantalla se les puede aplicar diferentes tratamientos tales como el escalado (ampliación o reducción de los sprites a un tamaño determinado, tal y como se hace en las aventuras gráficas cuando el personaje se aleja de la pantalla, reduciéndose), la rotación, la inversión, la impresión basándose en tablas sinusoidales, etc. Los tratamientos más sencillos son, por ejemplo, el borrado (escribiendo el fondo encima o un bloque vacío si no hay fondo) y el movimiento (borrándolo y dibujándolo en otra posición de forma continua).

COMO GUARDAR LOS SPRITES

Los sprites se guardan en disco de manera similar a como se realizó con las fuentes de texto. Se puede abrir un fichero, guardar allí todas las variables que lo definen (generalmente, la anchura, la altura y la paleta) y guardar, después, el cuerpo gráfico (el bitmap) que lo representa. La lectura es igual de sencilla.
En ocasiones interesará guardar muchos sprites dentro de un sólo archivo y lo que se hace en esos casos es utilizar una cabecera (en el inicio del fichero) que indique cuántos sprites contiene el archivo. La cabecera va seguida por los datos individuales de cada sprite (ancho, alto y cuerpo gráfico) desde el primer sprite hasta el último.
Para mayor comodidad en la creación de sprites se ha incluido una pequeña utilidad propia (SPROFF.EXE) mediante la cual podemos extraer bitmaps, sprites y paletas de ficheros de dibujo PCX, RAW y SCR realizados con cualquier programa de dibujo, para crear nuestros propios sprites o utilizar los de otros programas, grabándolos en ficheros SPR, pudiendo incluir anchura, altura, paleta y otros datos gráficos, además de realizar conversiones entre diferentes formatos gráficos simples.

LISTADO 1. DIBUJO DE SPRITES POR SCANLINES

```
void DrawSprite(int x, int y, int ancho, int
alto, char far *sprite )
{
unsigned int offs;
offs = (y*320)+x;

asm {
 push ds
 lds si, sprite
 mov ax, 0xA000
 mov es, ax
 mov di, [offs]
 mov dx, [alto]

BucleY:
 mov cx, [ancho]
 rep movsb
 add di, 320
 sub di, [ancho]
 dec dx
 jnz BucleY

 pop ds
   }
}
```

PROXIMO MES

Por razones de espacio nos vemos obligados a cortar aquí, el próximo mes retomaremos el tema con pantallas virtuales, veremos el caso concreto de pantallas virtuales en el modo 13.