

PROGRAMACIÓN DE GRÁFICOS EN LENGUAJE C

Para producir la imagen de video, la mayoría de los miembros de la familia PC requiere un adaptador de video. Los adaptadores de video fueron diseñados para el PCjr, pero inmediatamente se generalizó su uso al resto de las PC's, a tal punto que se llega a un nivel como el de VGA (Video Graphics Array) quien claramente ha demostrado ser el adaptador de video más popular de los últimos tiempos.

Las aplicaciones de software han evolucionado gigantescamente. Si nos remontamos una o dos décadas en el tiempo veremos que el entorno en el que giraban los programas era muy limitado y, de poseer capacidades gráficas, estas eran apenas unos garabatos cuadriculados (monocromo, por supuesto). Hoy en día no nos imaginamos una aplicación de software sin su adecuado entorno gráfico (La prueba más concisa: El dominio en el mercado de los “*Software for Windows*”). Es por este motivo que todo programador debe tener al menos una noción de programación gráfica.

Adaptadores de Video y Controladores de Gráficos:

El adaptador de video conecta el ordenador al monitor a través de un *chip* llamado controlador de CRT. El adaptador también tiene un conjunto de puertos de E/S programables, una ROM generadora de caracteres y memoria RAM para almacenar información del visualizador.

Los generadores de video operan fundamentalmente según dos modos diferentes, llamados por IBM, *modo texto* y *modo gráfico*. El modo texto solo permite visualizar caracteres (El código ASCII completo), aunque hay quienes programan en este modo de una forma pseudográfica. El modo gráfico se utiliza principalmente para producir dibujos complejos, pero puede reproducir caracteres de texto con gran variedad de formas y tamaños.

Ninguna función gráfica de “C” puede funcionar sin un controlador de gráficos en memoria. Los controladores de gráficos están contenidos en los archivos .BGI (Borland Graphics Interface), los cuales deben estar disponibles en el sistema. Los diversos drivers que Borland provee para esta versión de Turbo C se listan a continuación:

1. CGA
2. MCGA
3. EGA
4. EGA64 (EGA de 64K)
5. EGAMONO (EGA Monocromática)
6. IBM8514
7. HERCMONO (Hercules)
8. ATT400 (Adaptador de AT&T 6300 PC)
9. VGA
10. PC3270 (IBM 3270)

Es importante destacar que todas las declaraciones de funciones gráficas, el Turbo C las realiza en un archivo llamado graphics.h que será necesario incluir en el programa, de la siguiente manera:

```
#include <graphics.h>
```

Inicialización del sistema de gráficos:

Antes de que se pueda usar cualquiera de las funciones gráficas es preciso colocar el adaptador de video en uno de los modos gráficos usando la función **initgraph()**, que tiene la siguiente sintaxis:

```
void far initgraph (int far *controlador, int far *modo, const char far *camino);
```

La función **initgraph()** carga en memoria un controlador de gráficos que corresponda al número indicado por *controlador*. Este número está estrechamente ligado con los 10 enumerados en el tema anterior, pudiendo ser 0 para la opción de autodetección (cuya macro es: DETECT). El parámetro *modo* indica un número entero que especifica el modo de video usado por las funciones gráficas. Por último se deberá especificar un *camino* al controlador si este es distinto al directorio de trabajo.

Los distintos modos, de acuerdo a cada controlador, son los que se muestran en la siguiente tabla:

Controlador	Modo	Valor	Resolución
CGA	CGAC0	0	320 x 200
	CGAC1	1	320 x 200
	CGAC2	2	320 x 200
	CGAC3	3	320 x 200
	CGAHI	4	640 x 200
MCGA	MCGAC0	0	320 x 200
	MCGAC1	1	320 x 200
	MCGAC2	2	320 x 200
	MCGAC3	3	320 x 200
	MCGAMED	4	640 x 200
	MCGAHI	5	640 x 480
EGA	EGALO	0	640 x 200
	EGAHI	1	640 x 350
EGA64	EGA64LO	0	640 x 200
	EGA64HI	1	640 x 350
EGAMONO	EGAMONOH	3	640 x 350
HERC	HERCMONOH	0	720 x 348
ATT400	ATT400C0	0	320 x 200
	ATT400C1	1	320 x 200
	ATT400C2	2	320 x 200
	ATT400C3	3	320 x 200
	ATT400CMED	4	640 x 200
	ATT400CHI	5	640 x 200
VGA	VGALO	0	640 x 200
	VGAMED	1	640 x 350
	VGAHI	2	640 x 480

PC3270	PC3270HI	0	720 x 350
IBM8514	IBM8514LO	0	640 x 480
	IBM8514HI	1	1024 x 768

Para dejar de usar un modo de video gráfico y volver a modo texto, se usa o bien `closegraph()` o `restorecrtmode()`. Sus prototipos son:

```
void far closegraph (void);
void far restorecrtmode (void);
```

La diferencia entre ambas estriba principalmente en que la primera descarga toda la memoria asignada a especificaciones gráficas, mientras que la segunda la deja intacta para permitir el regreso al modo indicado cuando se encuentre la siguiente función:

```
void far setgraphmode (int modo);
```

Ejemplo: Veamos el siguiente programa que alterna de modo gráfico a texto y viceversa para mostrar el uso de estas funciones.

```
#include <graphics.h>
#include <stdio.h>
#include <conio.h>

void CRTModeDemo(void)
{
    struct viewporttype vp;
    int Modo;

    getviewsettings( &vp );
    Modo = getgraphmode(); /* Guardamos Modo para cuando volvamos a gráficos */
    setttextjustify( CENTER_TEXT, CENTER_TEXT );

    outtextxy( (vp.right-vp.left)/2, (vp.bottom-vp.top)/2, "Esto es Modo
Gráfico..." );
    outtextxy( (vp.right-vp.left)/2, vp.bottom-textheight("H"), "Presione
una tecla para Modo Texto..." );
    getch();

    restorecrtmode();
    printf( "Ahora estamos en Modo Texto.\n\n" );
    printf( "Presione una tecla para volver a Modo Gráfico..." );
    getch();

    setgraphmode( Modo );
    setttextjustify( CENTER_TEXT, CENTER_TEXT );
    outtextxy( (vp.right-vp.left)/2, (vp.bottom-vp.top)/2, "Vuelta en Modo
Gráfico..." );
}
```

```

        outtextxy( (vp.right-vp.left)/2, vp.bottom-textheight("H"), "Presione
una tecla para salir...");
        getch();
    }

main()
{
    int Controlador = DETECT;          /* Forzamos la autodetección del controlador*/
    int Modo;                          /* Valor del Modo Gráfico */
    int CodigoError;                   /* Detectamos cualquier error en manejo gráfico */
    initgraph( &Controlador, &Modo, " ");
    CodigoError = graphresult();       /* Vemos el resultado de la
inicialización*/
    if( CodigoError != grOk ){          /* Error mientras inicializaba... */
        printf(" Ha ocurrido un Error: %s\n", grapherrormsg( CodigoError )
);
        exit( 1 );
    }
    CRTModeDemo();
    closegraph();                     /* Retorna el sistema a modo texto, definitivamente */
}

```

Cuando se deja al sistema de gráficos de Turbo C que establezca el modo de video, el programa necesita alguna manera de conocer el entorno. La función **getviewsettings()** devuelve las dimensiones de la ventana y **getmaxcolor()** devuelve el número de colores permitido en el modo de video aplicable.

La sintaxis es la siguiente:

```

void far getviewsettings( struct viewporttype far *info );
int far getmaxcolor( void );

```

Se ve claramente que la función **getviewsettings()** devuelve el entorno de trabajo a través de una estructura pasada por referencia. La estructura **viewporttype** se define en *graphics.h* como se muestra a continuación:

```

struct viewporttype {
    int left, top, right, bottom;
    int clip;
};

```

Los campos **left**, **top**, **right** y **bottom** contienen las coordenadas de los extremos de la ventana. Cuando **clip == 0** no existe salida que sobrepase los límites de la ventana. Si este es distinto de cero, se realizará corte manual para no sobrepasar los límites.

Las funciones básicas:

Las funciones elementales en cualquier lenguaje para el trazado de gráficos en general, son las que dibujan: puntos, líneas y círculos. En Turbo C estas funciones son llamadas **putpixel()**, **line()** y **circle()** respectivamente. Sus prototipos son:

```

void far putpixel (int x, int y, int c);
void far line (int x_ini, int y_ini, int x_fin, int y_fin);
void far circle (int x, int y, int r);

```

La función **putpixel()** coloca un pixel de color *c* en la posición dada por *x* e *y*.

La función **line()** dibuja una línea desde la posición (*x_ini* , *y_ini*) hasta la posición especificada por (*x_fin* , *y_fin*) en el color preestablecido.

La función **circle()** dibuja un círculo de radio *r* centrado en (*x* , *y*).

Ejemplo: El siguiente programa muestra principalmente el uso de **putpixel()** y **circle()**. Su implementación se ha pensado para que sirva también para mostrar el funcionamiento de funciones como **outtextxy()**, **rectangle()**, **bar()** (y las funciones elementales de seteo de parámetros gráficos) que serán analizadas más adelante.

```

#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <graphics.h>

#define ESC 0x1b          /* Defino la tecla de escape */

int MaxColors,MaxX,MaxY; /* Variables Globales */

void Circulos(void);
void Puntos(void);
void Pausa(int);        /* Toma un caracter del Buffer de teclado (si no hay uno, espera
                        a que se ingrese) y verifica si se trata de
ESC o no. */
void Ventana(int);     /* Dibuja una ventana ocupando la mitad de la pantalla
                        El argumento me indica si es la mitad izquierda o derecha.
*/
void Status(int, char *); /* Mensaje (char *) en el recuadro inferior de la pantalla,
en la ventana            especificada (int) */
void Borde(void);      /* Uso interno de Ventana(). */

main()
{
    int Controlador = DETECT;
    int ModoGrafico;

    initgraph( &Controlador, &ModoGrafico, " ");
    MaxColors = getmaxcolor() + 1; /* M xima cantidad de colores*/
    MaxX = getmaxx();             /* Tamazo de la pantalla */
    MaxY = getmaxy();
    while(1) {                    /* Ciclo Infinito: */

```

```

        Puntos();          /* La salida est en la funciøn Pausa() cuando se presiona
ESC */
        Circulos();
    }
}

void Circulos(void)
{
    int MaxRad;          /* Mximo radio permitido */

    Ventana( 1 );
    Status( 1,"Presione una tecla para cambiar (ESC para Salir)" );
    MaxRad = MaxY / 10;    /* Determina radio mximo */
    while( !kbhit() ){
        setcolor( random( MaxColors - 1 ) + 1 ); /* Selecciøn aleatoria de color
*/
        circle( random(MaxX), random(MaxY), random(MaxRad) );
    }
    Pausa(1);          /* Obtiene la selecciøn del buffer */
}

void Puntos(void)
{
    int s = 1958;        /* Semilla del generador aleatorio */
    int i, j, x, y, h, w, color;
    struct viewporttype vp;    /* Declaro variable tipo estructura (Cada miembro
es un dato de la porciøn activa de la pantalla)
*/
    Ventana( 0 );        /* Llamada a funciøn */
    getviewsettings( &vp );
    h = vp.bottom - vp.top;
    w = vp.right - vp.left;
    for( j=0;j<2;j++ ) {    /* 0.- Coloca los puntos // 1.- Los quita */
        srand( s );        /* Recomienda funciøn generadora de nmeros aleatorios */
        for( i=0 ; i<5000 ; ++i ){    /* 5000 puntos en pantalla */
            x = 1 + random( w - 1 );    /* Ubicaciøn Aleatoria */
            y = 1 + random( h - 1 );
            color = (j) ? getpixel( x, y ) : random( MaxColors );
            if( (j) ? ( color == random(MaxColors) ) : 1 ) putpixel( x, y,
(j) ? 0 : color);
            /* Al colocar los puntos no hay problema. */
            /* Al quitarlos tengo que sincronizar. */
        }
    }
    Pausa(0);          /* Espera respuesta del usuario */
}

void Pausa(int cual)
{
    static char msg() = "Presione una tecla para cambiar (ESC para Salir)";
    int c;

```

```

Status( cual, msg );
c = getch();           /* Lee un caracter del buffer */
if( ESC == c ){       /* Quiere Salir? */
    closegraph();     /* Cambiamos a modo texto */
    exit( 1 );        /* Retorna al DOS */
}
if( 0 == c ){         /* no-ASCII? */
    c = getch();      /* Leer devuelta */
}
}

void Ventana( int pos)
{
    setcolor( MaxColors - 1 );      /* Color blanco */
    if( !pos ) {                    /* Equivale a preguntar por (pos == 0) */
        setviewport( 0, 0, MaxX/2-1, MaxY, 1 );
        Borde();
        setviewport( 1, 1, MaxX/2-2, MaxY-1, 1 );
        clearviewport();
    }
    else {
        setviewport( MaxX/2, 0, MaxX, MaxY, 1 );
        Borde();
        setviewport( MaxX/2+1, 1, MaxX-1, MaxY-1, 1 );
        clearviewport();
    }
}

void Status( int pos, char *msg )
{
    int h;

    setcolor( MaxColors - 1 );
    settextstyle( SMALL_FONT, HORIZ_DIR, 4 );      /* Mensaje inferior en letras
pequeñas ...*/
    settextjustify( CENTER_TEXT, TOP_TEXT );      /* ... y centradas */
    setlinestyle( SOLID_LINE, 0, NORM_WIDTH );
    setfillstyle( EMPTY_FILL, 0 );                /* Para que bar() me limpie la pantalla
*/
    h = textheight( "H" );
    if( !pos ) {
        setviewport( 0, 0, MaxX/2-1, MaxY, 1 );
        bar( 0, MaxY-(h+4), MaxX/2-1, MaxY );
        rectangle( 0, MaxY-(h+4), MaxX/2-1, MaxY );
        outtextxy( MaxX/4, MaxY-(h+2), msg );      /* Mostramos pie de
p gina. */
        setviewport( 1, 1, MaxX/2-2, MaxY-(h+5), 1 );
    }
    else {
        setviewport( MaxX/2, 0, MaxX, MaxY, 1 );
        bar( MaxX/2, MaxY-(h+4), MaxX, MaxY );
        rectangle( 0, MaxY-(h+4), MaxX/2, MaxY );
    }
}

```

```

        outtextxy( MaxX/4, MaxY-(h+2), msg );          /* Mostramos pie de
p gina. */
        setviewport( MaxX/2+1, 1, MaxX-1, MaxY-(h+5), 1 );
    }
    settextstyle( DEFAULT_FONT, HORIZ_DIR, 1 );
}

```

```

void Borde(void)
{
    struct viewporttype vp;

    setcolor( MaxColors - 1 );
    setlinestyle( SOLID_LINE, 0, NORM_WIDTH );
    getviewsettings( &vp );
    rectangle( 0, 0, vp.right-vp.left, vp.bottom-vp.top );
}

```

También existe otro par de funciones para facilitar el trazado de líneas:

void far linerel (int dx, int dy);

Este comando es útil cuando trabajamos en forma relativa. Su misión es trazar una línea tomando como referencia la posición actual y, por medio de los parámetros dx y dy , introducimos un incremento positivo o negativo en la coordenada final.

Sino tenemos:

void far lineto (int x, int y);

Que también es utilizada para generar líneas, solo que ahora introducimos la coordenada final en forma absoluta mediante los parámetros x e y .

Ejemplo: A continuación se verá un programa que, ejecutandolo paso a paso, nos mostrará el funcionamiento sutilmente distinto entre ambas funciones, aunque la disposición impuesta hace que con ambas podamos hacer lo mismo.

```

#include <math.h>
#include <conio.h>
#include <graphics.h>

#define MAXPTS    15

struct PTS {
    int x, y;
};    /* Estructura para un punto    */

struct PTS p(MAXPTS),dp(MAXPTS+1)(MAXPTS+1);
    /* Arreglo con intersecciones en forma absoluta ( p[] ) y relativa ( dp[][] ) */

void LineToDemo(double);
void LineRelDemo(double);
void Parametros(double);

```

```

void main( void )
{
    int GraphDriver = DETECT;
    int GraphMode;
    int xasp, yasp;
    double Relacion;

    initgraph( &GraphDriver, &GraphMode, " " );
    getaspectratio( &xasp, &yasp ); /* funci3n que lee el aspecto real de hardware */
    Relacion = (double) xasp / (double) yasp; /* factor de correcci3n */

    LineToDemo(Relacion);
    getch();
    cleardevice();
    LineRelDemo(Relacion);
    getch();

    closegraph();
}

void Parametros( double Rel )
{
    struct viewporttype vp;
    int i, j, k, l, h, w, xc, yc, ka;
    int r, ang, paso;
    double rads;

    getviewsettings( &vp );
    h = vp.bottom - vp.top;
    w = vp.right - vp.left;
    xc = w / 2; /* Determina el centro del circulo */
    yc = h / 2;
    r = (h - 80) / (Rel * 2);
    paso = 360 / MAXPTS; /* Determina numero de incrementos */
    ang = 0; /* Comienza con un ngulo de 00 */
    for( i=0 ; i<MAXPTS ; ++i ) { /* Determina intersecciones absolutas */
        rads = (double)ang * M_PI / 180.0; /* Convierte ngulo a radianes */
        p(i).x = xc + (int)( cos(rads) * r );
        p(i).y = yc - (int)( sin(rads) * r * Rel );
        ang += paso; /* Siguiete */
    }
    for( j=1 ; j<=MAXPTS ; j++ ) {
        for( i=0,l=0 ; i<MAXPTS ; i++,l+=j ) { /* Determina intersecciones relativas entre
si */
            k=l%MAXPTS;
            ka=(l-j)%MAXPTS;
            dp(i)(j-1).x = p(k).x - p((l)?(ka):(k)).x;
            dp(i)(j-1).y = p(k).y - p((l)?(ka):(k)).y;
        } /* La condi3n de adentro es para que el primer movimiento relativo sea 0
*/
}
}

```

```

        dp(MAXPTS)(j-1).x = p(0).x - p(MAXPTS-j).x;      /* Los últimos ser n los
primeros */
        dp(MAXPTS)(j-1).y = p(0).y - p(MAXPTS-j).y;
    }
    circle(xc,yc,r);          /* Circulo de Borde */
}

void LineToDemo( double para )
{
    int i, j;

    Parametros( para );
    for( i=0 ; i<MAXPTS ; ++i ) {          /* Recorro las intersecciones */
        for( j=i ; j<MAXPTS ; ++j ) {     /* Para las intersecciones faltantes */
            moveto(p(i).x, p(i).y);      /* Coordenada inicial */
            lineto(p(j).x, p(j).y);     /* Línea hacia los otros puntos */
        }
    }
}

void LineRelDemo(double para)
{
    int i, j;

    Parametros(para);
    for( i=0 ; i<=MAXPTS ; ++i ) {
        moveto(p(0).x, p(0).y);          /* Coordenada inicial */
        for( j=0 ; j<=MAXPTS ; ++j ) {
            linerel(dp(j)(i).x, dp(j)(i).y); /* Línea relativa a la última intersección
*/
        }
    }
}

```

Lógicamente se intuye que es necesario contar con una función que me establezca el color de dibujo aplicable al usar **line()** o **circle()** (o muchas más). Por ello surge:

```
void far setcolor (int color);
```

El valor de color tiene que estar comprendido en el rango válido para el modo de gráficos actual. Esto es entre 0 y **getmaxcolor()**.

Funciones para el tratamiento de figuras geométricas:

La siguiente es una síntesis de las funciones más usadas para todo tipo de figuras geométricas. Como se puede apreciar, son de funcionamiento bastante intuitivo.

```
void far rectangle (int left, int top, int right, int bottom);
```

Realiza un rectángulo con los límites establecidos.

void far arc (int x, int y, int StartAngle, int EndAngle, int Radio);

Realiza un arco, donde (x , y) son las coordenadas del centro, *StartAngle* y *EndAngle* nos da el ángulo (en radianes) donde comienza o termina respectivamente de trazar el arco, y *Radio* fija el valor del radio.

void far bar (int left, int top, int right, int bottom);

void far bar3d (int left, int top, int right, int bottom, int depth, int topflag);

Usados para la creación de gráficos estadísticos.

void far ellipse (int x, int y, int StartAngle, int EndAngle, int RadioX, int RadioY);

Elipse con centro en (x , y); valores de radio máximo y mínimo (*RadioX* e *RadioY* alternativamente). *StartAngle* y *EndAngle* nos da la posibilidad de no generar la elipse completa.

void far sector (int x, int y, int StartAngle, int EndAngle, int RadioX, int RadioY);

Dibuja y rellena una porción elíptica.

Funciones para el tratamiento de áreas:

Se puede rellenar cualquier figura cerrada usando la función `floodfill()` cuya sintaxis es:

void far floodfill (int x, int y, int ColorBorde);

Al usar esta función para el relleno de figuras cerradas, llámese por las coordenadas de un punto dentro de la figura y el color de las líneas que constituyen la figura (su contorno). Está demás aclarar que deberá tratarse de una figura cerrada. Usando `setfillstyle()` se puede determinar la forma usada para el relleno.

void far setfillstyle (int modelo, int color);

Los valores para *modelo* y sus macros equivalentes se listan a continuación:

Macros	Valor	Significado
EMPTY_FILL	0	Relleno con color de fondo
SOLID_FILL	1	Relleno con textura uniforme
LINE_FILL	2	Relleno -----
LTSLASH_FILL	3	Relleno //////////////////////////////////////
SLASH_FILL	4	Relleno ////////////////////////////////////// con líneas gruesas
BKSLASH_FILL	5	Relleno \\\ con líneas gruesas
LTBKSLASH_FILL	6	Relleno \\\
HATCH_FILL	7	Relleno con espaciado ligero
XHATCH_FILL	8	Relleno con espaciado denso

INTERLEAVE_FILL	9	Relleno con líneas entrecortadas
WIDE_DOT_FILL	10	Relleno punteado con gran espaciado
CLOSE_DOT_FILL	11	Relleno punteado con poco espaciado
USER_FILL	12	Relleno definido por el usuario

Salida de texto en modo gráfico:

Aunque las funciones estándares de texto de Turbo C, tales como la **printf()**, pueden usarse en la mayoría de los modos gráficos, no son la alternativa más flexible. Para aprovechar al máximo el entorno gráfico de Turbo C será necesario usar las funciones de salida de texto en modo gráfico, descritas a continuación:

void far outtext (char *cadena);
void far outtextxy (int x, int y, char *cadena);

Estas funciones sacan la cadena (apuntada por el puntero) a la ventana gráfica predefinida. Las principales ventajas de usar **outtext()** sobre la de usar **printf()** son sus posibilidades en el manejo de strings en diferentes tipos de caracteres, tamaños, direcciones o resolución en las posiciones. También es una ventaja la posibilidad de cortar la salida que desbordaría la ventana. Por el contrario, **printf()** no puede cortar la salida.

void far settextstyle (int tipo, int direccion, int tamaño);

El parámetro *tipo* determina el juego de caracteres a ser usados. Por omisión, es un tipo “mapa de 8x8 bits”. Se puede dar a *tipo* uno de los siguientes valores:

Tipo	Valor	Significado
DEFAULT_FONT	0	Tipo mapa de 8x8 bits
TRIPLEX_FONT	1	Letras Grandes
SMALL_FONT	2	Letras pequeñas
SANS_SERIF_FONT	3	Tipo Sans Serif
GOTHIC_FONT	4	Letra Gótica

La dirección en la que se visualiza el texto, izquierda a derecha o de abajo a arriba, se determina por el valor de *dirección* que puede ser **HORIZ_DIR** o bien **VERT_DIR**, correspondiendo cada uno de estos a 0 o 1 respectivamente.

El parámetro *tamaño* es un multiplicador que aumenta el tamaño del carácter. Su rango válido es de 0 a 10.

Al igual que lo hacíamos con las ventanas gráficas, para obtener información respecto a las especificaciones de texto en modo gráfico, puede usarse una función cuyo argumento es una estructura pasada por referencia.

void far gettextsettings (struct textsettingstype *TextInfo);

Esta estructura ha sido definida en *graphics.h* de la siguiente manera:

```

struct textsettingstype {
    int font;
    int direction;
    int charsize;
    int horiz;
    int vert;
};

```

Cambio de estilo de línea:

Turbo C permite cambiar la forma en la que se dibuja una línea. Todas las líneas son sólidas por omisión, pero mediante la especificación correcta esta puede ser de puntos, de trazos, de puntos y trazos contiguos o personalizado. Para efectuar estos cambios usaremos la función:

```

void far setlinestyle (int estilo, unsigned modelo, int ancho);

```

A continuación se detallan los posibles valores del parámetro *estilo*:

Valor	Significado
SOLID_LINE	Línea continua
DOTTED_LINE	Línea de puntos
CENTER_LINE	Eje de simetría
DASHED_LINE	Línea de trazos
USERBIT_LINE	Línea definida por el usuario

Si se desea un estilo personalizado mediante este último valor, la forma de la línea se introduce por el parámetro *modelo* como si se tratase de un mapa de bits. Por ejemplo si $(\text{estilo}==4)\&\&(\text{modelo}==0x1234)$, cada vez que tracemos una línea tendrá una forma que se corresponderá a 0001001000110100 donde los ceros son pixels apagados y los unos son pixels prendidos.

El valor del *ancho* podrá ser NORM_WIDTH o bien THICK_WIDTH siendo el primero una forma de especificar un espesor de 1 pixel, y el segundo un espesor de 3 pixels.

Funciones que trabajan con porciones de pantalla:

Aquí aparece una filosofía distinta de trabajo. Ya no se trata de dibujar con las nociones básicas de geometría (como se venía haciendo), sino de aprovechar el manejo de la memoria de video como un complemento útil a las funciones antes vistas para evitar la redundancia en el dibujo.

La función **getimage()** se usa para copiar una región de la ventana de gráficos en una memoria intermedia. La función **putimage()** pone el contenido de una porción de memoria en la pantalla. La sintaxis es la siguiente:

```

void far getimage (int left, int top, int right, int bottom, void far *buffer);
void far putimage (int left, int top, void far *buffer, int op);

```

La función **getimage()** copia el contenido de una porción rectangular de la pantalla definida por sus coordenadas en la memoria apuntada por el puntero *buffer*.

Se usa **putimage()** para visualizar una porción de pantalla contenida en memoria y apuntada por *buffer*. Mediante el parámetro *op* se determina la forma en la que se escribirá en pantalla pudiendo este tomar alguno de los siguientes valores:

Nombre	Valor	Significado
COPY_PUT	0	Sobreescribir el destino
XOR_PUT	1	OR-Exclusivo con destino
OR_PUT	2	OR con destino
AND_PUT	3	AND con destino
NOT_PUT	4	Invertir la imagen fuente

El tamaño de la memoria intermedia, en bytes, para una región determinada se proporciona por la función **imagesize()**. Se debe usar esta función en lugar de intentar calcular manualmente el espacio necesario, ya que **imagesize()** proporciona el valor correcto con independencia del modo de video que esté en uso. La sintaxis es la siguiente:

unsigned far imagesize (int left, int top, int right, int bottom);

Cabe aclarar que estas funciones que manejan porciones de pantalla directamente de la RAM de video, lo hacen de una forma similar al funcionamiento de la **memcpy()** definida en *mem.h*, solo que ahora uno de los punteros lo prefijamos de acuerdo al valor establecido por el hardware en uso. Esto le da cierta potencialidad a estas funciones aunque para ello tenemos que acercarnos un poco más al *nivel de máquina*.

Ejemplo: Como intento simplificado de hacer un protector de pantalla, veremos el manejo de las funciones mencionadas en un programa que mueve una porción de pantalla a travez de esta.

```
#include <dos.h>
#include <mem.h>
#include <conio.h>
#include <stdlib.h>

#include <graphics.h>

#define ON 1
#define OFF 0
#define MaxPts 6 /* Mximo número de v.rtices en poligono */

#define DentroX(x,w,l,r) ((l)+(x)+(w)-1) > (r) ? ((r)-(l)-(w)+1) : ((x <= 0) ? 0 : x);
/* Control de limite X */
#define DentroY(y,h,t,b) ((t)+(y)+(h)-1) > (b) ? ((b)-(t)-(h)+1) : ((y <= 0) ? 0 : y);
/* Control de limite Y */

struct PTS {
```

```

    int x, y;
};

void main( void )
{
    void Poligonos( void );
    void Imagen( void );
    int xasp, yasp; /* Used to read the aspect ratio */
    int GraphDriver = DETECT, GraphMode; /* Request auto-detection */

    initgraph( &GraphDriver, &GraphMode, " " );

    cleardevice();
    Poligonos();
    getch();
    Imagen();

    closegraph();
}

void Poligonos( void )
{
    void Borde( void );
    struct PTS poly( MaxPts ); /* Espacio en memoria para almacenar v.rtices */
    int i;
    int MaxColors, color; /* Color actual de dibujo */

    Borde();
    MaxColors = getmaxcolor() + 1; /* M ximo nmero de colores */
    while( !kbhit() ){ /* Hasta que el usuario presione
una tecla */
        color = 1 + random( MaxColors-1 ); /* Color aleatorio (NO negro) */
        setfillstyle( random(10), color ); /* Atributo de relleno */
        setcolor( color );
        for( i=0 ; i<(MaxPts-1) ; i++){ /* Determina un poligono */
            poly(i).x = random( getmaxx() ); /* coordenada X aleatoria */
            poly(i).y = random( getmaxy() ); /* coordenada Y aleatoria
*/
        }
        poly(i).x = poly(0).x; /* ltimo punto = primer punto ... */
        poly(i).y = poly(1).y; /* ... para que sea figura cerrada */
        fillpoly( MaxPts, (int far *)poly ); /* Dibuja Poligono */
    }
}

void Imagen( void )
{
    static int WH = 100; /* Unidad de medida para determinar alto-ancho */
    static int PT = 40; /* Tiempo de espera */
    struct PTS ant; /* Coordenadas anteriores para no volver al mismo
*/
    struct viewporttype vp;

```

```

int x, y, dx, dy; /* Coordenadas y Desplazamientos */
int s, ulx, uly, lrx, lry, w, h, p=50;
void *Pantalla, *Memoria; /* Punteros para buffer de pantalla*/

getviewsettings( &vp );
ulx = (vp.right-vp.left)/2 - WH; /* Recuadro en centro */
uly = (vp.bottom-vp.top)/2 - WH/2;
w = (lrx = ulx+WH*2) - ulx; /* Ancho del recuadro */
h = (lry = uly+WH) - uly; /* Altura del recuadro */
Pantalla = malloc( s=imagesize(ulx, uly, lrx, lry) );
Memoria = malloc( s ); /* Reservamos espacio para los punteros
*/

getimage( ulx, uly, lrx, lry, Pantalla); /* La zona delimitada por las coordenadas
es copiada a la zona de memoria
apuntada por Pantalla */
memcpy( Memoria, Pantalla, s*sizeof(size_t) ); /* Copia contenido de punteros
*/
x=ulx; y=uly;
while ( !kbhit() ) {
putimage( x, y, Pantalla, COPY_PUT ); /* Colocamos el contenido
del puntero, superponiendo
lo que exista */
do {
dx = (random(WH)%2) ? -w/p : w/p; /* Desplazamiento
aleatorio X */
x = DentroX( x+=dx, w, vp.left, vp.right );
dy = (random(WH)%2) ? -h/p : h/p; /* Desplazamiento
aleatorio Y */
y = DentroY( y+=dy, h, vp.top, vp.bottom );
} while( x==ant.x && y==ant.y ); /* Si es la posición anterior, vuelve */
getimage( x, y, x+w, y+h, Pantalla ); /* Salva antes de ...*/
putimage( x, y, Memoria, COPY_PUT );
memcpy( Memoria, Pantalla, s*sizeof(size_t) );
delay( PT ); /* Espera */
}
free( Memoria ); /* Liberamos memoria */
free( Pantalla );
}

void Borde( void )
{
struct viewporttype vp;

setcolor( getmaxcolor() - random( getmaxcolor() ) ); /* Seteamos color
aleatorio para el borde
*/
setlinestyle( SOLID_LINE, 0, NORM_WIDTH );
getviewsettings( &vp );
rectangle( 0, 0, vp.right-vp.left, vp.bottom-vp.top );
}

```