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

PROGRAMACIÓN ACTUAL

AÑO 1 • NÚMERO 5

ARGENTINA 10$ • CHILE 3000$ • PORTUGAL 1500$

INFERNO: ADIOS AL JAVA?

SÓLO 995 ptas

POWERBUILDER: DESARROLLO MULTIPLATAFORMA

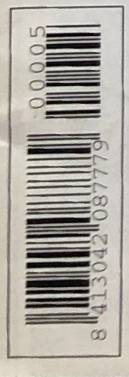
SECCIÓN NIVEL BÁSICO
Registros en el **CURSO DE PROGRAMACIÓN**, Estructuras en el **CURSO DE C**, Operaciones aritméticas en **ENSAMBLADOR**

SECCIÓN LAS EMPRESAS DEMANDAN
Análisis del nuevo **VISUAL C++5.0**, la sentencia Select de SQL en **BASES DE DATOS**, Proyectos **en Visual Basic 5.0**

SECCIÓN BAJO NIVEL
Rotaciones de vértices **ENTORNOS 3D**, Bios VESA en **DEMOSCENE**, Fuentes de letra en **PROGRAMACIÓN GRÁFICA**

SECCIÓN CAMPUS
Cómo instalar un servidor en **PROGRAMACIÓN EN INTERNET**, Autoconf y automake en **LINUX**, concentradores en **REDES LOCALES**, Paquetes en **ADA**

Prensa Técnica



VISUAL C++5.0: FARAÓNICO

EN NUESTRO CD ROM

JAVA WORKSHOP 2.0
TCL/TK 8.0 A2
COMPILADORES DE BASIC
WILLOWS PARA LINUX

Trials de Lotus
Bean Machine 1.0 y
NetObjects Fusion

Assembler Laboratory v1.0, Compilador de Módula-2, Smalltalk1.02 y mucho más...

Autor: Vicente Cubas y Miguel Cubas

Programación de la paleta (II)

El mes pasado comenzamos con la programación de la paleta, nos quedamos en la lectura y modificación de la paleta, retomamos ahora el tema estudiando los efectos fade.

Un efecto bastante vistoso para nuestros programas, que permite que la imagen desaparezca progresivamente de la pantalla hasta dejarla en negro es el fundido a negro. Más adelante veremos cómo realizar el paso inverso; a partir del color negro pasar con un fundido a otra paleta.

FUNDIDO A NEGRO (FADEOUT)

El fundido a negro va reduciendo de cada color sus correspondientes componentes de forma progresiva. Así, todos los colores tenderán a oscurecerse hasta convertirse en negro.
Para empezar nuestro programa debe tener un bucle que se repita 63 veces, debido a que el 63 es el máximo valor que puede tener una sola componente.
Dentro de este bucle debe comprobarse el valor de las componente de cada color. Si es mayor que cero, se debe decrementar el valor de dicha componente, y si es cero, no decrementarlo y pasar a la siguiente componente.
Al finalizar el bucle todos los colores tendrán el mismo RGB (0,0,0) que corresponde al color negro. Así, aunque en la pantalla esta la misma imagen, no se ve nada porque todos los colores de la paleta son negros. Es un efecto bastante sencillo de realizar, como puede verse en el listado 1. La espera al retrazo vertical antes de cambiar la paleta se hace por 2 causas: la primera, ralentizar el efecto para que pueda ser visto. La segunda, evitar el parpadeo (conocido en lenguaje informático como "snow", nieve) que se produce al cambiar la paleta durante el retrazado de las imágenes.
Podemos hacer un fundido más lento esperando 2, 3 o más retrazos, para conseguir distintas velocidades.
El listado 1 es un ejemplo de fundido a negro, en el que se pueden ver las distintas partes del proceso del fundido.
Se debe hacer una copia de la paleta antes de hacer el fundido si se quiere volver a utilizar la paleta original de la pantalla, por ejemplo para volver a poner la paleta correspondiente a la imagen.

FUNDIDO A COLOR (FADEIN)

En este fundido se trata de reconstruir la paleta a partir del color negro. Para esto, hace falta tener la paleta que se desea obtener al final del proceso para realizar las comprobaciones dentro de nuestro bucle.
La diferencia con el ejemplo anterior es que ahora la comprobación no se hace sobre el cero, sino sobre el valor de la paleta que se desea obtener. Por tanto, el bucle debe realizarse de forma inversa al caso anterior, en vez de decrementar cada elemento de la paleta, debe incrementarse desde 0 hasta el color indicado en la paleta final.
Este ejemplo está ilustrado en el listado 2.

ROTACION DE PALETA

Otro posible efecto que se puede realizar con la paleta es su rotación. Este efecto consiste en que los colores de desplacen en un sentido, y esto se consigue copiando en cada posición el color anterior. De esta forma el color 1 pasa a ser el mismo que el color 0, el color 2 al 1, etc. de forma sucesiva.
Para este efecto se debe conseguir que se desplacen los colores hacia un lado, y el último pase a ser el primero para que se mantengan los colores de la paleta, sino el último color se perdería (o el primero, dependiendo del sentido de giro de los colores). Para mover un color deben desplazarse las tres componentes conjuntamente, por lo que se debe de tratar a la paleta no por sus 768 componentes, sino como 256 colores compuesto cada uno por un RGB. Además después de cada rotación se debe poner la nueva paleta para ver el efecto en la pantalla, sino se rotaría la paleta en el array pero en pantalla no se vería ningún cambio.
Por tanto, en un modo de 256 colores se debe realizar un bucle que se repita 256 veces (0..255) y desplace cada vez un color.
Si se dejara la paleta así el ultimo color se perdería y el primer color no se cambiaría, por lo que para este efecto se debe mover el ultimo color a la primera posición.
Una posible solución sería la siguiente:

```
char R, G, B;

R = paleta[765];
G = paleta[766];
B = paleta[767];
for (x=3;x<768;x+=3)
```

LISTADO 1. FUNCION QUE HACE EL FUNDIDO DE UNA PALETA A NEGRO

```
void FadeOut(void)
{
  char paleta[768];
  int x, y;

  GetPalette (paleta);

  for (y=0; y<63; y++)
  {
     for (x=0;x<768;x++)
   if (paleta[x] > 0) paleta[x]—;
     WaitVRetrace ();
     SetPalette(paleta);
  }
}
```

LISTADO 2. FUNCION QUE HACE EL FUNDIDO DE NEGRO A UNA PALETA

```
void FadeIn ( char *paleta )
{
  char pal[768];
  int x, y;

  for (y=0;y<768;y++) pal[y] = 0;

  for (y=0; y<63; y++)
  {
     for (x=0;x<768;x++)
   if (pal[x] < paleta[x]) pal[x]++;

  WaitVRetrace();
  SetPalette(pal);
  }
}
```

```
{
    paleta[x] = paleta[x-3];
    paleta[x+1] = paleta[x-2];
    paleta[x+2] = paleta[x-1];
}
paleta[0] = R;
paleta[1] = G;
paleta[2] = B;
```

Con este proceso se obtiene la paleta rotada. Una vez acabado el proceso debe colocarse esta paleta con los métodos que se han explicado anteriormente (mediante el DAC, interrupciones del MS-DOS, etc.) para que en pantalla aparezca la paleta ya rotada. Si se repite el proceso se obtiene una rotación contínua de la paleta.

LAS FUENTES DE LETRA

En anteriores entregas nos adentramos más en la programación gráfica con el cálculo del offset y el posicionamiento de los pixels en la VGA, con lo que ya podemos dibujar distintas formas gráficas en pantalla. En esta entrega nos encargamos de establecer comunicación entre el software y el usuario mediante las fuentes de letra.
Las fuentes de letras son la base de la información textual y muchas veces se hace imprescindible para el programador ofrecerle al usuario una serie de datos e informaciones, tales como un menú de ayuda, cualquier texto de pantalla, información en los botones del entorno gráfico, etc. En los modos gráficos ya no disponemos de la fuente de letras de que disponíamos en el modo de texto, así que tenemos que ingeniárnoslas para, a partir de las primitivas gráficas de que disponemos, representar en pantalla el equivalente a los símbolos gráficos que representan las letras.
En anteriores artículos vimos la base de la programación gráfica con el desarrollo de la función PutPixel(), que representaba un punto en pantalla con el color especificado. En principio, el manejo de ésta función va a ser la base de las fuentes de letra, pero ahora introduciendo en nuestros ejemplos más ensamblador proporciona un mejor resultado en cuanto a velocidad de ejecución y eficiencia se refiere.

ORIENTACION DEL CURSO

En principio veremos, como siempre, la manera más sencilla y estándar de hacer las cosas y luego haremos una adaptación particular a MS-DOS usando ensamblador para acelerarlas. Puede decirse que éste curso consta de dos objetivos o partes: una es proporcionar pseudocódigo y conceptos para poder aplicar nuestros conocimientos a cualquier plataforma y sistema operativo, mientras que en la otra parte del curso nos encargamos de que el lector aprenda a utilizar los conceptos y programar de forma más optimizada, añadiendo código en ensamblador.
En este sentido de aplicación al MS-DOS, a medida que el curso avance, se irá eliminando código C y añadiendo nuevas instrucciones ASM que dotarán a nuestros programas de mejores resultados. Como siempre dispondremos del seudocódigo de cada función, será sencillo portarlo a cualquier otro sistema simplemente cambiando la función más interna, la específica de cada sistema operativo. Esto quiere decir, en pocas palabras, que si queremos portar la rutina de dibujar sprites a Linux, por ejemplo, tan sólo habremos de sustituir la rutina PutPixel() en asm que habíamos creado para DOS por otra específica para Linux. De ésta manera, con un pequeño cambio podemos adaptar todas las rutinas y luego optimizarlas para cada sistema.

DEFINICION DE LETRA

En los modos de texto la unidad mínima era la celdilla (unidad de pantalla en que cabía un carácter Ascii con su correspondiente color de plano y de fondo), y disponíamos de algunas funciones de la BIOS que nos permitía escribir caracteres en estas celdillas. En los modos gráficos no disponemos de estas celdillas. A cambio disponemos de una resolución mayor de pantalla en la que podemos representar las letras como conjuntos de pixels.
Como muchos ya habrán intuido, vamos a tratar cada letra como un bloque de pixels (como un bitmap o sprite), de manera que podremos guardarlas en arrays y dibujarlas en pantalla con uso de bucles anidados.
Lo anterior viene a decir que nosotros podemos crearnos nuestra propia definición de la letra 'A' (por poner un ejemplo) definiéndola como si fuera un sprite y dibujarla en pantalla de igual manera que hicimos en el número anterior:

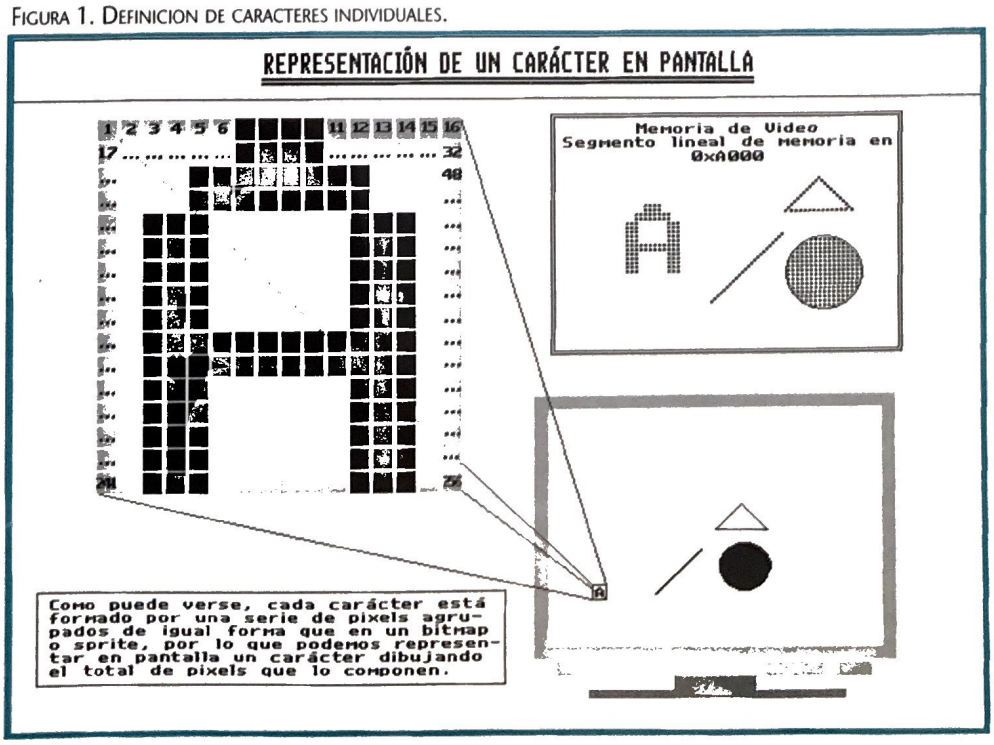
El caracter "A":

```
...XXXX...
.XXX..XXX.
XXX....XXX
XXXXXXXXXX
XXXXXXXXXX
XXX....XXX
XXX....XXX
XXX....XXX
```

LISTADO 3. EJEMPLO QUE PONE LA PALETA CON REP OUTSB.

```
void SetPal (char far *paleta)
{
    lds si, paleta
    mov cx, 768
    xor al, al
    mov dx, 3c8h
    out dx, al
    inc dx
    rep outsb
}
```

FIGURA 1. DEFINICION DE CARACTERES INDIVIDUALES.



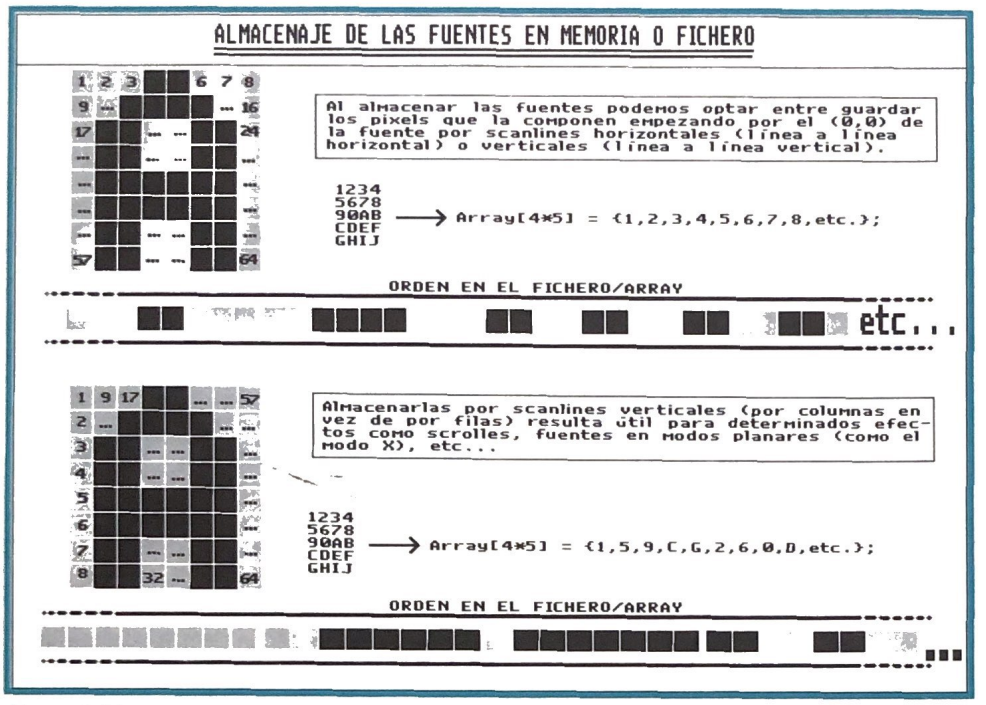


FIGURA 2. METODOS DE ALMACENAJE DE LAS FUENTES.

Este carácter que nosotros hemos dibujado con distintos puntos (representados por las X) y por huecos donde no se dibuja (en este caso los puntos '.'), podemos transformarlo a códigos de colores comprensibles por el ordenador. Supongamos que queremos una letra azul (color 1) con fondo negro (color 0). Nuestro anterior carácter viene a ser representado ahora como:

El carácter "A":
0001111000
0111001110
1110000111
1111111111
1111111111
1110000111
1110000111
1110000111

Al transformalo en un array, quedaría almacenado así:

char Letra[10][8]=
{0,0,0,1,1,1,1,0,0,0,0,1,1,1,0,0,1,1,1,0,etc ... }

Ya tenemos una letra guardada en el array. ¿Cómo la dibujamos en pantalla? Muy sencillo: supongamos inicializado cualquier modo gráfico (ya sea 320x200, 640x480 ó 1024x768) y disponiendo de una función PutPixel(x,y,color) que dibuje el punto (x,y) con el color especificado. Dibujar la letra en la posición (160,100) (por ejemplo) se reduce a dibujar todos los pixels que estén activados (con el color 1) y no dibujar nada en los que no deban ser modificados (el fondo de la letra, que es el color 0 en este caso):

```
#define ANCHO 10
#define ALTO  8

for( by=0; by<ALTO; by++)
 for( bx=0; bx<ANCHO; bx++)
  PutPixel(160+bx, 100+by, Letra[bx][by]);
```

Nótese que podemos crear las letras del tamaño que deseemos, de 8x16, de 16x16, etc. Así, eliminamos el problema de los modos de texto que disponían de un tipo de letra con un tamaño fijo. Ahora podemos representar en pantalla las fuentes del tamaño y tipo que queramos. Podríamos haber dibujado la A inclinada en el array (con lo que representaría una letra cursiva), podíamos haberla dibujado más rellena (negrita), usar otros estilos de letra, etc. En modo gráfico disponemos de la libertad de creación con la resolución de pantalla como única limitación. Cualquier forma gráfica, siempre que quepa en la pantalla, está en nuestras manos.
En el ejemplo anterior hemos anidado dos bucles de manera que en su interior contenía la instrucción PutPixel() que dibujaba el pixel correspondiente del array bidimensional en pantalla. La letra ha sido guardada en un array bidimensional que contiene <anchoxalto> pixels. También es posible guardar una letra en un array unidimensional, cuya ventaja es la de disponer de los pixels que forman la letra de manera más intuitivamente lineal en memoria. Los arrays bidimensionales son más sencillos de leer (en cuanto a lectura y compresión del código se refiere), pero cuando nos encontremos trabajando con distintos modos gráficos es más conveniente disponer de memoria lineal y acceder a estos arrays con la formula (ya bastante conocida por los lectores) de:

*Color (x,y) del array = Array[Ancho*y+x]*

Esto nos va a beneficiar en gran parte a la optimización del código que queramos desarrollar y siguen siendo igual de legibles que los bidimensionales:

Array unidimensional:

```
char Letra[ancho*alto] =
{
0,0,1,1,1,1,0,0,
0,1,1,0,0,1,1,0,
1,1,0,0,0,0,1,1,
1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,
1,1,0,0,0,0,1,1,
1,1,0,0,0,0,1,1,
1,1,0,0,0,0,1,1
};
```

De modo que, a partir de ahora, la mayoría de los ejemplos y listados estarán orientados mediante el uso de arrays unidimensionales, dada la sencillez y linealidad en memoria.

LAS FUENTES DE LETRAS

Las fuentes de letra o de texto son un conjunto de letras guardadas en un array o fichero (con la posibilidad de utilización en nuestros programas) desde donde podemos acceder a ellas para dibujarlas en pantalla.
Cada una de estas letras están guardadas de la manera en que guardábamos la letra 'A' en el array de memoria, pero ahora, en lugar de guardar una sola letra, guardamos todas las que contiene la tabla de caracteres. De esta manera, a la hora de desarrollar un programa seremos capaces de dibujar tanto caracteres individuales como frases completas, éstas últimas dibujando todos los caracteres que la componen consecutivamente en pantalla.
El modo texto nos limita a utilizar el único estilo de letra que existe (aunque puede cambiarse como veremos al final de este texto), pero en los modos gráficos podemos crear nuestros propios estilos que harán diferenciar a nuestros programas de los demás.
Hasta ahora hemos generalizado con las fuentes como conjunto de letras almacenadas en un array o fichero, donde cada uno de los caracteres está compuesto de tantos colores como de ancho x alto tenga dicho carácter (es

decir, un bloque o array de pixels). Pero las fuentes pueden ser guardadas de diversas maneras dependiendo del uso que se le vaya a dar. En este capítulo abordaremos las fuentes desde diferentes puntos de vista, planteando las diversas aplicaciones que pueden llegar a tener.
Las fuentes multicolor, las monocolor y las monotextura son algunos de los puntos que vamos a tratar en este capítulo, nombrando sus aplicaciones así como sus ventajas y desventajas.

FUENTES MULTICOLOR

Aunque todas las fuentes de texto tengan la finalidad de dar información, no todas tienen los mismos métodos de almacenaje. Las fuentes multicolor tiene la ventaja de ser fáciles de leer y escribir. El método que utiliza ésta fuente está basado en el tamaño que ocupa cada uno de los colores que forman las letras. Simplemente hemos de guardar en un array los colores que representan el sprite de la letra, de manera que si cada letra es de 16x16 (16 pixels de ancho por 16 de altura), para guardar una sola letra necesitaremos un array de 16x16=256 elementos. Trabajando en el modo 13h, en que cada color es representado por un byte, necesitaremos un array de 256 bytes (ya que cada color está entre 0 y 255 y éste es el rango de un byte). En otros modos como SuperVGA, donde cada color está representado por 2 ó 3 bytes (hasta 16.7 millones de colores), utilizaremos arrays de ints o longs para esto. Como el modo en que aplicamos nuestros ejemplos es el 13h tomaremos el byte (char) como el tamaño de cada elemento del array:

```
char Letra[16*16];
```

Aquí tenemos una función que nos dibuja un carácter de 16x16 guardado en un array lineal (o unidimensional), imprimiendo todos los pixels que lo forman de manera consecutiva:

```
void PutLetter (int x,int y)
{
 int bx, by;

for (by=0;by<16;by++)
 for (bx=0;bx<16;bx++ )
  PutPixel(x+bx, y+by, Letra[(by*16)+bx]);
}
```

Este ejemplo nos muestra la manera en que una letra almacenada en una array lineal es imprimida en pantalla mediante la función PutPixel(). Pero si recordamos, las fuentes son un conjunto de letras (como la del listado) guardadas en un array. Ahora hemos de guardar el "set" completo de caracteres. De nuevo recurrimos a arrays bidimensionales:

```
char Fuente[16*16][256];
```

En este array bidimensional guardamos espacio para 256 caracteres de 16x16 pixels. Como ya hemos comentado, la preferencia de usar arrays unidimensionales, y sabiendo que cada letra ocupa 16*16=256 bytes(en nuestro ejemplo), si queremos almacenar n letras necesitaremos (ancho*alto)*n bytes:

```
char Fuente[16*16*256];
```

De tal modo que si queremos acceder al carácter 71 (letra G) siendo que por delante están almacenados los demás caracteres, habrá que hacer un cálculo que nos de la dirección del comienzo de la letra 'G'. Y una vez obtenido el resultado del cálculo podremos, a partir de dicha dirección, empezar a leer los datos del carácter 71. Si tenemos una fuente de 16x16 con 256 caracteres y queremos obtener la dirección de comienzo de la letra 'G' (carácter 71) se tendrá que hacer el siguiente cálculo:

```
offset = carácter * (ancho*alto);
```

El listado 1.A nos muestra cómo, con un cálculo y dos bucles anidados, es posible visualizar un carácter contenido en una fuente de letras en VideoRAM, así como su versión ensamblador optimizada en el listado 2.A.
Este estilo de fuentes tiene la ventaja de poder contener variedad de color en un mismo carácter, pero por eso mismo tienen el inconveniente de ocupar mucho tamaño. Existen otros métodos que nos ayudan a ahorrar espacio, pero tienen unas desventajas que ya se comentarán más adelante.

FUENTES MONOCOLOR

Las fuentes monocolor son un estilo de letra muy diferente al de las multicolor. Un letra multicolor estaba compuesta por bytes que representaban los colores, y era así como se guardaban en el array o fichero. Las fuentes monocolor tienen otro método de escritura y lectura que nos ayudan a ahorrar memoria. En una letra monocolor de 8x16 (las más comunes) un carácter ocuparía 16 bytes debido a que cada color del carácter está guardado de forma simbólica en un bit, donde hay color si el bit está activado y no lo hay si el bit no lo está. Por lo tanto, un byte puede guardar la información de los 8 colores que debe tener la letra con el estado de sus bits.

LISTADO 1A.

```
/*_____________________________________
_____________
WriteChar(); Dibujo de caracteres multicolor
en lenguaje C.

Pseudocódigo:
 OFFSET_CARACTER = caracter*(ancho*alto)
  BUCLE1 Desde 0 a ALTOB
   BUCLE2 Desde 0 a ANCHO
    PutPixel(X+BUCLE2, Y+BUCLE1 ,
FUENTE[OFFSET])
   END BUCLE2
  END BUCLE1

_____________________________________
_____________ */

void WriteChar ( int x, int y, char ncar , int
ancho, int alto, char *fuente )
{
int offset_car;
char bx , by;

offset_car = ncar*(ancho*alto);

for (by = 0;by < alto;by ++)
 for (bx = 0;bx < ancho; bx++ )
  PutPixel ( x+bx , y+by ,
fuente[offset+(by*ancho)+bx] );
}
```

```
Byte 1:  00111100b
Byte 2:  01100110b
Byte 3:  11000011b
Byte 4:  11000011b
Byte 5:  11111111b
Byte 6:  11111111b
Byte 7:  11000011b
 ...     ...
Byte 16: 11000011b
```

Y en el array quedaría:

```
char
Letra[16]={60,102,195,195,255,255,195,...,19
5};
```

Sabiendo esto, ahora sólo hay que comprobar el estado de cada uno de los bits que contiene un byte con el fin de dibujar un pixel por cada bit que se encuentre activo. Esto debe hacerse con el cálculo del AND lógico de los operandos (& en lenguaje C), que nos da el resultado de 1 cuando dos bits se encuentran activos. De tal forma debemos calcular el AND entre un byte de la letra y el byte que contenga a 1 el bit que queramos comprobar. Así nos dará como resultado un

valor mayor de 0 si el bit especificado está activo y 0 si no lo está.

11000011b AND 10000000b = 10000000b (activo)
11000011b AND 01000000b = 01000000b (activo)
11000011b AND 00100000b = 00000000b (no activo)
... (comprobar con todos los bits) ...

Teniendo una fuente de 8x16, el código que visualiza un carácter contiene un único bucle que se repite 16 veces de alto que tiene el carácter (16 bytes), en cuyo interior desarrolla los cálculos que comprueban los bits del byte actual.

```
char Letra[16];
void PutChar (int x,int y,char color)
{
 char valor;
 for (bucle=0;bucle<16;bucle++)
 {
  valor = Letra[bucle];
  if ((valor & 128) != 0) PutPixel
(x,y+bucle,color);
  if ((valor &  64) != 0) PutPixel
(x,y+bucle,color);
  if ((valor &  32) != 0) PutPixel
(x,y+bucle,color);
  if ((valor &  16) != 0) PutPixel
(x,y+bucle,color);
  if ((valor &   8) != 0) PutPixel (x,y+bucle,color);
  if ((valor &   4) != 0) PutPixel (x,y+bucle,color);
  if ((valor &   2) != 0) PutPixel (x,y+bucle,color);
  if ((valor &   1) != 0) PutPixel (x,y+bucle,color);
 }
}
```

Este ejemplo visualiza en pantalla un solo carácter guardado en memoria. Pero al tener una fuente con todos los caracteres habrá que hacer el cálculo que realizábamos con las fuentes multicolor para desplazarnos al comienzo del carácter deseado (listado 3.A).

FUENTES MONOTEXTURA

Aunque las fuentes monocolor tienen la ventaja de ocupar poco espacio, su desventaja es que se dibujan en pantalla con el único color especificado por la función PutChar(x,y,color). Las fuentes monotextura van a tener también la ventaja de ocupar un mínimo espacio y la posibilidad de tener variedad de colores, algo que las monocolor no permitían.
Si recordamos, las letras multicolor guardaban el color en la misma fuente donde el color de un pixel lo contenía uno y cada uno de los bytes con los que eran formados los caracteres.

Ahora las fuentes monotextura podrán tener aplicada una variedad de colores como las multicolor guardando un bloque de bytes (textura) en un array que contenga tantos bytes como de alto por ancho sean los caracteres de la fuente monocolor. Así se podrá asignar cada uno de los colores de la textura a los pixels que forman un carácter. Aquí se muestra la manera en que una letra monocolor es dibujada en pantalla aplicándosele una textura de color.

```
char Letra[16];
char Textura[8*16];

void PrintLetter ( int x , int y )
{
char offset_letra, offs_text, valor;

offset_letra = offs_text = 0;

for(bucle=0;bucle < 16;bucle++,offset_letra++)
 {
  valor = Letra[offset_letra];
  if ((valor & 128) !=0)
   PutPixel(x,y+bucle,Textura[offs_text]);
   offs_text++;

  (...resto de condiciones...)

  if ((valor & 1) !=0)
   PutPixel(x+7,y+bucle,Textura[offs_text]);
   offs_text++;
 }
}
```

Ésta función nos muestra cómo se puede aplicar una textura guardada en un array a un carácter de una fuente monocolor. Pero si disponemos de un estilo de fuente monocolor y un array que contenga una textura podremos ser capaces mediante una función (listado 4.A) de dibujar en pantalla cualquier carácter de la fuente. Éste método puede ser muy útil y sencillo de manejar para simples programas que no requieran mucho espacio y, sin embargo, tengan una buena presentación en cuanto a caracteres.

FUENTES VECTORIALES

Las fuentes vectoriales son un tipo más complejo de fuentes cuya definición se realiza mediante vectores que representan la imagen de la letra. Gracias a esto, es posible rotarlas, escalarlas, etc., de una manera puramente matemática, rotando, escalando y trasladando los vectores que las definen. Este es el caso de las fuentes de Windows, por ejemplo, y que vamos a dejar por ahora debido a su complejidad.

EFECTOS CON LAS FUENTES

Además de la simple impresión de las letras podemos aplicar algunos efectos a las fuentes, tales como visualizar los textos centrados en pantalla o en un punto, dar sombra a éstos con tal de resaltar su imagen, invertirlos, etc.
El centrado de texto se consigue con una simple operación que calcula la coordenada X a partir de la cual el texto debe ser visualizado para estar centrado en un punto. Supongamos que quiero calcular la coordenada X de un texto que tiene n pixels de largo, contando los espacios entre caracteres, y queriéndolo centrar en el punto (100,50). El cálculo sería de la siguiente forma:

X_INICIAL = 100 - (n/2)

Otro nuevo efecto sería dotar a un texto de sombra, produciendo un efecto de relieve que da una imagen diferente de la habitual. Dar sombra a un texto no es nada complicado ya que se logra dibujando dos veces la cadena en diferentes posiciones. Teniendo como coordenadas los valores X e Y, la cadena que representa la sombra se dibujará en X+1 e Y+1, y la misma cadena se dibuja a continuación en las coordenadas X e Y. De tal forma el texto queda escrito sobre la sombra que se encuentra en una posición más hacia la derecha y hacia abajo (aunque también podemos dibujar más alejada o hacia arriba, pero el efecto más visual se consigue con (+1,+1)).
Para invertir un carácter tan sólo hemos de dibujar los pixels que lo componen en sentido inverso. En el caso de las fuentes monocolor es preciso comprobar los bits de todos los bytes, pero si antes lo hacíamos analizando del último bit al primero, ahora podemos hacerlos de forma invertida. Por tanto, el resultado de dibujar la letra 'F' invertida quedaría así:

```
XXXXXXXX
XXXXXXXX
.....XXX
..XXXXXX
..XXXXXX
.....XXX
.....XXX
.....XXX
```

Estos son algunos de los efectos que pueden hacerse con las fuentes de texto, pero todavía quedan muchos que deben salir de la propia imaginación del programador, como letras en cursiva al incrementar la x de dibujo en cada pixel, aplicación de la función seno durante el dibujo, escalación, etc.
Continuará...