UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES
ARAGÓN

PROGRAMACIÓN ELEMENTAL PARA ANIMACIÓN, COMPACTACIÓN Y ALMACENAMIENTO DE ARCHIVOS GRÁFICOS

T E S I S

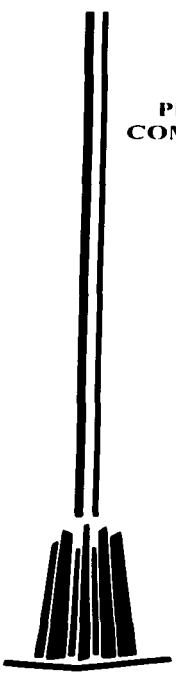
QUE PARA OBTENER EL TÍTULO DE :

INGENIERO EN COMPUTACIÓN

P R E S E N T A :

FRANCISCO CRUZ MATEOS

ASESOR:
LIC. ISRAEL JUÁREZ ORTEGA



MÉXICO

2003

DEDICATORIA

Gracias a Dios y a mis padres, que sin ellos no existiría para poder haber llegado hasta donde estoy hoy.

Dedicado a mi pequeña familia: Paty mi esposa y a María Fernanda que le han dado un rumbo y sentido muy especial a mi existencia, sin duda ellas son algo importantísimo.

A Paty, mi bella esposa, sin su compañía difícilmente tendría razón de ser lo que a diario realizo. A ti mi amiga y compañera.

A mis profesores, y todos aquellos que intervinieron en la formación del profesional que soy, a todos
MUCHAS GRACIAS...

A Israel Juárez Ortega que tiene un don muy especial para dirigir los trabajos finales de titulación, ya que podremos pagar su trabajo pero no su dedicación.

A todas las personas que en el pasado y en el presente han moldeado mi persona con un sin fin de consejos y vivencias agradables o no tan agradables, Dios los bendiga.

A mi querida universidad la UNAM, por permitirme el privilegio de pasar por sus aulas y conocer sus profesores. A ella y todo lo que representa
MUCHÍSIMAS GRACIAS....

ÍNDICE

OBJETIVOS

- Crear un instrumento que permita a los programadores conocer, programar y explotar los aspectos relevantes de las tarjetas gráficas VGA, CGA y EGA mediante un lenguaje de programación.

- Mediante la programación elemental realizar animaciones básicas con archivos gráficos.

- Por medio de algoritmos ya codificados, compactar archivos con información gráfica.

- Conocer desde lo más elemental, todos aquellos aspectos que permitan al programador de cualquier nivel la confección de juegos, graficadores de datos y cualquier tipo de elemento visual.

INTRODUCCIÓN

El color y las animaciones son en nuestros días elementos que sin duda atraen a cualquier persona. Éstas las encontramos desde los dibujos, publicidad, diversión (en los juegos), la televisión, el cine, los libros y en todos los ámbitos modernos productivos o educativos.

Contar con una herramienta de software para generar estos dos elementos resulta muy cómodo ya que solo habrá que comprarlo y comenzar a experimentar; en poco tiempo obtendremos los resultados deseados. Hoy, casi cualquier persona puede trabajar con estos elementos. ¿Y si estas herramientas no estuviesen en el mercado?...

Sin duda la gran gama de productos derivados de los mismos no existirían. Casi todos estos programas son construidos por personas ajenas a nosotros, dotan a la herramienta software de lo que ellos piensan nos puede servir. Estamos confinados a solamente utilizar aquello proveniente de estas grandes compañías productoras de software porque no tenemos la iniciativa de generar software nuestro porque nos resulta cómodo comprar y usar.

Gracias a mi experiencia docente, puedo afirmar que contamos con un gran potencial de creatividad en los adolescentes y jóvenes. Muchos de ellos son usuarios de estos programas y juegos. Los más intrépidos se cuestionan ¿cómo se construyen, o cómo podrían mejorar aquel programa o juego de su preferencia?...

Aquí es donde nace la inquietud de poner al alcance de los jóvenes programadores un trabajo que permita poner las bases en el manejo de los archivos y las tarjetas gráficas como primer paso hacia la generación de programas, juegos, graficadores, etc., ya realizados por programadores nacionales.

He elegido un lenguaje de programación versátil, ampliamente difundido en el mundo académico y comercial; cada aspecto analizado va acompañado de una breve explicación teórica; además de un código en Turbo Pascal 7.0 listo para su inmediata captura.

Desde el primer capítulo donde se detalla la teoría de las tarjetas gráficas, las características del lenguaje de programación y las consideraciones pertinentes en el manejo de archivos gráficos.

En el segundo capítulo abordamos la programación de la tarjeta CGA desde la generación de puntos en la pantalla, mencionando lo que son pantallas virtuales y reales, creando animación; al final se explica lo relacionado al modo 6 de alta resolución de esta tarjeta.

En el capítulo tres, por cierto el más extenso la tarjeta VGA es motivo de estudio ya que representa la más difundida, empleada y mejorada de las tarjetas

gráficas. Aquí se retoman casi todos los aspectos del capítulo anterior (generación de puntos, almacenamiento, lectura, resguardo, animaciones repetitivas), además de abordar la compactación de archivos mediante un algoritmo bastante sencillo ya codificado

El capítulo cuatro inicia nuestro estudio con el empleo de las páginas de memoria para la creación de nuestras primeras animaciones. Además del almacenamiento, resguardo, recuperación, lectura, compactación de archivos gráficos con la tarjeta CGA. La animación empleando el modo 13 y las diferencias entre el modo 13 y los modos 14, 16 de esta tarjeta y el modo 18 de tarjeta VGA.

El ultimo capítulo muestra la conversión de un modo a otro y la portabilidad que debemos incluir a nuestros archivos gráficos al momento de emplearlos en nuestros programas y algunas utilidades para cualquier tipo de tarjeta gráfica.

El presente trabajo de compilación y programación elemental es una pequeña contribución a todos aquellos programadores que no conocen los aspectos importantes de las tarjetas gráficas VGA, CGA y EGA cuando desean generar animaciones, manejo de archivos gráficos (creación, guardado, lectura) y compactación de archivos.

Espero sirva de algo a quienes se interesen en construir programas originales y novedosos...

CAPÍTULO 1

LA UTILIZACIÓN DE TURBO PASCAL 7.0 Y LOS MODOS GRÁFICOS

En este Capitulo abordaremos:

- ***Los conceptos más importantes en la teoría de las tarjetas gráficas***
- ***Las caracteristicas y aspectos importantes del Turbo Pascal 7.0 como lenguaje de programación.***
- ***Los diversos tipos de tarjetas gráficas existentes, además de las consideraciones que debe uno tener cuando las emplea en el manejo de imágenes.***
- ***Los principales formatos gráficos, indicando sus ventajas y desventajas***

1.1 El Turbo Pascal 7.0 como herramienta de desarrollo en modo gráfico

El Turbo Pascal tiene las siguientes características:

- Es un lenguaje estructurado. Entendiéndose por esto, que todos los procedimientos y funciones pueden escribirse independientemente del resto del programa. Esta facultad facilita la creación de una biblioteca de rutinas gráficas autónomas que pueden ser utilizadas en todos los programas posteriores. En modo gráfico, cada vez que sea posible, se hace uso de los procedimientos llamados MOVE y FILLCHAR (se incluyen en el CD).
- La compilación es extremadamente rápida, característica importante cuando ponemos a punto nuestro programa. Para los programas grandes, la compilación separada evita tener que recopilar el programa entero.
- Con la versión 7.0 no hay limitaciones de tamaño del programa. No las hay tampoco en el tamaño de los datos, ya que mediante los punteros, podemos tener acceso a toda la memoria, teniendo en cuenta de que los bloques, no sobrepasen los 64k de memoria, máximo direccionable por el DOS.
- Este lenguaje de programación es meramente académico de gran difusión. Además permite tener un primer contacto con todos los conceptos que hoy se utilizan en casi todos los lenguajes de cuarta y quinta generación, como la gestión de objetos y eventos.

Existen tantas formas de presentar imágenes gráficas, todas y cada una de ellas importantes para alguien. Resulta bastante útil, cómodo, tener la capacidad de soportar cada una de las tarjetas gráficas existentes desde un lenguaje de alto nivel; esto representaba problemas al momento de escribir un código rápido para la gestión de muchas tarjetas gráficas diferentes. Turbo Pascal desde la versión 4.0 solo se había limitado a la programación hacia el adaptador gráfico de IBM y sus limitaciones.

Borland, desde hace tiempo ha lanzado al mercado un sistema gráfico independiente del dispositivo, llamado INTERFAZ GRÁFICO DE BORLAND (BGI, Borland Graphics Interface). En su primera versión, el BGI soportaba los adaptadores gráficos de IBM incluyendo CGA, EGA, VGA, MCGA, y PC3270. Esta versión inicial también soportaba el popular adaptador gráfico Hércules y la tarjeta gráfica ATT 400 utilizada por las computadoras ATT. La versión 5.0 añade soporte para el adaptador gráfico 8514 de IBM y el modo VGA de 256 colores. La tabla siguiente muestra los diversos tipos de tarjetas gráficas y sus drivers que soporta la BGI de Borland, incluye características adicionales.

DRIVERS BGI:

DRIVER	MODOS	VALOR GMODE	RESOLUCION	PALETA
CGA gdriver=1	CGAC0 CGAC1 CGAC2 CGAC3 CGAHI	0 1 2 3 4	320x200 320x200 320x200 320x200 640x200	C0 C1 C2 C3 4 colores
MCGA gdriver=2	MCGAC0 MCGAC1 MCGAC2 MCGAC3 MCGAMED MCGAHI	0 1 2 3 4 5	320x200 320x200 320x200 320x200 640x200 640x480	C0 C1 C2 C3 4 colores 2 colores
EGA gdriver=3	EGALO EGAHI	0 1	640x200 640x350	16 colores 16 colores
VGA gdriver=9	VGALO VGAMED VGAHI	0 1 2	640x200 640x350 640x480	16 colores 16 colores 16 colores
OTROS VALORES:	Existen otros drivers BGI (Hercules, ATT400, IBM8514) que ya han caido en desuso y no re-sultan útiles para el programador.			
SVGA256.BGI: IBM256.BGI:	Borland C dispone ya de drivers SVGA en formato BGI, además de drivers para modos tweaked o unchained como el modo 13X, X, Y, Q, etc...			

1.2 El video en la computadora y paginación de la memoria de acceso aleatorio

Las imágenes que vemos en la pantalla de video son el reflejo de lo que sucede en una zona específica de la memoria. Esta es el área de memoria encargada de recoger toda la información que va a ser puesta en el monitor, recibe el nombre de **MEMORIA DE PANTALLA.**

La memoria de pantalla además de servir al microprocesador y a los programas, es accesible directamente al monitor para visualizar la información. Para ello se pone a disposición de la pantalla una circuitería específica que tiene vía directa a esta zona de memoria.

La comunicación directa entre la memoria de pantalla, los circuitos, el manejo de las E/S (Entradas y Salidas), se ponen a disposición físicamente en un mismo lugar; en la tarjeta conocida como ADAPTADOR DE VIDEO O ADAPTADOR DE PANTALLA, que ocupa una ranura o conector de expansión dentro de la Unidad Central de Procesamiento. Las funciones y componentes del adaptador se pueden resumir como las siguientes:

- - El control del CRT (Tubo de Rayos Catódicos)
- - El manejo de los puertos de E/S programables
- - ROM (Memoria sólo de lectura) generadora de caracteres
- - RAM (Memoria de Acceso Aleatorio) de la memoria de visualización
- - El manejo del cursor
- - Regulación de los modos o formatos de pantalla

Cada adaptador de pantalla utiliza sus propios modos de video y cada uno tiene sus propios requerimientos de memoria, siempre situada en los bloques específicos de la memoria. Para evitar problemas de compatibilidad se dispone que los chips de memoria de pantalla correspondientes se localicen en la propia tarjeta de video.

La paginación de la memoria

En modo texto ofrece la posibilidad de utilizar la memoria sobrante que existe el bloque B una vez utilizados los 4 Kb de memoria. La memoria restante se utiliza para que contenga pantallas independientes de 4kb. A estas pantallas se las conoce con el nombre de **PÁGINAS**, ***(páginas de pantallas o páginas de visualización).*** La tarjeta CGA utiliza 16Kb para su funcionamiento en los modos gráficos (reserva 16 kb, a partir de la dirección B800), mientras que en los modos de texto con 4kb. tiene suficiente; esto posibilita disponer de 4 páginas independientes de 4 kb cada una, cuyos comienzos se sitúan en las direcciones B800, B900, BA00, BB00. En los modos 0 y 1, que sólo utilizan 2kb, se pueden disponer de 8 páginas en lugar de 4. ***Con la paginación de la memoria se agilizan los procesos de visualización en pantalla***, pues mientras que el programa puede estar trabajando activamente en construir una página, podemos estar viendo lo que ocurre en las posteriores páginas no visibles. La tarjeta EGA, que dispone de mayor memoria, puede disponer de más páginas que la tarjeta CGA. El adaptador monocromático sólo tiene 4k de memoria lo que imposibilita la paginación de la memoria de pantalla, aunque mediante software se podría habilitar una parte de la memoria convencional para su paginación.

1.3 Modos de pantalla

Se denominan "MODOS" a las diferentes maneras de presentar los datos en la pantalla. En términos generales, existen dos tipos de modos de pantalla, los cuales son:

1. **MODOS DE TEXTO: Cuya unidad es el caracter**
2. **MODOS GRÁFICOS: Que utiliza a el pixel como unidad**

Pero es como se indicó, en términos muy generales, para profundizar este aspecto, a continuación se describen las características generales de los diversos tipos de modos de visualización en pantalla existentes en el mercado, para una visualización global, se muestra la tabla -1- .

MODO	ADAPTADOR	TIPO	TAMAÑO	COLORES Y DESCRIPCIÓN
0	CGA	TEXTO	40 X 25	Colores suprimidos (sin tonos grises)
1	CGA	TEXTO	40 X 25	16 colores en el primer plano y 8 colores de fondo
2	CGA	TEXTO	80 X 25	Colores suprimidos
3	CGA	TEXTO	80 X 25	16 colores en el primer plano y 8 colores de fondo
4	CGA	GRAFICO	320 X 200	4 colores, gráficos de mediana resolución
5	CGA	GRAFICO	320 X 200	Sin colores, gráficos de mediana resolución
6	CGA	GRAFICO	640 X 200	Sin colores, gráficos de alta resolución
7	MONOCROMATICO	TEXTO	80 X 25	Blanco/negro. Adapatador Monocromático de IBM
8	PC Jr	GRAFICO	160 X 200	16 colores, gráficos de baja resolución
9	PC Jr	GRAFICO	320 X 200	16 colores, de mediana resolución
10	PC Jr	GRAFICO	640 X 200	4 colores, gráficos de alta resolución.
11				NO UTILIZADO
12				NO UTILIZADO
13	EGA	GRAFICO	320 X 200	16 colores, mediana resolución
14	EGA	GRAFICO	640 X 200	16 colores, gráficos de alta resolución
15	EGA	GRAFICO	640 X 350	Blanco/negro, gráficos monocromáticos
16	EGA	GRAFICO	640 X 350	64 colores. Alta resolución
MODO	HERCULES	GRAFICO	740 X 348	Gráficos de alta resolución monocromáticos
MODO	VGA			Este tipo de tarjetas, ofrecen actualmente nuevos modos. Las diferentes combinaciones que se realizan tienen origen en la gran multitud de tarjetas VGA. Una resolución ya estandard es de 640x480 puntos por 256 colores por cada uno d ellos.

Tabla 1. Descripción de los diferentes modos de video

Observando la tabla, se debe considerar lo siguiente:

1) Los primeros 7 modos, MODOS DEL 0 AL 6 se aplican al adaptador de video conocido como CGA que puede mostrar tanto texto como gráficos. Tiene la capacidad de mostrar 16 colores, pero los modos 0 y 2 los colores no pueden ser visualizados, apareciendo en su lugar toda una gama de grises <tonos grises> (concretamente 16 tonos de gris). Estos son conocidos, junto a los 4 grados de grises del modo 5, **MODOS DE COLOR SUPRIMIDO o MOCROMÁTICOS.** Para su visualización puede utilizarse un simple MONITOR DE TELEVISIÓN (aunque necesita de un modulador llamado RF, aunque el tipo de monitores más utilizados con esta tarjeta son conocidos como MONITORES DE VIDEO COMPUESTO, que son monitores monocromáticos que funcionan bien para este tipo de adaptador, pues son capaces de aceptar señales tanto monocromáticas como en color mostrando éstas últimas formas de sombras grises. El tipo de monitor que es ideal para esta tarjeta, es el MONITOR RGB (Rojo-Verde-Azul).
2) Los modos de texto de 40 columnas se diseñaron para que se apreciaran mejor los caracteres que en la pantalla de TV.
3) El MODO 7 **es el único modo que dispone del ADAPTADOR MONOCROMATICO o MDA (Monocrome Display Adapter).** Parece similar al modo 2 de la CGA pero con dos diferencias fundamentales:
4) Los caracteres se dibujan mucho mejor que en la CGA, y los colores no pueden ser visualizados en ningún modo, (ni como tonos de gris), apareciendo en su lugar atributos especiales del modo texto como es el subrayado, video inverso, brillo encendido, etc.
5) El monitor que utiliza es el Blanco y Negro (B/N), también llamado MONITOR DE GESTION DIRECTA, que solo admite caracteres texto con sus atributos.
6) Los modos 8, 9 y 10 son exclusivos del PCjr.
7) Los modos 11 y 12 no son utilizados.
8) Los modos 13, 14, 15 y 16 se encuentran a disposición del adaptador conocido como EGA que puede realizar gráficos y texto con una mayor resolución que la CGA y el adaptador monocromático.
9) Para el modo 16, de los 64 colores se requiere de un **MONITOR especial llamado DE COLOR MEJORADO o MONITOR ECD.** Los modos 13 y 14 son utilizados por el estandard RGB. Los monitores de video compuesto pueden utilizarse con los modos 13, 14 y 15.

10)La tarjeta EGA dispone, además de los modos 13 al 16, los modos que son propios a la CGA y al adaptador monocromático, así se mantiene la compatibilidad hacia atrás que caracteriza a la familia de la computadoras personales.

11)Aunque no corresponde a un estandard marcado por IBM, el MODO HÉRCULES, por llamar de alguna forma al **ADAPTADOR GRÁFICO HÉRCULES** se popularizó tanto en los fabricantes que todos los diseñadores de software la toman en cuentan al momento de desarrollar sus programas.

El adaptador gráfico que ofrece grandes ventajas es el **VGA** y sus posteriores versiones. Las principales características de este adaptador son:

a) **Mantienen la compatibilidad hacia atrás**, es decir, soporta todos los modos anteriores.

b) **La resolución media para los modos gráficos viene a ser de unos 800x600 pixeles**, pudiéndose alcanzar resoluciones, en algún producto de 1024x768 pixeles, mientras que para modo texto la resolución es de 132 columnas x 43 renglones.

c) ***Permite la posibilidad de hasta 256 colores por pixel***, aunque es bastante usual tarjetas inferiores de tan solo 16 colores.

d) Los modos de alta resolución de VGA requieren un monitor especial llamado **MULTISYNC-VGA**, monitor muy caro pero es el único modo de ver las altísimas resoluciones superiores de la VGA. Las resoluciones inferiores a la VGA pueden visualizarse en monitores que no son Multisync.

1.4 Visualización de los modos tipo texto

Engloba a los diferentes modos que tienen la posibilidad de poder mostrar el juego de 256 caracteres ASCII en pantalla.

La presentación normal es de 80 x 25, es decir, la pantalla está dividida en 80 columnas x 25 líneas que conforman 2000 cuadros individuales y en cada posición se localiza un solo caracter.

La unidad en este modo **es el CARACTER**, *aunque éste a su vez está compuesto por pixeles, que en modo texto no se pueden manipular individualmente sino que están sujetas a las características del caracter que muestran.* Es decir, que **si queremos hacer un dibujo en modo texto, no podríamos crearlos pixel a pixel; sino caracter a caracter.**

A la forma de presentar los gráficos (caracter a caracter) se le conoce como **GRÁFICOS EN BAJA RESOLUCIÓN.** Cada posición en la pantalla queda definido por dos bytes situados en la memoria de pantalla, o lo que es lo mismo, el contenido de cada pareja de bytes se refleja en la posición de pantalla. Estos bytes representan dos elementos completamente diferentes pero que están íntimamente ligados entre sí:

- ***Caracter***.- Determina qué caracter de la tabla ASCII aparecerá en una posición determinada.
- ***Atributo***.- Determina en que condiciones debe aparecer dicho caracter (subrayado, video inverso, parpadeo, etc.).

El formato de pantalla más habitual presenta 80 columnas por 25 líneas (80x25) es decir 2000 caracteres por pantalla. Al ser necesarios dos bytes por cada posición, una pantalla de texto necesita 4000 bytes para ser controlada, que son aproximadamente 4 kbytes (sobrando 96 bytes del total de la memoria). Tanto el adaptador monocromático como los modos textos de adaptador CGA utilizan 4Kb de memoria RAM.

Las direcciones de memoria que reservan los modos de texto ocupan 4Kb a partir de la dirección B0000 Esta dirección es válida para todos los modos de texto de todos los adaptadores de video.

La disposición en memoria en modo texto es siempre la misma: ***se dispone en pares de bytes, correspondiendo al CARACTER el primer byte y el ATRIBUTO al segundo, a partir de la dirección B0000 (valor hexadecimal)en memoria RAM.***

La primera pareja de bytes que podemos encontrar en la memoria corresponde a la localización del caracter de la esquina superior izquierda de la pantalla, siguiendo la numeración conforme a la lectura de izquierda a derecha y de arriba hacia abajo. De esta forma, para localizar una posición de la memoria de pantalla, comenzando la numeración en 0 que corresponde a la posición B0000, se utiliza la fórmula:

POSICION = (FILA*80 + COLUMNA)*2

En los modos de texto monocromáticos, los atributos de un caracter se refleja con un solo byte. Este byte que se nos puede presentar en 256 formas diferentes aunque tan solo son válidas 10, las anteriores formas se representan en la tabla -2-:

CODIGO	VALOR BINARIO	APARIENCIA
0	00000000	Invisible
1	00000001	Subrayado
7	00000111	Normal
9	00001001	Subrayado brillante
15	00001111	Normal brillante
112	01110000	Inverso
129	10000001	Normal parpadeante
135	10000111	Subrayado brillante parpadeante
143	10001111	Normal Brillante parpadeante
240	11110000	Inverso parpadeante

Tabla 2. Apariencia del texto

En cambio en los modos de texto de 16 colores se puede utilizar los 256 combinaciones que el byte de atributos permite. En la tabla -3-, podemos observar como se originan los 16 colores en una tarjeta CGA.

BYTE DE ATRIBUTO	APARIENCIA DEL CARACTER
76543210	
1	Componente azul del caracter
10	Componente verde del caracter
100	componente rojo del caracter
1000	intensidad del caracter
10000	Componente azul del fondo
100000	Componente verde del caracter
1000000	Componente rojo del caracter
10000000	Parpadeo del caracter

Tabla 3. Apariencia del caracter

Por ejemplo, si nos encontramos con que un byte de atributo de un caracter es 26, es decir, de un valor binario de 00011010, significa que el caracter se visualizará verde brillante (el valor del segundo y cuarto byte es 1) con un fondo azul (el valor del byte número 5 también es 1).

Los cuatro componentes del caracter (tres colores + intensidad), pueden combinarse entre sí, originando la matriz de 16 colores que caracteriza a los modos de

la CGA. **A este esquema se le llama IRGB (Intensidad Rojo-Verde-Azul):** formado por 8 colores con las distintas combinaciones de los tres colores básicos con el factor de intensidad. La tabla -4- describe lo anterior:

BIT DE INTENSIDAD	COMPONENTE ROJO	COMPONENTE VERDE	COMPONENTE AZUL	DESCRIPCIÓN
0	0	0	0	NEGRO
0	0	0	1	AZUL
0	0	1	0	VERDE
0	0	1	1	CYAN
0	1	0	0	ROJO
0	1	0	1	MAGENTA
0	1	1	0	AMARILLO OSCURO
0	1	1	1	BLANCO (gris brillante)
1	0	0	0	GRIS OSCURO
1	0	0	1	AZUL BRILLANTE
1	0	1	0	VERDE BRILLANTE
1	0	1	1	CYAN BRILLANTE
1	1	0	0	ROJO BRILLANTE
1	1	0	1	MAGENTA BRILLANTE
1	1	1	0	AMARILLO
1	1	1	1	BLANCO BRILLANTE

Tabla 4. Esquema IRGB: matriz de colores con cuatro bits

1.5 Visualización de los modos tipos gráfico

La características principal que diferencia a los modos gráficos de los modos texto **es el PIXEL**, unidad que se controla individualmente. En los modos texto la unidad más pequeña era el caracter y no el pixel. **A esta forma de visualizar los gráficos, pixel a pixel, se conocen como GRÁFICOS EN ALTA RESOLUCIÓN.** Para controlar la visualización de un pixel sólo se necesita un byte. Este es el **byte de atributo que sólo controla el color de visualización del pixel.** El pixel sólo puede mostrar puntos coloreados lo que es en modo gráfico una unidad más simple que el modo texto, pero con ellos podemos construir dibujos más ricos y complejos. A continuación se describen los diferentes modos gráficos que fueron apareciendo con los diferentes estándares de IBM:

- ***El MODO 4 (CGA),*** tiene una resolución de 200x320 puntos, esto supone 64,000 pixeles en la pantalla, o 64,000 bits (8,000 bytes) en la memoria. Cada pixel puede mostrar 4 colores o 4 combinaciones de dos bits, que hace un total de 16,000 bytes o posiciones de memoria (aproximadamente 16Kb).
- ***El MODO 5 (CGA)*** se organiza de una forma parecida al modo 4, sólo que la gama de 4 colores se sustituye por 2, (blanco y negro), con unos requerimientos de memoria de 8kb.
- ***El MODO 6 (CGA)*** tiene dos colores (blanco y negro) y una resolución de 640x200, es decir, el doble de resolución pero solo la mitad de colores que el modo 4, manteniendo así los mismos requerimientos de memoria.
- ***El MODO 13 (EGA)*** requiere de 32kb como resultado de multiplicar la resolución de 320x200 por los 4 bits que definen los 16 colores de cada pixel y dividirlos entre los 8 bits que conforman un byte.
- ***El MODO 14 (EGA)*** necesita 64 Kb, pues es como el modo 13 pero con doble resolución.
- **El MODO 15 (EGA)** nos proporciona 350 líneas por 640 columnas en blanco y negro. Cada pixel está regulado por dos bits que representan 2 atributos típicos del modo texto como son el parpadeo y la intensidad de brillo. Este diseño pretende hacer compatibles las características del modo gráfico con las del modo texto. Así, este modo puede simular características de texto con una matriz de puntos de 8 x 14 puntos. Esto supone una forma de lograr mayor definición en los caracteres y aumentando, por fin, la resolución en su componente vertical.
- **El MODO 16 requiere 64 colores** (que pueden ser visualizados en un monitor especial ECD), esto implica la necesidad de 6 bits para definir el color de cada pixel, como hay 640 x 350 pixeles resulta que son necesarios 168,000 bytes que es mucho más que el bloque A y B juntos, (cabe señalar que los bloques A y B están reservados por el DOS para la memoria de

pantalla). El adaptador EGA debe realizar operaciones entre áreas libres de la memoria RAM y la memoria de pantalla.

- **El MODO HÉRCULES** con una resolución de 720 x 348 pixeles necesita 31,320 bytes de memoria. Es similar al modo 15 de la EGA. No es modo soportado por los compatibles por lo que no tiene definido ningún número, por eso genéricamente se le denomina MODO HERCULES.

Los modos gráficos de la tarjeta VGA y versiones posteriores (IBM lanzó una tarjeta con 17 nuevos modos gráficos, pero en el mercado hay muchísimas más) tiene unos requerimientos de la memoria que supera fácilmente los bloques A y B en el mapa de memoria RAM. Es la propia tarjeta quien debe gestionar la memoria de pantalla incorporando los chips de memoria necesarios para verlos. Algunas tarjetas, con diseños y resolución muy avanzados, disponen de su propio microprocesador para realizar correctamente las tareas de visualización. En la página siguiente se muestra la tabla -5- que ilustra el tipo de modo y la cantidad de memoria requerida, además de indicar en que dirección de memoria comienza.

1.5.1 El texto en modo gráfico

En el modo gráfico, no se hace uso de UN GENERADOR DE CARACTERES, los caracteres texto sólo pueden aparecer en la pantalla si se dibujan pixel a pixel como cualquier otra figura y luego se llaman como conjuntos ya definidos, esto se describe en la tabla 5.

MODO	MEMORIA MINIMA UTILIZADA EN KILOBYTES	DIRECCIÓN DE INICIO
4	16	B800
5	16	B800
6	16	B800
8	16	
9	32	SU LOCALIZACIÓN VARIA
10	32	
13	32	A800
14	32	A800
15	64	A000
16	32	A800
HERCULES	32	B800

Tabla 5. Características adicionales de los modos de visualización

Cuando en modo gráfico aparece un mensaje escrito, este puede ser dibujado de dos formas diferentes:

1. ***Por software.*** Existen programas que son capaces de "dibujar" ellos mismos las frases, informes, etc., que aparecen en pantalla, atendiendo a sus propias necesidades de tamaño, estilo, etc.
2. ***Por mediación de la ROM-BIOS.*** Una tabla de caracteres se almacena en una área específica de memoria de la ROM-BIOS. Esta tabla que comienza en una dirección determinada, se accede a ella por la interrupción 31 y suele comenzar en la dirección F:FA6E y en ella se recogen la

situación ON/OFF de los bits que deben aparecer en la pantalla en forma de pixeles.

1.6 FORMATOS GRÁFICOS

¿Qué son los formatos gráficos? Básicamente, los formatos gráficos son archivos en los cuales se guarda información que conforma una imágen. Cada formato es independiente. Las posibilidades que ofrece cada formato con respecto a la gama de colores, a la compatibilidad, a la rapidez de carga, etc., merece ser explicada para determinar cuál de ellos es el más adecuado.

Con respecto a la estructura, la mayoría posee un **header** que indica al programa que lo solicite las características de la imagen que almacenan; por ejemplo su color, tipo, resolución, etc. Sin importar su nombre, todos los formatos se pueden dividir en dos grandes grupos: **los formatos vectoriales y los formatos bitmap. Cada formato que a continuación se describe, tiene una organización propia de su estructura**.

1.6.1 BMP: Junto con el surgimiento de Windows 3 se desarrolla un nuevo formato gráfico bitmap que constituye el standard adoptado por este entorno operativo, en el cual están almacenadas las imágenes que constituyen los llamados wallpapers. Este formato guarda las imágenes descomprimidas, lo que significa mayor velocidad de carga y mayor espacio requerido. Con respecto a la resolución, cualquiera es aceptable. Las imágenes pueden ser de 1, 4, 8 y 24 bits. La estructura de los BMPs es sencilla: se trata de un header que contiene varias características de la imagen; está compuesto por información acerca del tamaño, el número de colores, y una paleta de colores (si es necesario) de la imagen. A continuación, se encuentra la información que constituye la imágen en sí. Tiene una curiosa forma de almacenarla: comienza desde la última línea inferior. Es por eso que los programas encargados de exhibir los BMPs en pantalla trazan la imagen de abajo hacia

arriba. Es un formato muy utilizado en la actualidad y la mayoría de las aplicaciones lo emplean.

1.6.2 CDR: Es el formato standard de Corel Draw. Es de tipo vectorial, pero pueden insertarse elementos bitmap en las imágenes. Es uno de los formatos con más posibilidades con respecto al color, a la calidad de los diseños y al manejo de fonts. La principal desventaja de este formato es que es único. Por lo tanto, es de los formatos más inestables hasta el día de hoy. Justamente debido a la incapacidad de otras aplicaciones de almacenar imágenes bajo este formato, utilizar CDR puede resultarle una verdadera molestia.

1.6.3 CGM: Es de tipo vectorial y soporta elementos bitmap para incluir en las imágenes. No surge de una aplicación específica como los demás formatos. Fue creado por el ANSI (American National Standards Institute) y posee una gran desventaja: cada programa brinda su organización propia de la estructura. Esto provoca que no todos los archivos CGM sean iguales, y pueden darse casos de incompatibilidad entre archivos de este mismo formato. A pesar de este inconveniente, el CGM es un formato vectorial realmente bueno, y en la actualidad muchas de las aplicaciones que soportan este tipo de formato lo reconocen.

1.6.4 CUT: Proviene de la antigua aplicación D.O.S. llamada Dr. Halo. Es de tipo bitmap y soporta hasta 256 colores. Es un formato muy poco práctico, pues por cada imagen de más de dos colores que almacenamos en un archivo, debe existir otro archivo que indique la paleta de la imagen (por ejemplo, el archivo FOTO.CUT que contiene una imagen de 256 colores debe estar acompañado por el archivo PAL). Si el archivo PAL se pierde, la imagen se tornará blanco y negro teniendo en cuenta la siguiente regla: cada pixel negro queda igual, y todos los demás pixels se vuelven blancos sean del color que sean. Considerando su antigüedad y su ausencia en las aplicaciones más

importantes, **se concluye que este es uno de los formatos realmente inútiles (en todos sentidos).**

1.6.5 DXF: Este el formato vectorial por default de AutoCAD. Soporta hasta 256 colores. Su estructura no contiene información comprimida como en la mayoría de los demás formatos, sino números y órdenes a realizar escritos en ASCII. Esta información indica la ubicación de puntos flotantes matemáticos (floating points), utilizados para exhibir la imagen en pantalla. Este sistema, al ser más lento que el común, requiere hardware avanzado. Hoy, además del AutoCAD, no hay muchas aplicaciones que reconozcan este formato, aunque Corel Draw puede manejar DXFs sin mayores dificultades.

1.6.6 EPS: Este formato utiliza el lenguaje PostScript diseñado en 1982 por los fundadores de Adobe Systems. Soporta gráficos de tipo vectorial y posee limitaciones en cuanto al uso de elementos bitmap. Su principal ventaja es el manejo de fonts: posee un extraordinario número de tipografías a utilizar y una gran cantidad de efectos especiales para aplicar al texto (rotación, variación del tamaño, half-toning, etc.). Tiene su Talón de Aquiles: para imprimir imágenes PostScript hay que utilizar una impresora que reconozca ese lenguaje. Como si esto fuera poco, los archivos son enormes y la memoria puede faltar algunas veces. Las aplicaciones más populares soportan este formato.

1.6.7 GEM: Surge del GEM, una aplicación D.O.S. diseñada por Digital Research. En dicha aplicación se utilizaban dos formatos gráficos: el IMG para gráficos bitmap y el GEM para gráficos vectoriales. Estos dos formatos, aunque no ofrecen grandes posibilidades, son utilizados por importantes aplicaciones en el campo del Desktop Publishing. La principal limitación de este formato es que las imágenes deben ser simples, sin mayores complicaciones, o de lo contrario gran parte de la imagen se perderá. Además, los GEMs solamente

soportan hasta 16 colores. Para manipular este formato, puede utilizarse el GEM Artline que funciona bajo el entorno GEM y Corel Draw reconoce este formato sin dificultad.

1.6.8 GIF: Es el formato gráfico bitmap por excelencia. Fue creado por Compuserve en junio de 1987 y con el paso del tiempo se ha convertido en el formato más difundido en el mundo. Los GIFs utilizan una paleta de entre 2 y 256 colores. Poseen una rutina de compresión muy eficaz que, aunque demora un poco la carga, reduce los archivos a una tamaño mucho menor que otros formatos. El GIF es el formato óptimo para ser bajado de BBS o Internet. La resolución máxima alcanzada es la de 1024 x 768 pixels en 256 colores, pero no hay razón por la cual no pueda crearse una imagen de mayor tamaño. Incluso hay GIFs que almacenan más de una imagen en un solo archivo, Su estructura está basada en bloques, estos pueden contener uno de estos elementos: una imagen, instrucciones acerca de cómo exhibirla, texto, información característica de alguna aplicación, un marcador que determina el final del archivo, etc. Muchos GIFs solamente contienen un bloque que determina su imagen. Todos los GIFs poseen dos tipos de paleta: la paleta global y la paleta local. Existe un bloque llamado comment block, o 'bloque de comentarios', donde puede incluirse un breve comentario personal acerca de la imagen en cuestión. Incluso existe una opción para aplicar a los GIF llamada interlacing. Gracias a la popularidad de este formato, se han desarrollado infinidad de programas shareware para manipular GIFs. Ya sea para exhibirlos, modificarlos, convertirlos o incluso comprimirlos. Si alguno de estos programas modifica el archivo .GIF, es muy probable que aparezca alguna información sobre esta aplicación en el application block.

1.6.9 HPGL: Es el formato utilizado por las impresoras Hewlett Packard. Es un formato de gráficos vectoriales y su extensión puede variar (por ejemplo: HGL, HPP, PGL, PLT, etc.). Posee muchas limitaciones con respecto a la

calidad de los diseños. Generalmente, cuando se crea una imagen y se graba bajo este formato, es fácil descubrir que muchos de los rellenos utilizados han desaparecido. Como si esto fuera poco, las capacidades tipográficas del HPGL son extremadamente pobres y posee una gama de colores que deja mucho que desear. Este formato solamente puede resultar útil si se realizan diseños lineales, como esquemas electrónicos, mecánicos o arquitectónicos, sin mayores complejidades que unas cuantas líneas sin relleno. Es uno de los formatos menos utilizados.

1.6.10 IFF/LBM: Surge de la Commodore Amiga, y en sus siglas no se incluye ninguna palabra como 'imagen' o 'gráfico'. Es porque este formato puede no sólo almacenar imágenes bitmap sino también música, texto, o cualquier tipo de información en general. Con respecto al color, existe una variada gama disponible similar a la de los GIFs. El único inconveniente es la demora en el acceso a estos archivos, ya que la amiga utiliza un sistema de 'planos' para almacenar las imágenes, y las PCs tardan un poco en acceder a dichos planos. Hoy existen algunas aplicaciones que reconocen este formato (además de Deluxe Paint).

1.6.11 MG: Este formato bitmap es el utilizado por la antigua aplicación GEM. Es capaz de almacenar imágenes de entre 2 y 256 colores. No hay restricciones con respecto al tamaño de las imágenes y utiliza una rutina de compresión medianamente eficaz. El único inconveniente es que hay muy pocas aplicaciones que lo utilizan, y se hace un poco pesado tener que convertir este formato a otro más conocido cuando haya que modificarlo y luego volver a convertirlo a IMG.

1.6.12 JPEG: El formato JPEG ofrece los imprescindibles 16 millones de colores (true color), unido a una compresión realmente asombrosa (valores superiores a 20:1 son habituales). Sólo tiene una limitación: para obtener

esos valores de compresión modifica sutilmente la imágen, descartándose su uso en aplicaciones en las que se desea mantener una calidad bit a bit. El diseño de este formato está pensado para almacenar imágenes del "mundo real", también llamadas imágenes de tono continuo, como digitalizaciones o renderizaciones de alta calidad. Si se intenta almacenar imágenes de tipo vectorial o dibujos sencillos no realísticos, se observará como la compresión disminuye enormemente, y las modificaciones hechas sobre la imagen original por el algoritmo de compresión se observan a simple vista. El formato JPEG sólo puede almacenar imágenes de 24 bits (true color), utilizando tres canales para su almacenamiento o de escala de grises, usando sólo un canal. La compresión JPEG consiste en una serie de complejas operaciones matemáticas, tales como: conversión del formato del color, transformación separada de coseno (DCT), cuantizaciones y codificación entrópica. JPEG, junto con GIF, son los formatos de imágenes usados en WWW.

1.6.13 MAC: Originario de las Macintosh, este formato bitmap presenta varios inconvenientes. Por empezar, no utiliza colores y su resolución máxima es de 576 x 720 pixels. La única ventaja existente es la de poder crear una imagen en una Macintosh utilizando programas como el MacPaint y luego trasladar el archivo a una PC para utilizarlo en algún otro programa que se conforme con poco. Por lo demás, es uno de los formatos menos recomendables debido a que muy pocas aplicaciones lo requieren.

1.6.14 MSP: Diseñado por Microsoft para ser utilizado en la antigua versión de Windows Paint, este formato bitmap solamente utiliza blanco y negro, pero no posee limitaciones acerca del tamaño de las imágenes. Posee un método de compresión normal y su estructura difiere del resto de los formatos: está compuesta por una tabla interna que permite que un programa descomprima únicamente las líneas de la imagen que requiere, dejando el resto de la imagen comprimida. De esta manera, se ahorra tiempo al no tener que

descomprimir toda la imagen. Hay grandes garantías de compatibilidad entre los MSP. No es muy utilizado, lo que es un formato un tanto obsoleto.

1.6.15 PCC Surge del PC Paintbrush, de Z-Soft, y es creado por antiguas versiones de este programa. Un PCC es como un fragmento de un PCX completo. Ahora no se utiliza este formato, sino directamente el PCX. Para convertir de PCC a PCX basta con renombrar la extensión, pues estructuralmente son idénticos.

1.6.16 PCX: Uno de los formatos bitmap mas conocidos. Es un formato bitmap y soporta imágenes de hasta 24 bits en color (unos 16 millones de colores). No hay restricciones con respecto al tamaño de las imágenes. Su método de compresión apunta a la rapidez de acceso en vez de a la reducción de tamaño de los archivos. Su mayor virtud: la compatibilidad. La gran mayoría de los programas de desktop publishing y de tratamiento de imágenes en sí soportan este formato, incluso en su última versión de 24 bits.

1.6.17 PIC: La resolución máxima alcanzada por este formato es de 320 x 200 pixels en 256 colores. Si se desea utilizar una resolución de 640 x 480 pixels, los colores deberán ser 16. El método de compresión es eficaz siempre y cuando se trabaje con imágenes relativamente simples, y no con pantallas escaneadas. La escasez de colores en altas resoluciones y la incompatibilidad entre PICs contribuyen a que sea uno de los menos utilizados.

1.6.18 TGA: Es el formato utilizado por las tarjetas Targa. Las imágenes son bitmap y pueden ser de cualquier tamaño y contener tantos colores como se pueda imaginar (de 2 a 32 bits en colores). La principal desventaja es el tamaño de los archivos. Este formato es especial para retocar diseños profesionales más que con simples programas shareware, debido a que la amplia gama de

colores produce un efecto muy realista y sumamente elaborado. También es muy útil cuando se trabaja con scanners de alta calidad. **Es el mejor formato por su tratamiento de los colores.**

1.6.19 TIFF: Más que una imagen en una archivo, el formato TIF contiene una serie de bloques que conforman la imagen. Los bloques contienen cierta información sobre la imagen en sí, su tamaño, su manejo del color, información a las aplicaciones que utilicen ese archivo, texto, y hasta thumbnails. Un thumbnail o miniatura es una pequeña representación de una imagen mucho más extensa, a la cual el programa accede rápidamente y no pierde tiempo descomprimiendo toda la imagen. Sirve para ver el contenido del archivo de una manera rápida y segura. Este formato es totalmente compatible con PC y Macintosh. Soporta gran cantidad de colores y es uno de los formatos preferidos por hoy en día. Es el formato más usado cuando se trabaja con scanners debido a su útil manejo del color.

1.6.20 WMF: Las funciones gráficas complejas de Windows han provocado la creación de WMF. Es un formato muy útil y sus archivos son increíblemente fáciles de crear. Las aplicaciones Windows utilizan este formato como un tipo de 'grabadora gráfica', al copiar en un archivo los comandos para realizar la imagen en cuestión ahorrando una cantidad considerable de espacio. Teóricamente, cualquier cosa que se pueda dibujar en una ventana Windows puede ser almacenada en un WMF, ya sea imágenes bitmap, texto, o gráficos lineales sumamente complejos. Gracias a su facilidad de manejo, hay muchas aplicaciones que lo utilizan en nuestros días.

1.6.21 WPG: Excepto el formato EPS, WordPerfect no importa otro formato que no sea el WPG. Este formato soporta tanto gráficos bitmap como vectoriales de 2 a 256 colores. A veces ocurren ciertas incompatibilidades de conversión al trabajar con gráficos bitmap y vectoriales en la misma imágen. La tarea de

conversión de otros formatos al WPG se hará algo habitual al trabajar con WordPerfect, aunque no por eso deje de ser bastante incómodo.

Ya descritos los términos fundamentales en este primer capítulo, a continuación se aborda la programación de la primera tarjeta gráfica conocida en el mercado. En 1981 apareció la tarjeta CGA, (Color Graphics Array) o matriz de gráficos en color. esta tarjeta, pese a disponer de pocos modos gráficos en baja resolución (como 320x200x4 o 640x200x2) y con tan sólo 4 colores, supuso el cambio de la PC hacia su estado actual.

CAPÍTULO 2

PROGRAMACIÓN DE LA TARJETA CGA

En este Capítulo abordaremos:

- ***Que es la pantalla real y la pantalla virtual***
- ***Almacenamiento y lectura de archivos de imágenes en disco***
- ***Animación con la tarjeta CGA***
- ***El uso del modo 6 de "alta resolución" de la tarjeta CGA***

2.1 Inicialización y escritura de un punto

Son raros los lenguajes de programación que nos permiten la escritura de un punto en la pantalla en el modo CGA. Por ejemplo, Turbo Pascal tiene PUTPIXEL. Estos procedimientos escritos en ensamblador, suelen estar optimizados, por lo que aquello que escriba en Turbo Pascal, generalmente, no se ejecutará con mayor rapidez. En los modos gráficos CGA, tan solo contamos con una pantalla gráfica. Para realizar animaciones, la única solución es crear mediante un puntero, una zona de memoria conocida como ***"pantalla virtual"*** en la que el programa modificará los bits según la imágen deseada y que se puede visualizar cuando esta lista.

El siguiente procedimiento es capaz de escribir un punto en la pantalla. En modo CGA, el tampón de visualización está situado en la dirección $B800:0. Las líneas m pares comienzan en la dirección $BA00:0

```
PROCEDURE punto_CGA(x,y:integer;color:byte);
var
        mascara:byte;
function fondo BYTE;
begin
        fondo:= 2*(3-x mod 4);
end;
begin
        color:= color SHL fondo;
        mascara := 3 SHL fondo;
        if y mod 2=0 then
          mem($b800:80*(y div 2) + x div 4):= (mem($B800:80*(y div2) + x div 4)
            and (not mascara)) or color
        else
          mem($BA00:80 * (y div 2) + x div 4) := (mem($BA00:80 * (y div 2) + x
          div 4) and (not mascara)) or color
end;
```

El objetivo consiste en no modificar más que un pixel, aunque la memoria este repleta de bytes. En modo CGA, cuatro pixeles se codifican en un byte.
La primera instrucción se encarga de confeccionar la máscara de bits correspondiente a la posición y al color del pixel que a de meterse a la memoria. La segunda, crea la máscara que permite poner con anterioridad a cero el contenido de la memoria en el punto de visualización.

Si no se realizace así, obtendría una suma de bit a bit, obteniendo un color no deseado, excepto cuando vemos en fondo negro donde todos los bits están en cero.
A continuación se lleva a cabo una prueba para indicar el byte correcto a modificar, dependiendo de si es una línea par o impar. Al terminar se ponen en cero dos bits mediante una compuerta lógica (and) con el complemento a 1 y metemos dos bits del pixel con una compuerta lógica (or).

Este procedimiento se ejecuta más rápido con un poco de más trabajo. Existe la necesidad de reescribir dos pequeños procedimientos útiles. Ya fueron vistos en el capítulo anterior, solo que hay que modificarlos para tener en cuenta los valores nuevos de dirección y el tamaño de la RAM de video.

```
PROCEDURE color_pantalla(color:byte);
begin
  FILLCHAR(mem($B800:0),16384,color);
end;
```

Si coloreamos la pantalla en blanco, todos los bits puestos a 1 deben tomar el valor FFh 0 255 en decimal. La tabla –6- muestra los bytes que hay que cargar a memoria para iluminarla con otros colores:

Byte	Color de la paleta 0	Color de la paleta 1
$00	Negro	negro
$55	Verde	cyan
$AA	Rojo	magenta
$FF	amarillo	blanco

Tabla 6. Colores para las paletas del CGA

Para no tener que recordar esta tabla, hay que reescribir el procedimiento para que se escoja el color simplemente como un valor comprendido entre 0 y 3, pasando como constante.

```
PROCEDURE color_pantalla(color:byte);
const
      tabla_colores:array(0..3) of byte
                  =($00,$55,$AA,$FF);
begin
      FILLCHAR(mem($B800:0),16384,tabla_colores(color));
end;
```

Y para cerrar los dos procedimientos se escribe:

```
PROCEDURE borrar_pantalla;
begin
      color_pantalla(0);
end;
```


2.2 Pantalla real y pantalla virtual

PANTALLAS VIRTUALES

Una Pantalla Virtual (en inglés Virtual Screen o VScreen) es un buffer de memoria el cual tratamos igual que si fuera la VideoRAM, escribiendo en esta memoria como si lo hiciéramos en videomemoria. Esto quiere decir que de la misma forma que escribimos bytes en la VRAM podemos escribirlos en nuestra pantalla virtual sabiendo que no van a aparecer en pantalla, pero que están almacenados ahí, en ese buffer de RAM convencional.

Cuál es su utilidad? La respuesta es muy sencilla. Al dibujar directamente en pantalla el usuario va viendo (aunque en un tiempo mínimo) como construimos la pantalla, un sprite tras otro, un bitmap tras otro. Esto ya de por sí es un efecto desagradable, pero si además al mover un sprite lo borramos para dibujar el siguiente fotograma de la animación, durante un tiempo también mínimo no habrá nada en pantalla (entre el

borrado y el dibujado), de tal forma que el Sprite parecerá parpadear (por la repetición del proceso <sprite, fondo, sprite, fondo...>).

Para evitar estos 2 problemas, se dibuja en una pantalla virtual, con lo que el usuario ya no puede ver el proceso de construcción de la pantalla, se espera a un retrazo vertical y se "vuelca" el contenido de la VScreen a la RAM de Video. El proceso de volcado consiste en copiar cada punto de la pantalla virtual en el punto que le corresponde de la VRAM.

Para crear una zona que reproduzca exactamente el contenido de la pantalla real:

1. Se puede construir las imágenes fuera de la vista del usuario para visualizarlas casi simultáneamente en la pantalla mediante la transferencia del bloque de bytes gracias al un procedimiento MOVE.
2. Permite guardar una imágen de pantalla para restaurarla a voluntad. La creación de un puntero para direccionar una zona de memoria de tamaño idéntico al de pantalla se realiza mediante rutinas muy parecidas al del capítulo anterior. Solo cambia el tamaño de los bloques, que ahora son de 16,384 bytes, y la dirección de la pantalla, el código seria:

```
type
      pantalla_ptr=^tipo_pantalla;
      tipo_pantalla=array(0..16384) of byte;
var
      salva_pantalla_ptr:pantalla_ptr;
      pantalla:tipo_pantalla OABSOLUTE $B8000:$0000;
PROCEDURE salva_pantalla;
begin
      move(pantalla,salva_pantall_ptr^,16384);
end;
PROCEDURE restaura_pantalla
begin
      MOVE(salva_pantalla_ptr^,pantalla,16384);
end;
```

Para escribir un pixel en la pantalla virtual, el código es:

```
PROCEDURE punto_virtual(x,y:integer;color:byte);
var
 mascara:byte;
 function
 begin
  fondo:= 2*(3-x mod 4);
 end;
begin
 color:= color SHL fondo;
 mascara := 3 SHL fondo;
 if y mod 2 = 0 then
  salva_pantalla_ptr^(80*(y div 2) + x div 4):=
  (salva_pantalla_ptr^(80*(y div 2) + x div4) and (not mascara)) or color
 else
  salva_pantalla_ptr^(80*(y div 2) + x div 4):=
  (salva_pantalla_ptr^(80*(y div 2) + x div 4) and (not mascara)) or color;
end;
```


8192 bytes son 2000h, la diferencia entre las direcciones de visualización de las líneas pares e impares de la pantalla (real o virtual). ***De este modo se puede dibujar en la pantalla virtual.***

2.3 Almacenamiento de imágenes en disco, lectura de archivos de imágenes

Salvar el contenido de la pantalla en disco se escribe en forma idéntica al del modo 19 de la tarjeta VGA que posteriormente se aborda, pero sólo en apariencia; porque *tipo_pantalla* es declarado así:

```
type
    typo_pantalla = array(0..16383) of byte;

PROCEDURE grabar_pantalla_en_disco(nombre_archivo:string);
var
```

```
        archivo:file of tipo_pantalla;
begin
        assing(archivo,nombre_archivo);
        rewrite(archivo);
        write(archivo,pantalla);
        close(archivo);
end;

PROCEDURE leer_pantalla_de_disco(nombre_archivo:string);
var
        FICHERO:FILE OF TIPO_PANTALLA;
begin
        assing(archivo,nombre_archivo);
        reset(archivo);
        read(archivo,pantalla);
        close(archivo);
end;
```

2.4 Animación con la tarjeta CGA

La animación es la creación de la ilusión de movimiento al visionar una sucesión de imágenes fijas generadas por computadora. Antes de la llegada de las computadoras, la animación se realizaba filmando secuencias dibujadas o pintadas manualmente sobre plástico o papel, denominados celuloides, un fotograma cada vez. Al principio, las computadoras se utilizaron para controlar los movimientos de la obra artística y simular la cámara.

La animación puede utilizarse para crear efectos especiales y para simular imágenes imposibles de generar con otras técnicas. La animación también puede generar imágenes para datos científicos, y se ha utilizado para visualizar grandes cantidades de datos en el estudio de las interacciones de sistemas complejos, como la dinámica de fluidos, las colisiones de partículas y el desarrollo de tormentas. Estos modelos de base matemática utilizan la animación para ayudar a los investigadores a

visualizar relaciones. La animación informática es usada en casos judiciales para la reconstrucción de accidentes.

Cómo funciona la animación

En la animación tradicional de fotograma a fotograma, la ilusión de movimiento se crea filmando una secuencia de celuloides pintados a mano y, a continuación, proyectando las imágenes a mayor velocidad, por lo general de 14 a 30 fotogramas por segundo. En animación, las ilustraciones se crean mediante programas informáticos, fotograma a fotograma y, a continuación, se modifican y se reproducen.

Otra técnica infográfica es la animación en tiempo real, en la que los fotogramas son creados por la computadora y se proyectan inmediatamente en la pantalla de la computadora. Esta técnica elimina la fase intermedia de digitalización de las imágenes. En la actualidad la animación en tiempo real no es capaz de producir resultados de alta calidad o con gran riqueza de detalles. Es más adecuada para la creación de animaciones simples y de juegos de computadora.

Animación asistida por computadora

En el proceso de animación tradicional, primero se dibuja una ilustración del argumento, escena por escena, se elabora la pista de sonido y un animador especializado crea los fotogramas animados. Más tarde, otros animadores dibujarán los fotogramas entre una posición clave y otra, agregarán color y, por último, se filmarán todos los fotogramas. Las computadoras pueden utilizarse como auxiliar o sustituto de cada fase de este proceso de animación.

Intercalación

El proceso de creación de los fotogramas intermedios para rellenar la acción entre dos posiciones clave se denomina intercalación (en inglés, in-betweening). Se han desarrollado técnicas que permiten que la computadora cree estos fotogramas mediante el cálculo de los puntos comunes entre un fotograma clave y otro. En el caso

más sencillo, la computadora dibuja el movimiento intermedio de dos puntos correspondientes calculando la distancia al punto medio. La repetición de cálculos del punto medio puede generar la ilusión de un movimiento fluido y continuo.

Sistemas de pintado

El pintado a mano de los celuloides animados es un proceso laborioso: un ilustrador experimentado alcanza una producción media de 25 celuloides por día. En ocasiones, las celuloides se apilan para crear diferentes imágenes: por ejemplo, los celuloides pueden interactuar entre sí, superponerse o servir como fondos de otras imágenes. Cuando se apila un gran número de celuloides, las capas transparentes se hacen ligeramente opacas. Por consiguiente, el ilustrador debe compensar este efecto variando los colores de la imagen, proceso que a menudo provoca errores. Las computadoras pueden eliminar estos errores e incrementar la productividad coloreando de manera uniforme las áreas más complejas de los fotogramas. El pintado por computadora utiliza un proceso de coloreado, o rellenado, en el que el artista especifica un color y, a continuación, selecciona un píxel. A renglón seguido, la computadora cambia todos los píxeles adyacentes que tienen el mismo color (o aproximadamente el mismo color) por el nuevo color especificado.

Filmadoras y montaje

Una vez pintados los fotogramas, es necesario filmarlos. Tradicionalmente, un expositor de animación sitúa los celuloides y la filmadora de manera que las capas de celuloides y la filmadora puedan moverse de forma independiente. La computadoras simula el expositor de animación y la filmadora. Controla esta filmadora virtual en un espacio tridimensional, mientras enfoca las series de imágenes bidimensionales, los celuloides, en su memoria. En realidad, tanto los celuloides como la filmadora residen dentro de la computadora. Con la filmadora virtual es posible simular características especiales de las filmadoras reales, por ejemplo las lentes gran angulares y las reflexiones de lente. Esta capacidad de controlar una filmadora virtual, combinada con

las potentes herramientas de montaje del vídeo digital, permiten al animador completar la película totalmente dentro del entorno generado por la computadora.

Animación modelada por computadora

La animación modelada por computadora es el proceso de crear modelos tridimensionales de objetos animados. Por lo general, esto se consigue representando los objetos mediante los siguientes métodos: mallas de alambre, caras o facetas y sólidos.

Las representaciones en malla de alambre se especifican mediante un conjunto de segmentos de línea, por lo general los bordes del objeto y un conjunto de puntos sobre la superficie denominados vértices. Aunque una representación de malla de alambre no suele generar imágenes muy realistas, es muy práctica para estudios rápidos, por ejemplo cuál será el movimiento de un objeto y su adecuación a determinada escena. Las representaciones de caras se especifican mediante un conjunto de características primitivas, por ejemplo un grupo de polígonos, para generar curvas y caras uniformes. Aunque es posible modelar perfectamente la superficie del objeto como un conjunto de características primitivas, puede no ser práctico medir y almacenar estas características ya que los objetos complejos pueden requerir un número infinitos de características para generar una superficie uniforme. Las representaciones de sólidos se especifican mediante un conjunto de formas primitivas, o partes de formas primitivas. Por ejemplo, un ser humano puede representarse mediante una esfera para la cabeza y cubos para componer el tronco y las extremidades. Las representaciones sólidas pueden especificarlas superficies internas y externas de un objeto.

El proceso de creación de una escena tridimensional real se denomina digitalización. Se proporciona a la computadora una descripción detallada de los objetos que comprenden la escena, junto con las especificaciones de la filmadora. Para crear imágenes de calidad fotográfica, la computadora debe calcular la perspectiva de los visionadores de la imagen, los objetos y superficies visibles; agregar sombreado

mediante la determinación de la iluminación disponible determinando la luz disponible para cada superficie; agregar reflexiones y sombras; incorporar texturas, patrones y rugosidad a las superficies para que los objetos tengan un aspecto más real; incorporar transparencia a los objetos; y eliminar las superficies ocultadas por otros objetos .

Una vez digitalizados los objetos y la iluminación en una escena tridimensional, el animador especifica sus movimientos dentro de la escena, así como los movimientos de la filmadora. Las posiciones clave sincronizan el movimiento de los objetos, al igual que en el modelo asistido por computadora, y deben crearse los fotogramas intermedios. Una técnica, denominada animación paramétrica de posiciones clave, interpola (o combina) las imágenes intermedias. Otra técnica, llamada **animación algoritmica**, controla el movimiento mediante la aplicación de reglas que determinan cómo deben moverse los objetos. Cuando se han especificado los objetos y su comportamiento, la filmadora virtual digitaliza cada escena fotograma a fotograma. Por último, se reproduce la secuencia animada final.

A pesar de la potencia de las computadoras actuales y de las innovaciones utilizadas para acelerar los procesos de animación tradicionales, las animaciones modernas requieren computadoras aún más rápidas y potentes para aprovechar las nuevas técnicas y efectos potencialmente fotorrealistas. En el largometraje animado de Disney "Juguetes" (Toy Story, 1995), los estudios de animación PIXAR emplearon una media de 3 horas en calcular cada fotograma, y algunos requirieron hasta 24 horas. Para esta película de 77 minutos, se generaron 110.880 fotogramas. Se emplearon técnicas de computación distribuida; una sola estación de trabajo hubiera tardado 38 años.

Con la tarjeta CGA...

El modo CGA, por su número limitado de colores utilizable presenta una desventaja respecto a los otros. Por otra parte, presenta como ventaja que las imágenes ocupan un tamaño mínimo. De este modo se tiene la facultad de crear un

mayor número de punteros de tipo "pantalla virtual" dentro de un espacio. ***Esta ventaja puede utilizarse para programar dibujos animados (normalmente cíclicos) con cuatro veces más imágenes que en el modo VGA.***

El programa de animación del OVNICGA (incluido en el CD) ha sido codificado para este modo y su código completo se muestra a continuación:

```
USES
 crt,dos,iniciar;
TYPE
 pantalla_ptr  = ^tipo_pantalla;
 tipo_pantalla = ARRAY[0..16383] of BYTE;
VAR
 pantalla  : tipo_pantalla ABSOLUTE $B800:0;
 archivo   : FILE of tipo_pantalla;
 res       : REGISTERS;
 sonido    : INTEGER;
 i         : BYTE;
 salva_pantalla_ptr,dibujo_ptr,pantalla_virtual: pantalla_ptr;

PROCEDURE escribir_punto(x,y,color : INTEGER);
BEGIN
 WITH res DO
  BEGIN
   AH := $0C;
   AL := color;
   CX := x;
   DX := y;
  END;
 INTR($10,res);
END;

PROCEDURE mostar_dibujo(x,y,numero : BYTE);
TYPE
 dibujo  = ARRAY[0..7] of BYTE;
CONST
 alto  : dibujo = (13,14,11,10,10,08,10,19);
```

```
  largo : dibujo = (20,20,23,25,23,25,26,30);
 VAR
  linea : BYTE;
 BEGIN
  FOR linea := 0 TO alto[numero] DO
   BEGIN
    MOVE(dibujo_ptr^[80*(25*(numero DIV 2)+linea)+40*(numero MOD 2)],
       pantalla_virtual^[80*(y+linea)+x],largo[numero]);
    MOVE(dibujo_ptr^[80*(25*(numero DIV 2)+linea)+8192+40*(numero MOD 2)],
       pantalla_virtual^[80*(y+linea)+x+8192],largo[numero]);
   END;
 END;

 PROCEDURE restaurar_pantalla_virtual(zona : BYTE);
 BEGIN
  MOVE(salva_pantalla_ptr^[80*zona],pantalla_virtual^[80*zona],2000);
  MOVE(salva_pantalla_ptr^[80*zona+8192],pantalla_virtual^[80*zona+8192],2000);
 END;

 PROCEDURE salvar_pantalla;
 BEGIN
  MOVE(pantalla,salva_pantalla_ptr^,16384);
 END;

 PROCEDURE mostrar_pantalla_virtual(zona : BYTE);
 BEGIN
  MOVE(pantalla_virtual^[80*zona],pantalla[80*zona],2000);
  MOVE(pantalla_virtual^[80*zona+8192],pantalla[80*zona+8192],2000);
 END;


 PROCEDURE vent;
 VAR
  i : INTEGER;
 BEGIN
  FOR i := 1 TO 500 DO
```

```
    SOUND(sonido + 16*RANDOM(255));
   IF sonido > 500 THEN
     sonido := sonido - 500;
   SOUND(24);
  END;

  PROCEDURE mostar_imagen(nombre,x,y,numero : BYTE; dx,dy : INTEGER);
  VAR
   i,
   j : BYTE;
  BEGIN
   FOR i := 1 TO nombre DO
     BEGIN
      j := y+dy*i;
      restaurar_pantalla_virtual(j);
      mostar_dibujo(x+dx*i,j,numero);
      vent;
      mostrar_pantalla_virtual(j);
     END;
  END;

  **** Programa principal ****
  BEGIN
   inicializar_modo(4);
   NEW(salva_pantalla_ptr);
   NEW(dibujo_ptr);
   NEW(pantalla_virtual);
   ASSIGN(archivo,'PLATILLO.CGA');
   RESET(archivo);
   READ(archivo,dibujo_ptr^);
   CLOSE(archivo);
   FOR i := 0 TO 175 DO
    escribir_punto(2*RANDOM(160),RANDOM(200),1);
  salvar_pantalla;
  sonido := 13000;
  mostar_imagen(10,58,70,0,-2,0);
```

```
mostar_imagen(4,38,70,1,-2,0);
mostar_imagen(4,30,70,2,-2,0);
mostar_imagen(4,22,70,3,-2,0);
mostar_imagen(4,18,70,4,-1,-2);
mostar_imagen(4,18,60,5,1,-2);
mostar_imagen(4,22,50,6,2,-2);
mostar_imagen(5,30,40,7,4,-4);
DELAY(100);
NOSOUND;
inicializar_modo(3);
END.
```

Las diferencias con el modo VGA son:

1. **La no inclusión de la rutina de descompactación**
2. **El tratamiento por separado de las líneas pares de las impares**
3. **El cambio de la pantalla completa, y**
4. **La adaptación de las coordenadas de visualización a las nuevas medidas de la pantalla.**

Turbo Pascal y C++ cuenta con dos procedimientos que sirven para la animación de dibujos: ***GETIMAGE y PUTIMAGE.*** Las definiciones de los modos CGA están contenidas en la unidad GRAPH. Podemos utilizarlos respetando las declaraciones de los manuales. GETIMAGE guarda una porción de la pantalla y PUTIMAGE la restituye en el lugar deseado. Estos son los procedimientos en versión CGA donde comprobaremos que la ejecución es muy rápida:

```
Procedure salva_imagen(x1,y1,x2,y2:byte);
var
  linea:byte;
begin
  for i := 0 to y2-y1 do
  begin
    MOVE[pantalla[(y1+linea)*80+x1],salva_pantalla_ptr^[linea*80],x2-x1);

MOVE(pantalla[(y1+linea)*80+x1+8192],salva_pantalla_ptr^[linea*80+8192],x2-x1+1);
```

```
    end;
  end;
  procedure restaura imagen(x,y,longitud,altura:byte);
  var
    linea:byte;
  begin
    for linea:= 0 to altura do
    begin
      MOVE(salva_pantalla_ptr^[linea*80],pantalla[(y+linea)*80],longitud);

MOVE(salva_pantalla_ptr^[linea*80+8192],pantalla[(y+linea)*80+x+longitud);
    end;
  end;
```

Como anteriormente no se realiza un prueba para verifcar que x2 y y2 son menores que x1 y y1, el programa deberia hacerlo siguiendo el siguiente criterio:

longitud = x2-x1+1

Altura = y2-y1+1

Una gran difrencia con los procedimientos escritos para el modo 19 de VGA es que la longitud x1 y x2 son bytes y por tanto, en CGA, multiplos de 4 pixeles Altura, y1 y y2 corresponden a las líneas pares e impares que componen la pantalla. LA IMAGEN SE GUARDA DE ARRIBA A LA IZQUIERDA DE LA PANTALLA VIRTUAL, por lo que su tamaño puede ser cualquiera y es aquí donde utilizamos la pantalla virtual; destinada normalmente a recibir la pantalla ya almacenada. Pero, podemos direccionar cualquier otra variable tipo puntero del mismo tipo. Pueden almacenarse varias imágenes en la pantalla virtual al mismo tiempo, siempre y cuando no sean demasiado grandes. El ejemplo perfecto para describir lo anterior se encuentra en el listado y programa llamado "DUPIMCGA,PAS" que se incluye en el CD complementario de este trabajo.

Pero una revolución llegó con la aparición de la tarjeta VGA (Video Graphics Array) Matriz Gráfica de Video, ya que incorporaba todos los modos de video soportados por las tarjetas anteriores (CGA, MCGA y EGA), añadiendo el modo 640x480 a 256 colores, y funcionando con cualquier tipo de monitor (analógicos, multifrecuencia, estándar, etc...), cosa que no hacían las tarjetas anteriores. La programación de esta tarjeta se aborda en el capítulo siguiente, es el más largo de los cinco y el más importante.

CAPÍTULO 3

PROGRAMACIÓN EN MODO 19 DE LA TARJETA VGA/256 COLORES

En este Capítulo abordaremos:

- ***Las características más importantes de la tarjeta VGA/256 colores***
- ***Desde como escribir un punto gráfico y su manipulación hasta el tratamiento de imágenes.***
- ***El principio básico de las animaciones con imágenes, guardado lectura de información gráfica.***
- ***Programas completos donde se aplican cada concepto estudiado.***
- ***Animación en modo 19 de esta tarjeta gráfica***

3.1 Almacenamiento de la imagen captada

Ahora que se dispone de los medios para la realización de dibujos multicolores, se utiliza la función RANDOM de Turbo Pascal para rellenar la pantalla de puntos:

```
uses
  crt;
begin
  inicializar_modo($13);
  repeat
    punto(2*random(160),random(200),random(256));
  until keypressed;
end.
```

y es suficiente para terminar el programa con solo oprimir una tecla. Puede resultar desagradable presentar una aplicación en la que el dibujo aparezca progresivamente, esto denotaría nuestra falta de experiencia en la programación, excepto cuando deseamos un efecto especial, es siempre preferible la visualización de una imagen terminada. Por lo tanto, hace falta:

1. Crear un medio para que nuestra imagen este fuera de la pantalla en tiempo real y verla una vez terminada.
2. Guardar nuestra imagen de pantalla creada para ocuparla posteriormente.

Definamos un tipo de "pantalla" que será una tabla del mismo tamaño de la pantalla, esto es 200 líneas x 320 pixeles = 64,000 bytes y un tipo "puntero a pantalla virtual"

```
type pantalla_ptr = ^tipo pantalla;
```

En el caso (1), la escritura de un punto en la pantalla virtual se realizará mediante el programa:

```
uses
  crt,iniciar;
type
  pantalla_ptr=^tipo_pantalla;
  tipo_pantalla=array(0..63999) of byte;
var
  salva_pantalla_ptr:pantalla_ptr;
  pantalla:tipo_pantalla ABSOLUTE $A000:$0000;
```

```
begin
  NEW(salva_pantalla_ptr);
  inicializar_modo($13);
  repeat
salva_pantalla_ptr^(320*random(200)+2*random(160)):=random(256);
  until keypressed;
  move(salva_pantalla_ptr^,pantalla,64000);
end.
```

escribirse como sigue:

```
PROCEDURE punto(x,y:word;color:byte);
begin
  pantalla(320*y+x):=color;
end;
```

Para el caso 2, es muy similar y de lógica exactamente la inversa del procedimiento para restaurar la imágen y así se codificaria:

```
PROCEDURE restaura_pantalla;
begin
  move(salva_pantalla_ptr^,pantalla,64000);
end;
```

Con lo anterior, los procedimientos anteriores serian:

```
uses
  crt,iniciar;
type
  pantalla_ptr=^tipo_pantalla;
  tipo_pantalla=array(0..63999) of byte;
var
  salva_pantalla_ptr:pantalla_ptr;
  pantalla:tipo_pantalla ABSOLUTE $A000:$0000;
begin
  NEW(salva_pantalla_ptr);
  inicializar_modo($13);
```

```
 repeat
  punto(2*random(160),random(200),random(256));
 until keypressed;
 salva_pantalla;
end.
```

Para asegurar que todo está bien, hay que borrar la pantalla y recuperar la imágen almacenada. Para borrar la pantalla, basta con llamar el siguiente procedimiento:

```
PROCEDURE color_pantalla(color:byte);
begin
 fillchar(pantalla,64000,color);
end;

PROCEDURE borrar_pantalla;
begin
 color_pantalla(0);
end;
```

borrar_pantalla iluminará toda la pantalla de color negro. El procedimiento *color_pantalla*, es más general y permite colorearla con un color especial. El programa final sería:

```
 uses
  crt,iniciar;
type
  pantalla_ptr=^tipo_pantalla;
  tipo_pantalla=array(0..63999) of byte;
var
  salva_pantalla_ptr:pantalla_ptr;
  pantalla:tipo_pantalla ABSOLUTE $A000:$0000;
begin
  NEW(salva_pantalla_ptr);
  inicializar_modo($13);
```

```
repeat
 punto(2*random(160),random(200),random(256));
until keypressed;
salva_pantalla;
borrar_pantalla;
DELAY(1000);
restaura_pantalla;
end.
```

No hay nada de complicado en lo anterior y por tanto ya se tienen las bases para realizar la animación.

3.2 Almacenamiento de imágenes en disco, y lectura de archivos de imágenes

Para guardar del contenido en la pantalla en disco se escribe un procedimiento muy simple:

```
PROCEDURE grabar_pantalla_en_disco(archivo:string);
var
  archivo:FILE OF tipo_pantalla;
begin
  ASSING(archivo,archivo);
  REWRITE(archivo);
  write(archivo,pantalla);
  close(archivo);
end;
```

La lectura del archivo así creado y representado por "nombre_archivo", se realiza mediante el procedimiento:

```
PROCEDURE leer_pantalla_de_disco(nombre_archivo:string);
var
  archivo:FILE OF tipo_pantalla;
begin
  ASSING(archivo,nombre_archivo);
  RESET(archivo);
  read(archivo,pantalla);
  close(archivo);
end;
```

Hay que notar lo rápido del reguardado y lectura de la imagen mejorada por la lectura del archivo, que solo contiene un elemento.

3.3 Métodos de compactación

La imagen es un mapa de bits, y ahora trataremos de almacenarla en el disco con un determinado formato para que el tamaño del archivo resultante sea menor o igual que la imagen real.

Se van a explicar dos métodos de compresión para almacenar los formatos gráficos en disco. El primero de ellos es el RAW o formato crudo, y el otro método es la compresión RLE (del inglés Run Lenght Encode) que es el utilizado en los archivos PCX.

Ahora utilizaremos imágenes que pueden ser generadas desde un programa de edición que soporte este tipo de compresión. Con esto se consigue que los programas tengan unos gráficos más vistosos y presentables, e incluso se puede utilizar programas de diseño de imágenes 3D y utilizar la imagen resultante.

Lo que se pretende es generar la manera de obtener archivos binarios en nuestro disco duro o diskette conteniendo el mapa de bits de una imagen para una posterior utilización en cualquier programa mediante su apertura, carga y volcado en VideoRAM. Para esto podemos hacer 2 cosas:

- Bien volcar el bitmap directamente en el archivo tal y como está en videomemoria
- Comprimir este mapa de bits con algún método de compresión para reducir este espacio en disco, ya sea el método LZW (archivos GIF), el RLE (archivos PCX), compresión fractal, etc.

Vamos a ver el más sencillo, el Run Length Encoding o RLE de los archivos PCX de ZSoft.

También podemos utilizar la imagen comprimida para almacenar los sprites que se utilicen en los programas. Después sólo habrá que descomprimir la imagen en una zona de memoria, y sabiendo el desplazamiento de cada sprite se pueden hacer

animaciones, etc. De esta forma los dibujos no hace falta que estén en el ejecutable (en forma de arrays, etc.).

Explico el primero de los dos formatos, el formato crudo o sin compresión RAW.

EL FORMATO CRUDO

Básicamente el formato se compone de una cabecera, la paleta y el cuerpo o bitmap de la imagen, aunque esta estructura no es estándar. Cada usuario puede hacer variantes cambiando los datos de la cabecera de la forma que más le favorezca para su propósito, o incluso puede prescindir de ella si se conocen de antemano los datos de la imagen. Sin embargo no esta de más incluir al menos el tamaño de ésta, de forma que podremos utilizar imágenes de distintos tamaños con un pequeño cambio en el código, puesto que sólo se verán modificados los bucles que se dediquen a dibujar en la pantalla.

Las imágenes que dibujaremos pueden ser tan grandes como la pantalla o incluso más grandes que ésta. Para almacenarlas podemos utilizar la linealidad de la memoria de video. La memoria está organizada por bancos consecutivos, Por ejemplo, en 13h, sólo necesitamos utilizar el primer banco, porque utilizamos tan sólo 64.000 bytes (320x200 bytes). Si habláramos de modos SVGA con resoluciones como por ejemplo 800x600, necesitaríamos 480.000, y como la memoria está dividida en segmentos de 65.536 bytes la tarjeta las maneja por medio de los bancos. Se pueden almacenar en el archivo los datos de cada banco uno tras otro.

Sin embargo, sólo será necesario cambiar de bancos cuando el tamaño de la imagen sobrepase los primeros 65536 bytes del primer banco. Si el tamaño de la imagen es inferior no tendremos que cambiarlo porque todos los datos de la imagen están en él.

En cuanto al cuerpo de la imagen, este formato es el más sencillo de todos porque no se utiliza ningún método de compresión para la imagen, simplemente se almacena en el disco de manera lineal. Es decir, se almacenan de forma consecutiva

los datos de la imagen y de la paleta en el archivo, y opcionalmente la cabecera, que normalmente se coloca al principio. Con este método se puede saber el tamaño del archivo crudo resultante, sabiendo el ancho y el alto de la imagen, el número de colores y la longitud de la cabecera y la paleta. Los archivos RAW (a veces también llamados SCR, CLP o similar) se basan pues simplemente en el almacenamiento de los mapas de bits en archivos binarios.

EL FORMATO PCX

EL formato de archivos con extensión .PCX contiene uno de los métodos de compresión más sencillos, el Run Length Encoding o RLE, basado en la repetición de datos consecutivos. En primer lugar veremos el formato de la cabecera, puesto que tendremos que leerla e interpretar sus datos para visualizar la imagen correctamente. La cabecera de un PCX se compone de los campos que aparecen en la tabla –7- siguiente:

Campo	Tamaño	Descripcion
Manufacturer	byte identificación.	10 = Zsoft .pcx;
Versión	Byte	Información sobre la versión 0 = Version 2.5 del PC Paintbrush 2 = Version 2.8 3 = Version 2.8 4 = PC Paintbrush para Windows 5 = Version 3.0 y > de Paintbrush y PC Paintbrush +, Archivos 24-bit.
Encoding	Byte	1 = .PCX Run Lenght Encoding"
BitsPerPixel	byte	Nº de bits que representan 1 pixel. (por Plane) - 1, 2, 4, or 8"
Window [4]	word	Dimensiones: Xmin,Ymin,Xmax,Ymax"
Hdpi	word	Resolucion Horizontal en DPI
Vdpi	word	Resolucion Vertical en DPI
Colormap [48]	byte	Paleta para modos de 16 colores
Reserved	byte	Reservado. Debe ser 0
Nplanes	byte	Número de planos de la imagen
BytesPerLine	word	Numero de bytes para representar una línea. Debe ser un número par. NO calcular como Xmax-Xmin
PaletteInfo	word	Cómo interpretar la paleta - 1 = Color/BW, 2 = Grayscale
HscreenSize	word	Tamaño horiz. de la pantalla en pixels
VscreenSize	word	Tamaño vertical de la pantalla en pixels
filler [54]	byte	Espacios en blanco para rellenar hasta los 128 bytes de la cabecera

Tabla 7. Estructura de la cabecera del formato PCX

Para la descompresión los datos que más interesa saber son la ventana, con la que se averigua el tamaño de la imagen, el campo bits_por_pixel que indica la cantidad de colores de la imagen, y el campo bytes_por_linea que es la cantidad de bytes que hay por cada línea y plano del dibujo.

El campo ventana viene dada por cuatro enteros, que determinan la posición X e Y máxima y mínima, por lo que se puede averiguar el tamaño de la imagen simplemente restando las cantidades mínimas a las máximas (por ejemplo Xmax-Xmin). De esta forma podemos adaptar la resolución de la pantalla al tamaño de la imagen para visualizar la imagen entera.

Otro campo importante es el que nos indica la cantidad de colores que tiene la imagen, ya que habrá que leer este campo para adaptar el modo de la pantalla. Para esto se deben tener en cuenta este campo junto con el campo "Ventana". Los "bits_por_pixel" indican el número de bits que hacen falta para representar un pixel de pantalla.

El campo Bytes_por_linea indica la longitud en bytes de una línea horizontal. Esto es útil porque los archivos originarios de Zsoft están comprimidos línea a línea en vez de comprimir el bloque completo de la imagen; es decir, se coge un ScanLine del bitmap, se comprime o descomprime y se almacena; y así hasta el final de la imagen o archivo. De esta forma se evita que al descomprimir, si la imagen es más estrecha que la resolución de la pantalla, se descompriman los primeros colores de la línea siguiente a continuación de esta primera línea.

También es interesante antes de hacer ninguna operación el averiguar si el archivo es realmente un PCX. Para esto hay unos campos de identificación, como el primer byte que debe ser un 10 (0ah) y es el identificativo de todo PCX. Además también se puede saber con qué versión se ha creado este archivo, ya que hasta la versión 5 no se admitían imágenes de 24 bits por pixel (16 millones de colores). Si queremos pues que un programa de dibujo no pueda cargar archivos PCX que utilicemos en alguno de nuestros programas, o no queremos que sean reconocidos, podemos cambiar en el archivo el valor de este byte (cambiar el 0ah por otro valor) para que éste no pueda ser identificado y cargado.

Otro detalle importante es que la paleta en los PCX varía de lugar dependiendo del número de colores. En el caso de tener 16 colores la paleta se almacena en el

campo Colormap de la cabecera. Este campo tiene un tamaño de 48 bytes, correspondientes a los 16 colores y a las tripletas RGB de cada color (16*3=48).

Sin embargo, si la imagen es de 256 colores la paleta está en el final del archivo, después del cuerpo de la imagen y de un byte que indica si la paleta está al final. Este valor es el 12 decimal. Esto es debido a que la paleta tiene una longitud de 768 bytes (256 * 3 RGB) y no cabe en la cabecera, por lo que se decidió ponerla al final.

Las imágenes de más de 256 colores no tienen paleta, en cambio se almacena la imagen como si fuera de 8 bits por pixel y 3 planos, siendo cada plano la componente correspondiente (R, G y B).

LA DESCOMPRESION RLE

La compresión de la imagen en el PCX es el llamado RLE, de Run Lenght Encode (Codificación del tamaño durante la ejecución), en el que trata de comprimir la imagen codificando la cantidad de veces que se repite un mismo color.

De esta forma, si en la imagen original se encuentran 20 valores iguales y consecutivos, en el archivo comprimido apareceria un byte indicando el número de veces que se repite y después el color en cuestión. Es decir, en vez de almacenarse en el archivo 20 veces el color correspondiente, se almacenarían tan sólo 2 bytes: el 1° el indicador del número de repeticiones y el 2° el valor a repetir.

La diferencia entre el formato crudo o formato con compresión sería:

Crudo: 0,0,0,0,0,0,etc. (20 ceros)

RLE : 20,0

Por lo tanto se puede decir que en este tipo de compresión hay dos tipos diferenciados de valores: el que se debe colocar en la pantalla (el color) y el que indica compresión (llamado byte de Run, o RunByte).

En realidad hay un problema, y es que se debe hacer una distinción entre los bytes que tienen compresión de los que no. Para hacer la distinción entre un valor comprimido y otro sin comprimir, se activa los dos últimos bits del valor. Es decir, si el valor leido tiene activado estos dos bits, se trata de un valor comprimido, y los restantes seis bits indican el número de veces que se repetirá el byte siguiente. De esta forma el número máximo de repeticiones de un mismo valor es el 63 puesto que este es el número máximo de combinaciones que se pueden conseguir con seis bits.

Por tanto habrá que desactivarle estos dos últimos bits de mayor peso para que el valor resultante sea el indicado por los primeros seis bits, que es el número de veces que se ha de repetir el siguiente pixel. Con este sistema no se podrían representar en la pantalla ningún color superior al 191 porque tendrían estos bits activados y la descompresión lo tomaría como un byte del tipo comprimido.

La forma de salvar este problema es que si se quiere poner un valor superior al 191 se hará con dos bytes: el primero será de tipo comprimido y con el valor del contador a 1, y el segundo será el color que deseamos obtener en la pantalla, que puede ser superior al 191. Esto se puede traducir a pseudocódigo para comprobar como se realiza la descompresión:

```
  Read (archivo, valor1)
if valor1 AND 192 = 192
 then Begin
          read (archivo, valor2)
   for x=0 hasta (valor 1 AND 00111111b)
    poke (videoram, valor2)
 end else
 poke (videoram, valor1)
```


Sea cual sea el número de colores de la imagen la descompresión se realiza de la misma forma, puesto que se trata de una descompresión byte a byte. Después sólo hay que interpretar correctamente los valores obtenidos. Esto significa que si un pixel de la pantalla está compuesto por dos bytes (que es el caso de los 64K colores) se deberán de coger dos bytes resultantes para componer el pixel de la pantalla, esto se ilustra en la tabla –8-.

Comprimido	Descomprimido
10	(menor de 192 = descomprimido)10
193	200 (contador = 193-192 = 1)200
230 5	5 5 5 5 .... 37 veces
230 240	240 240 240 .... 37 veces

TABLA 8. CASOS POSIBLES DE DATOS EN EL ARCHIVO COMPRIMIDO

Un ejemplo de un programa en C se puede ver en el listado siguiente, hecho a partir del pseudocódigo anterior. Por simplicidad la rutina está adaptada para imágenes de 256 colores. Para las imágenes de 16 colores hace falta cambiar de planos para conseguir la imagen.

```
/* Bucle de la descompresión RLE:*/

 int a, x, y;
 dword pos;
 byte dato1, dato2;

  pos = 0;
  fseek (archivo, 128, SEEK_SET);
 /* Saltar cabecera */

 for (a=0;a<Ytotal;a++)
 {
  for (x=0;x<cabecera.BytesPerLine;)
  {
    fread (&amp;dato1, sizeof (dato1), 1, archivo);
    if (dato1 > 192 )
    {  /* Byte comprimido */
            fread (&amp;dato2, sizeof (dato2), 1, archivo);
            for (y=0;y<(dato1 - 192);y++,pos++, x++)
                PutPixel ( pos, dato2 );
    }
    else
    {
              PutPixel ( pos, dato1);
              pos++; x++;
    }
  }
  pos += ancho-cabecera.BytesPerLine ;
  /* Si el ancho de imagen es menor *que el ancho de pantalla, */
  /* se añade al offset el trozo que falta para completar el
    ancho de la pantalla */
}
```

El anterior pseudocódigo sólo descomprime un byte del archivo, sin embargo para descomprimir toda la imagen se deben seguir los pasos siguientes:

1) Leer la cabecera y tomar los datos más relevantes. Recordar que el ancho real de la imagen es el ancho leido de la cabecera + 1, y lo mismo ocurre con el alto.
2) Ir al final del archivo, retornar 769 bytes, y comprobar que el byte en esa posición es el 0Ch (12 decimal).
3) Leer los restantes 768 bytes del archivo, que son los valores correspondientes a la paleta.
4) Dividir todos los valores de la paleta por 4 para que entren en el rango de 0 - 63 del DAC.
5) Colocar la paleta en el DAC.
6) Colocarse el byte nº 129 del archivo. Este es el primer valor después de la cabecera.

7) Declarar las variables temporales que contendrán la posición X e Y actual. Sus valores iniciales serán las coordenadas superior izquierda especificados en la cabecera.
8) Leer un byte. Comprobar si los dos bits de más peso están activados. Se puede hacer de dos formas:
   (1) Hacer un AND con C0h (192 dec) y comprobar si el resultado es C0h (192 dec).
   (2) Comprobando si el byte es >= 192. Si NO es el caso, saltar al paso 11.
9) Se ha detectado compresión. Coger el valor de los restantes 6 bits, haciendo un AND con 3Fh (63 dec). El valor resultante es el contador.
10) Leer otro byte del archivo. Este es el valor del color de tantos pixels como lo indique el contador.
11) Colocarse en la posición de la pantalla indicada por las variables temporales y poner el pixel del color leido.
12) Incrementar la posición de las variables temporales, teniendo en cuenta el ancho y alto especificado en la cabecera. Si se ha llegado al ancho máximo, saltar al paso 14.

13)Si se está ejecutando un byte comprimido, decrementar el contador y volver al paso 11 hasta que contador = 0. Si no es un byte comprimido, o el contador está a cero, saltar al paso 8.

14)La imagen está dibujada en la pantalla. Sólo queda cerrar el archivo!

Veamos una función que realiza la descompresión, en el siguiente listado:

Descompresión RLE

```
void descomprime (FILE * archivo)
{
  int a, x, y;
  dword pos;
  char valor;
  word ancho=0;

  pos = 0;
  fseek (archivo, 768, SEEK_SET);

  /* aquí habria que leer y poner la paleta del PCX*/

  fread (&amp;valor, 1, 1, archivo);
  while (!feof(archivo))
  {
    fread (&amp;valor, 1, 1, archivo);
    if (ancho == cabecera.ancho)
      {
            pos += (ancho_pant - cabecera.ancho)-1;
            ancho = 0;
      } else ancho++;
    PutPixel (pos, valor);
    pos++;
  };
}
```


Hasta aquí hemos visto cómo descomprimir el formato PCX de 256 colores principalmente, pero para poder descomprimir imágenes SuperVGA es imprescindible cambiar de banco para poder visualizar la imagen completa.

LA COMPRESION RLE

El algoritmo de compresión es al revés del de descompresión. Se debe de comprimir la imagen línea a línea escribiendo en el archivo la cantidad de pixels iguales y consecutivos hay en la imagen.

Al igual que al descomprimir, se debe de distinguir los bytes que indiquen que se ha producido la compresión de los que no, y esto se hace activando los dos bits de más peso del primer byte (que es el que a la vez hace de contador) y escribiendo a continuación el byte del color que se repite en la imagen original. Además se debe de escribir en el archivo destino como comprimido cualquier color que sea superior al 192.

añadiendo un contador de 1. Es decir, un pixel de color mayor que el 192 se almacenaría como un RunByte de 1 y luego el color (2 bytes para 1 sólo pixel), de ahi que algunas imágenes comprimidas con RLE (aquellas que tienen muchos colores de valor mayor que 192) ocupen más que el equivalente crudo de la misma. El algoritmo es este:

```
Repite Y veces
 Repite X veces
 {
  - Contar el número de veces que se repite un color,
    máximo hasta 64 veces o límite de línea.
  - Si hay más de uno consecutivo, comprimir al archivo.
  - Si sólo es un color aislado
     - Si es inferior al nº 192
          escribir al archivo sin compresión
     - Si es igual o superior, escribir como comprimido (contador a 1)
 }
```

Sin embargo, una de las cosas más importantes de la compresión es generar la cabecera de forma correcta para que los programas comerciales o cualquier programa nuestro, posteriormente, reconozcan el formato e interpreten sus datos de manera correcta, y así poder trabajar con estas imágenes, aunque puede que no interese que los programas detecten el tipo de formato para evitar que las imágenes sean modificadas, y por tanto de forma indirecta nuestro programa. Un claro ejemplo son los juegos, que para que las imágenes creadas no se distribuyan con facilidad se comprimen con un formato que sólo los creadores conocen, por lo que sólo ellos saben las especificaciones de esta cabecera y, por tanto, como interpretar el bitmap. El código de la compresión RLE, a continuación se incluye:

```
Compresión RLE
contador = (cabecera.ancho+1);
contador *= (cabecera.alto+1);
bytecounter = 1;
```

```
offset = 0; ancho = 0;
b1 = GetPixel (offset);
offset++;

do {
 b2 = GetPixel (offset);
 offset++; ancho++;

  while ((b2 == b1) &&
        (bytecounter < 63) &&
        (ancho < cabecera.ancho))
  {
     bytecounter++;
     b2 = GetPixel (offset);
     offset++;ancho++;
  }
  if (ancho >= cabecera.ancho)
  {
    offset += ancho_pant-ancho;
    ancho = 0;
  }

  if (bytecounter != 1)
          {
           car = 192+bytecounter;
           fwrite (&car, 1, 1, fich);   // Escribe como comprimido
           fwrite (&b1, 1, 1, fich);    // Escribe color
          }

  if (bytecounter == 1)
          {
           if (b1 < 192) fputc (b1, fich);
           else
             {
             car = 193;       // 192 + 1 }
             fwrite(&car, 1, 1, fich);
             fwrite(&b1, 1, 1, fich);
             }
          }

bytecounter = 1;
b1 = b2;
} while (offset <= contador);   /* Ancho * alto */
```

PCX (RLE)VS RAW

Después de haber visto estos dos formatos para almacenar las imágenes en el disco, lo más conveniente es elegir en cada caso cuál es el que nos interesa más de los dos. Generalmente será el que el tamaño del archivo resultante sea menor. Sin embargo, **aunque el PCX es un formato que intenta que el tamaño en disco de las imágenes sea menor a veces no lo consigue, y el archivo tiende a ocupar más.**

Esto ocurre con las imágenes que sus valores no se repitan consecutivamente, y que además cada valor es superior a 192.

En el caso de imágenes tomadas con un escáner, los valores no se suelen repetir, por lo que normalmente la compresión RLE tiende a expandir el tamaño resultante. En estos casos es conveniente utilizar el formato RAW. Así que se debe saber cuál es el que conviene utilizar en cada caso. Por supuesto, **existen métodos de compresión mucho más complejos, basados en algoritmos de encadenado (o agrupación de subcadenas) como el LZW del formato GIF de Compuserve, o el algoritmo Inflate/Deflate usado en el PKZIP y en el nuevo formato gráfico estrella, el PNG, imágenes comprimidas con compresión fractal, etc.** La descripción de todos estos formatos gráficos requeriría una sección por sí misma, así que nosotros, tras conocer las bases del almacenamiento de archivos gráficos, se deben abordar temas relacionados con la programación elemental para animación, compactación y resguardo de archivos gráficos.

3.4 Desplazamiento y animación de imágenes

Para dar una primera idea acerca de lo que es una animación, conviene recordar los pequeños fantasmas que se pasean por la pantalla cuando jugamos PACKMAN (juego bastante viejo, pero muy divertido). Es por esto que un **sprite** corresponde al desplazamiento de un "objeto" por la pantalla. Un dibujo más sofisticado sería un personaje animado.

En un primer momento, hay que ver el principio de desplazamiento de un sprite por la pantalla. Por pantalla hay que entender no un fondo negro, sino un gran dibujo

cualquiera que este sea. La imagen se puede crear a través de un digitalizador o bien otro paquete, transformarla en un archivo de 64,000 bytes y cargarla como fondo para nuestra animación. En la práctica, esto se logra a través del procedimiento ***leer_pantalla_de_disco*** para ver la imagen en pantalla después de haber inicializado con el modo 19.

El siguiente programa hace aparecer a un personaje previamente elaborado en un programa gráfico. Un fragmento del programa es el siguiente y está incompleto:

```
TYPE
  TIPO_PANTALLA = ARRAY(0..63999) OF BYTE;
VAR
  pantalla: tipo_pantalla absolute $A000:$0000;
begin
  inicializar_modo($13);
  leer_imagen_de_disco('alberto.mcg');
end.
```

La cara que aparece, está dibujada con los 16 colores de la paleta EGA. El fondo es una degradación del gris (registros de color comprendidos entre 16 y 31).

Para hacer más interesante lo anterior, se lista un programa que llama un archivo grafico y se hace botar una pelota alrededor de la pantalla. Aquí se aplican muchos conocimientos que ya se describieron.

```
program Alberto;
uses
 crt,dos,iniciar;
type
 pantalla_ptr  = ^tipo_pantalla;
 tipo_pantalla = array[0..63999] of byte;
var
 pantalla : tipo_pantalla absolute $A000:$0000;
 salva_pantalla_ptr : pantalla_ptr;
 tocado : char;
 x,y,dx,dy,xp,yp : integer;
```

```
procedure leer_pantalla_de_disco(nombre_archivo : string);
var
 archivo : file of tipo_pantalla;
begin
 assign(archivo,nombre_archivo);
 reset(archivo);
 read(archivo,pantalla);
 close(archivo);
end;

procedure salva_pantalla;
begin
 move(pantalla,salva_pantalla_ptr^,64000);
end;

procedure pelota(x,y : word);
var
 j : word;
begin
 fillchar(pantalla[320*y + x + 2],4,10);
 fillchar(pantalla[320*(y+1) + x + 1],6,10);
 for j := 2 to 5 do
  fillchar(pantalla[320*(y+j) + x],8,10);
  fillchar(pantalla[320*(y+7) + x + 2],4,10);
  fillchar(pantalla[320*(y+6) + x + 1],6,10);
end;

procedure restaura_fondo(x,y : word);
var
 j : word;
begin
 for j := 0 to 7 do
  move(salva_pantalla_ptr^[320*(y+j) + x],
  pantalla[320*(y+j) + x],8);
end;
```

```
procedure escrita_teclado;
begin
 if keypressed then
  begin
   tocado:= readkey;
   if tocado = #0 then
    tocado := readkey;
    case tocado of
     '1',#79: begin
                dx := -1;
                dy := 1;
              end;
     '2',#80: begin
                dx := 0;
                dy := 1;
              end;
     '3',#81: begin
                dx := 1;
                dy := 1;
              end;
     '4',#75: begin
                dx := -1;
                dy := 0;
              end;
     '6',#77: begin
                dx := 1;
                dy := 0;
              end;
     '7',#71: begin
                dx := -1;
                dy := -1;
              end;
     '8',#72: begin
                dx := 0;
                dy := -1;
              end;
```

```
        '9';#73: begin
                   dx := 1;
                   dy := -1;
                 end;
       else    begin
                   dx := 0;
                   dy := 0;
                 end;
     end;
   end;
   if (x < 1) or (x > 310) then
     dx := - dx;
   if (y < 1) or (y > 191) then
     dy := - dy;
   x := x + dx;
   y := y + dy;
end;

                              ***** programa principal *****
begin
 inicializar_modo($13);
 leer_pantalla_de_disco('ALBERTO.MCG');
 new(salva_pantalla_ptr);
 salva_pantalla;
 x := 100;  y := 100;  dx := 1;  dy := 1;
 tocado:= ' ';
 pelota(x,y);
 repeat
  xp := x;
  yp := y;
  escruta_teclado;
  restaura_fondo(xp,yp);
  pelota(x,y);
  delay(8);
until tocado = #27;
inicializar_modo(3);
```

```
end.
```

Aquí se puede modificar la velocidad de desplazamiento de la pelota cambiando el valor de la instrucción DELAY (de 1/500 seg). Cuanto más corto sea el delay, mayor será la impresión de parpadeo de la pelota, debido a que desaparece durante una fracción de segundo, empleada para restaurar el fondo. Esto puede ser molesto si el microprocesador empleado no es rápido, y el único interés del procedimiento ***restaura_fondo*** radica en lo simple que es.

Algunos programadores opinan que el parpadeo molesta y para esto; existe una solución. Después de haber explorado el teclado, se visualiza la pelota en su nueva posición y únicamente después se restaura, punto a punto, la porción de pantalla que constituye la diferencia entre la antigua zona y la nueva. La animación gana fluidez y el detalle anterior desaparece. La programación se vuelve un poco pesada y pierde generalidad en la que depende de la forma del dibujo. Se puede variar la velocidad del desplazamiento con las teclas "+" y "-".

Aquí presento el programa:

```
program pelota_2;
uses
 crt, dos, iniciar;
type
 pantalla_ptr  = ^tipo_pantalla;
 tipo_pantalla = array[0..63999] of byte;
var
 pantalla : tipo_pantalla absolute $A000:$0000;
 salva_pantalla_ptr : pantalla_ptr;
 tocado : char;
 x,y,dx,dy,xp,yp,velocidad : integer;

procedure leer_archivo_de_disco(nombre_archivo : String);
var
 archivo : file of tipo_pantalla;
begin
 assign(archivo,nombre_archivo);
```

```
  reset(archivo);
  read(archivo,pantalla);
  close(archivo);
 end;

 procedure salva_pantalla;
 begin
  move(pantalla,salva_pantalla_ptr^,64000);
 end;

 procedure pelota(x,y : word);
 var
  j : word;
 begin
  fillchar(pantalla[320*y + x + 2],4,10);
  fillchar(pantalla[320*(y+1) + x + 1],6,10);
  for j := 2 to 5 do
   fillchar(pantalla[320*(y+j) + x],8,10);
  fillchar(pantalla[320*(y+7) + x + 2],4,10);
  fillchar(pantalla[320*(y+6) + x + 1],6,10);
 end;

procedure restaura_fondo(x,y : word);
var
 j : word;
 procedure escribir_segmento(direcc,n : word);
 begin
  move(salva_pantalla_ptr^[direcc],pantalla[direcc],n);
 end;
 procedure escribir_punto(direcc : word);
 begin
  pantalla[direcc] := salva_pantalla_ptr^[direcc];
 end;
begin
 if dx > 0 then
  begin
```

```
        escribir_segmento(320*y + x - 1,3);
        escribir_segmento(320*(y+7) + x - 1,3);
        for j := 1 to 6 do
          escribir_punto(320*(y+j) + x - 1);
        escribir_punto(320*(y+1) + x);
        escribir_punto(320*(y+6) + x );
      end;
    if dx < 0 then
      begin
        escribir_segmento(320*y + x + 6,3);
        escribir_segmento(320*(y+7) + x + 6,3);
        for j := 1 to 6 do
          escribir_punto(320*(y+j) + x + 8);
        escribir_punto(320*(y+1) + x + 7);
        escribir_punto(320*(y+6) + x + 7);
      end;
    if dy > 0 then
      begin
        escribir_segmento(320*(y-1) + x,8);
        escribir_segmento(320*y + x,2);
        escribir_segmento(320*y + x + 6,2);
        escribir_punto(320*(y+1) + x);
        escribir_punto(320*(y+1) + x + 7);
      end;
    if dy < 0 then
      begin
        escribir_segmento(320*(y+8) + x,8);
        escribir_segmento(320*(y+7) + x,2);
        escribir_segmento(320*(y+7) + x + 6,2);
        escribir_punto(320*(y+6) + x);
        escribir_punto(320*(y+6) + x + 7);
      end;
  end;

  procedure recoger_de_teclado;
  begin
```

```
if keypressed then
 begin
  tocado:= readkey;
  if tocado= #0 then
   tocado:= readkey;
   case tocado of
    '1',#79: begin
              dx := -1;dy := 1;
             end;
    '2',#80: begin
              dx := 0;dy := 1;
             end;
    '3',#81: begin
              dx := 1;dy := 1;
             end;
    '4',#75: begin
              dx := -1;dy := 0;
             end;
    '6',#77: begin
              dx := 1;dy := 0;
             end;
    '7',#71: begin
              dx := -1;dy := -1;
             end;
    '8',#72: begin
              dx := 0;dy := -1;
             end;
    '9',#73: begin
              dx := 1;dy := -1;
             end;
     '+'   : if velocidad< 10 then
              inc(velocidad);
     '-'   : if velocidad> 0 then
              dec(velocidad);
    else     begin
              dx := 0;dy := 0;
```

```
                end;
            end;
          end;
        if (x < 1) or (x > 310) then
          dx := - dx;
        if (y < 1) or (y > 191) then
          dy := - dy;
        x := x + dx;
        y := y + dy;
      end;
                              ******  programa principal  *****
      begin
        inicializar_modo($13);
        leer_archivo_de_disco('ALBERTO.MCG');
        new(salva_pantalla_ptr);
        salva_pantalla;
        x := 100;  y := 100; dx := 1;  dy := 1;
        velocidad:= 0;  tocado:= ' ';
        repeat
          recoger_de_teclado;
          pelota(x,y);
          restaura_fondo(x,y);
          delay(10 - velocidad);
        until tocado= #27;
        inicializar_modo(3);
      end.
```


este programa está en el archivo "pelo2vga.pas"

3.5 Animación en modo 19 con la tarjeta VGA

Con lo trabajado en los diferentes aspectos de la visualización y los desplazamientos, podemos hasta este punto notar que la animación no tiene que

provocar problemas. Los únicos elementos que intervienen en este trabajo son **SU ALMACENAMIENTO Y SU CAPTURA DE UNO DE ELLOS EN LA PANTALLA.**

Estos sprites constituyen diferentes vistas de un mismo objeto. Supongamos que un personaje ocupa una zona de pantalla de 32 pixeles de largo y 50 de ancho. **En 64,000 bytes podemos cargar hasta 64000/(32*50) = 40 actitudes diferentes de este personaje según el principio del dibujo animado**. Después, hay que ejecutar una serie de instrucciones MOVE de esta zona de memoria en lugar donde se encuentra almacenado el sprite elegido, bien en la pantalla real, bien en otra pantalla virtual en la que preparo la imágen completa. Por falta de espacio en los 64K, habrá que direccionar esta zona de memoria mediante un puntero. Dentro de este marco de suposiciones referidas se muestra el siguiente procedimiento:

```
VAR
      ZONA_SPRITE:PANTALLA_PTR;
PROCEDURE mostrar_sprite(x,y:word;linea,columna:byte);
var
      i:word;
begin
      for i:= 0 to 49 do
        move(zona_sprite^(320*j+50*linea+32*columna),pantalla(320*(y+j)+x),32);
end;
```

linea y columna representan la posición del sprite que deseamos visualizar mediante los 40 (4 líneas a 10 columnas), almacenados con anterioridad en la zona de memoria reservada. Basta sustituir la pantall virtual, siguiendo el principio anterior. Hay que incluir la instrucción NEW(zona_sprite) al principio del programa, antes de cargar la imágen en la zona de memoria. El listado del programa completo se muestra a continuación:

```
program platillo_VGA;
uses
 crt,dos,iniciar;
type
 pantalla_ptr= ^tipo_pantalla;
```

```
  tipo_pantalla = array[0..63999] of byte;
 var
  pantalla  : tipo_pantalla absolute $A000:0;
  vson      : integer;
  i         : byte;
  res       : registers;
  salva_pantalla_ptr,
  dibujo_ptr,
  pantalla_virtual    : pantalla_ptr;

 procedure escribir_punto(x,y,color : integer);
 begin
  with res do
   begin
    ah:= $0C; al:= color; cx:= x; dx:= y;
   end;
  intr($10,res);
 end;

 procedure vuelo(x,y,numero : byte);
 type
  duende  = array[0..7] of byte;
 const
  alto  : duende = (13,14,11,10,10,08,10,19);
  largo : duende = (20,20,23,25,23,25,26,30);
 var
  linea : byte;
 begin
  for linea := 0 to 2*alto[numero] do
   move(dibujo_ptr^[320*(50*(numero div 2) +
                         linea)+ 160*(numero mod 2)],
    pantalla_virtual^[320*(y*2 + linea) + x*4],4*largo[numero]);
 end;

 procedure restaura_pantalla_virtual(zona : byte);
 begin
```

```
 move(salva_pantalla_ptr^[320*zona],
        pantalla_virtual^[320*zona],15000);
end;

procedure salva_pantalla;
begin
 move(pantalla,salva_pantalla_ptr^,64000);
end;

procedure salva_pantalla_virtual(zona : byte);
begin
 move(pantalla_virtual^[320*zona],pantalla[320*zona],15000);
end;

procedure viento;
var
 i : integer;
begin
 for i := 1 to 300 do
 sound(vson + 16*random(255));
 if vson > 500 then
  vson := vson - 500;
 sound(24);
end;

procedure mover_imagen(nombre,x,y,
                                numero : byte; dx,dy : integer);
var
 i, j : byte;
begin
 for i := 1 to nombre do
  begin
   j := y + dy*i;
   restaura_pantalla_virtual(2*j);
   vuelo(x+dx*i,j,numero);
   viento;
```

```
      salva_pantalla_virtual(2^);
    end;
  end;

  procedure descompactar;
  const
   largo= 5153;
  type
   compactado = ^archivo_compactado;
   archivo_compactado = array[1..largo] of byte;
  var
   primero       : boolean;
   mem_compactado : compactado;
   archivo       : file of archivo_compactado;
   archivo_original, archivo_comprimido, j, i: word;
  begin
   new(mem_compactado);
   assign(archivo,'PLATILLO.CPT');
   reset(archivo);
   read(archivo,mem_compactado^);
   close(archivo);
   archivo_comprimido := 0;
   archivo_original := 0;
   primero:= true;
   repeat
    if archivo_original > 0 then
     inc(archivo_original,2)
    else
     inc(archivo_original);
   j := mem_compactado^[archivo_original];
   for i := 1 to j do
    begin
     if not primero then
      inc(archivo_comprimido)
     else
      primero:= false;
```

```
        dibujo_ptr^[archivo_comprimido] :=
                mem_compactado^[archivo_original+1];
      end;
    until archivo_original = largo-2;
    dibujo_ptr^[archivo_comprimido+1] :=
                mem_compactado^[archivo_original+1];
    dispose(mem_compactado);
  end;
                        **** Programa principal ****
  begin
    inicializar_modo(19);
    new(salva_pantalla_ptr);
    new(dibujo_ptr);
    new(pantalla_virtual);
    descompactar;
    for i := 0 to 170 do
      escribir_punto(2*random(160),random(200),15);
    salva_pantalla;
    vson := 13000;
    mover_imagen(10,58,70,0,-2,0); mover_imagen(4,38,70,1,-2,0);
    mover_imagen(4,30,70,2,-2,0); mover_imagen(4,22,70,3,-2,0);
    mover_imagen(4,18,70,4,-1,-2); mover_imagen(4,18,60,5,1,-2);
    mover_imagen(4,22,50,6,2,-2); mover_imagen(5,30,40,7,4,-4);
    delay(100);
    nosound;
    restaura_pantalla_virtual(40);
    salva_pantalla_virtual(40);
    inicializar_modo(3);
  end.
```

Al principio del programa se lee el archivo de imágenes compactadas. El tamaño de archivo no puede ser variable y está definido por una constante. El archivo está guardado en el puntero memoria_compactado creado para tal objetivo pero que es borrado por DISPOSE cuando termine el procedimiento. La imágen descompactada se incluye en la tabla reservada con el puntero sprite_ptr.

Después hay que dibujar el cielo. Una visualización aleatoria de puntos lo realiza. Este cielo es el fondo que se guarda por el puntero *salva_pantalla_ptr*. Según el principio en "simuladores de vuelo", hay que ejecutar la sentencia para cada imágen de animación:

1. Restaurar el fondo en la pantalla virtual mediante la instrucción *restaura_pantalla_virtual*. Este procedimiento cambia solo un cuarto de pantalla que engloba la posición del sprite.
2. Para mostrar el sprite en el fondo (*mostrar_sprite*), el procedimiento utiliza una serie de MOVE, uno por cada línea de pantalla. La visualización no hace más que en altura y en longitud de cada imágen, valores incluidos en la tabla de constantes. En lo respecta a la longitud, se incluye el cuarto del valor para facilitar la transportabilidad a la versión CGA.
3. Activar la pantalla virtual. Para hacerlo, hay que transferir mediante MOVE, un cuarto del contenido en la pantalla real.

El conjunto de esto, se encuentra en el procedimiento *mover_imagen*.

Un dibujo animado no se concibe sin sonido. la rutina ***viento*** así lo hace. Determinamos en la partida un umbral de frecuencias elevada, llamado sonido. Se disminuye en cada imágen para dar la impresión de un efecto especial. Un ciclo produce un "ruido" aleatorio de frecuencia ligada a ese umbral.

3.6 Rotación de la pantalla

Para simular el rotamiento de la pantalla hay que desplazar las filas de bytes con una serie de instrucciones MOVE, pero como hay que desplazar 64000 bytes cada vez, la operación resulta lenta y la impresión de "ondular" la imágen en la pantalla. Es más práctico y simple utilizar una facilidad que dan las tarjetas VGA y CGA, de modificar la dirección inicial de la memoria de video, tomada por el monitor de visualización. Para hacerlo, hay que introducir el valor 1DH (13) en el registro ·D4h, lo que significa que queremos modificar el byte de menor peso de la dirección en tampón de visualización. Después hay que introducir en el puerto **PORT 3d5h** tantos elementos de 1/80 de largo de la pantalla que queremos desplazar esta dirección. Ya dentro del programa, el desplazamiento se efectúa bajo un bucle. Así mismo, para efectuar la comprobación hay que introducir la temporalización.

```
program rotacion_de_pantalla;
uses
 crt,dos,iniciar ;
```

```
type
  tipo_pantalla= array[0..63999] of byte ;
var
  pantalla: tipo_pantalla absolute $A000:$0000 ;

procedure leer_pantalla_de_disco(nomarchivo : String) ;
var
  archivo : file of tipo_pantalla;
begin
  assign(archivo,nomarchivo) ;
  reset(archivo) ;
  read(archivo,pantalla) ;
  close(archivo) ;
end ;

procedure rotacion(derecho: boolean) ;
var
  k : byte ;
begin
  port[$03D4] := 13 ;
  if derecho then
    for k := 1 to 78 do
      begin
        port[$03D5] := k ;
        delay(20) ;
      end
  else
    for k := 1 to 78 do
      begin
        port[$03D5] := 79 - k ;
        delay(20) ;
      end ;
  port[$03D5] := 0 ;
end ;
**** programa principal ****
begin
```

```
  inicializar_modo($13) ;
  leer_pantalla_de_disco('ALBERTO.MCG') ;
  repeat
   rotacion(true) ;
   rotacion(false) ;
  until keypressed ;
  inicializar_modo(3) ;
end.
```

En el programa se puede cambiar el valor de la temporalización para ver que el desplazamiento puede ser más rápido.

Una aplicación de lo anterior, lo tenemos en "hacer temblar la pantalla". Con un ruido adecuado, el efecto nos hará pensar que es un choque. Se puede incorporar esta rutina de aplicaciones en lugar de un simple bip sonoro para comunicarle al usuario que se ha equivocado de tecla. El programa es el siguiente:

```
program pantalla_temblorosa;
uses
 crt,dos,iniciar;
type
 tipo_pantalla= array[0..63999] of byte ;
var
 pantalla: tipo_pantalla absolute $A000:$0000 ;

procedure leer_pantalla_de_disco(nombre_archivo : string);
var
 archivo : file of tipo_pantalla;
begin
 assign(archivo,nombre_archivo);
 reset(archivo);
 read(archivo,pantalla);
 close(archivo);
end ;
procedure temblar;
var
```

```
  i,j : byte ;
 begin
  for i := 1 to 6 do
   begin
    port[$3D4] := 13 ;
    port[$03D5] := 1 + random(2) + 80*(random(3));
    for j := 1 to 32 do
     begin
      sound(20+random(40));
      delay(1);
     end ;
   end ;
  port[$3D4] := 13 ;
  port[$03D5] := 0 ;
  nosound ;
end ;
***** Programa principal ******
begin
 inicializar_modo($13);
 leer_pantalla_de_disco('ALBERTO.MCG');
 repeat
  temblar;
  delay(2000);
 until keypressed ;
 inicializar_modo(3);
end.
```

El siguiente paso de este trabajo, es analizar y aprovechar las bondades de la tarjeta EGA en el modo 13. Al igual que en los capítulos anteriores se abordan aspectos importantísimos, base para crear herramientas software con una interface gráfica vistosa.

CAPÍTULO 4

PROGRAMACIÓN EN MODO 13 DE LA TARJETA EGA/16 COLORES

En este Capitulo abordaremos:

- ***Los aspectos más importantes a considerar de la tarjeta EGA.***
- ***La generación, almacenamiento, compactación y tratamiento de archivos gráficos para esta tarjeta.***
- ***Los principios para la generación de animaciones a través del empleo de las páginas de memoria RAM.***
- ***La programación en modo 13 y sus diferencias con otros modos de esta tarjeta.***

4.1 Inicialización del modo y escritura de un punto

Para la inicialización en este modo, bastaría con emplear la siguiente instrucción:

inicializar_modo(SD);

Para la escritura de un punto, habrá que describir algunos asuntos. Se presenta el código del procedimiento llamado -punto_ega-.

```
procedure punto_ega(x,y:word;color,pagina:byte);
const
  puerto_seq = $3C4;
  mascara_seq = 2;
  puerto_gr = $3CE;
  mascara_gr = 8;
var
  direccion1,direccion2:word;
  bidon,i,numero,mascara:byte;
begin
  direccion1:= $A000 + $200*pagina;
{aqui hay que reemplazar
     $400*pagina en el modo Eh
     $800*pagina en los modos 10h y 12h}
  direccion2 := y*40 + x div 8;
{ Aqui reemplazamos y*80 en los modos Eh,10h, y 12h}
  numero := 7 - x mod 8;
  mascara := 1 SHL numero;
  PORT[puerto_gr]:= mascara_gr;
  PORT[puerto_gr+1] := mascara;
  bidon := MEM[direccion1:direccion2];
  MEM[direccion1:direccion2]:= $FF;
  PORT[puerto_seq]:= mascara_seq;
  PORT[puerto_seq+1]:= 15;
  PORT[puerto_gr]:= mascara_gr;
  PORT[puerto_gr+1]:= $FF;
end;
```

```
  mascara := 1 shl num;
  port[port_gr] := mascara_gr;  {8 dentro de 3CE : seleccion de
 registro de mascara de bit}
  port[port_gr + 1] := mascara; {carga la mascara de bit}
  bidon := MEM[direccion1:direccion2];
  MEM[direccion1:direccion2] := $00;
  port[port_seq] := mascara_seq; {puntero al registro de planos de
 mascaras}
  port[port_seq + 1] := col;  {activa los diferentes planos}
  bidon := MEM[direccion1:direccion2];
  MEM[direccion1:direccion2] := $FF;
  port[port_seq] := mascara_seq;
  port[port_seq + 1] := $0F;
  port[port_gr] := mascara_gr;
  port[port_gr + 1] := $FF;
 end;

procedure anillo(destello,pagina : byte);
var
 i     : word;
 angulo : real;
begin
 for i := 1 to 500 do
  begin
   angulo := random(360)*PI/180;
   punto_ega(160+round((destello+random(destello))*
   cos(angulo)),100+round((destello+random(destello))*
   sin(angulo)),1+random(15),pagina);
  end;
end;
procedure activar_pagina(pagina : byte);
begin
 with reg do
  begin
   ah := 5;
   al := pagina;
  end;
 intr($10,reg);
```

```
end;
{programa principal}
begin
 inicializar_modo(13);
 directvideo := false;
 gotoxy(3,11);
 write('Creando la imagen en la p gina ');
 gotoxy(4,14);
 write('   un poco de paciencia..');
 for i := 1 to 7 do
  begin
   gotoxy(36,11);
   write(i);
   anillo(6*i,i);
  end;
 i := 0;
 sentido:= TRUE;
 repeat
  if sentido then
   inc(i)
  else
   dec(i);
  if i = 7 then
   sentido := FALSE;
  if i = 1 then
   sentido := TRUE;
  activar_pagina(i);
 until keypressed;
 inicializar_modo(3);
end.
```

```
    write('Tamano = ',arch_compac,' compactado
     a ',cpt,'%');
    assign(archivo,nom_arch + '.CPT');
    reset(archivo);
    for cpt := 1 to arch_compac do
      write(archivo,pant_compac^[cpt]);
    close(archivo);
  end;
```

Para utilizar este procedimiento, es necesario haber almacenado con anterioridad el contenido de la pantalla en la variable llamada -plano-. Al principio de este procedimiento se crean dos punteros de 32k de longitud que son deshechados al final. El puntero -pant_orig- recibe la imagen; el segundo, -pant_compac- es usado para recibir la pantalla ya compactada. La imágen que ya esta compactada se almacena en un archivo que tiene por extensión ".cpt".

Como complemento a este procedimiento de compactación, se escribe a continuación su contraparte contenida en una unidad especial de descompactación para volver a la imágen a su estado original.


```
unit descompactacion;
interfase
 type
   plano_ptr = ^tipo_plano;
   tipo_plano = array[0..3,0..7999] of
    byte;
 var
   plano:plano_ptr;
procedure descompactar_ega(nom_arch:string);

implementation
procedure descompactar_ega(nom_arch:string);
type
 plano_4_ptr = ^planox_4;
 planox_4    = array[1..32000] of byte;
var
 memcompp,plano_4:plano_4_ptr;
 archivo: file of byte;
 arch_orig,arch_compac,j,i,talla,
```

```
  longitud:word;
 p: byte;
begin
 new(memcompp);
 new(plano_4);
 new(plano);
 assign(archivo,nom_arch);
 reset(archivo);
 longitud := filesize (archivo);
 for i:= 1 to longitud do
  read(archivo,memcompp^[i]);
 close(archivo);
 arch_compac := 0;
 arch_orig := 0;
 repeat
  if arch_orig = o then
   inc(arch_orig,2);
  else
   inc(arch_orig);
  j := memcompp^[arch_orig];
  for i := 1 to j do
  begin
   inc[arch_compac];
   plano_4^[archcompac]:= memcompp^
    [arch_orig + 1];
  end;
 until arch_orig = longitud-2;
 plano_4^[arch_compac + 1]:= memcompp^
 [arch_orig + 1];
 for i := 1 to 3 do
 move(plano_4^[8000*i + 1],plano^[i,0],
 8000);
 dispose(memcompp);
 dispose(plano4);
```

La forma de hacer uso de este procedimiento es insertar "descompactar" dentro de la serie de archivos con terminación ".TPU" llamados mediante la instruccion USES, y podremos releer todas las imágenes compactadas en el

formato EGA modo 13, sin necesidad de indicar la longitud del archivo obtenido mediante la instrucción **FILESIZE**.

4.6 Animación con el modo 13 de la tarjeta EGA

Para la animación, se volvió a utilizar la imágen del modo VGA, que está disponible con todos sus colores originales (ya que en este modo se creó), retomando la animación estrella para resaltar la idea precisa y de sus complicaciones en cuanto a lo pesado que es programar la tarjeta EGA en relación con la tarjeta VGA. Al utilizar la misma fórmula de programación, la pelota del programa "pelotega.pas", vemos que la misma se desplaza con una lentitud que impacienta de forma que el programa no parece útil. Para mejorar lo anterior, si utilizamos dos páginas de pantalla.

Alternativamente, activamos una cuando en la otra la borramos la pelota y la colocamos en su nueva posición. El tiempo de rotación de una página a otra esta en el orde de los milisegundos. A todo esto, se le aumentó un tiempo de restauración del fondo, pero la lentitud de la pelota es que en su creación se realiza pixel a pixel. El código del programa es:

```
program pelota_ega_de_Albert;
uses
 crt,dos, iniciar;
const
 port_seq = $3C4;
 port_gr  = $3CE;
 mascara_seq = 2;
 mascara_gr  = 8;
type
 plano_ptr  = ^tipo_plano;
 tipo_plano  = array[0..3] of array[0..7999] of byte;
var
 res    : registers;
 plano  : plano_ptr;
 tocado  : char;
 x, y, dx, dy: integer;
 bascula, p: byte;
```

```
procedure activa_pagina(pagina : byte);
begin
 with res do
  begin
   ah:= 5;
   al:= pagina;
  end;
 intr($10,res);
end;
procedure punto_ega(x,y : word; color,pagina : byte);
var
 direccion1,
 direccion2  : word;
 bidon,
 numero,
 mascara : byte;
begin
 direccion1 := $A000+pagina*$200;
 direccion2 := y*40 + x div 8;
 numero := 7 - x mod 8;
 mascara := 1 shl numero;
 port[port_gr] := mascara_gr;
 port[port_gr + 1] := mascara;
 bidon := MEM[direccion1:direccion2];
 MEM[direccion1:direccion2] := $00;
 port[port_seq] := mascara_seq;
 port[port_seq + 1] := color;
 bidon := MEM[direccion1:direccion2];
 MEM[direccion1:direccion2] := $FF;
 port[port_seq] := mascara_seq;
 port[port_seq + 1] := $0F;
 port[port_gr] := mascara_gr;
 port[port_gr + 1] := $FF;
end;

procedure restaura_pantalla;
begin
 for p := 0 to 3 do
```

```
    begin
     port[$3C4] := 2;
     port[$3C5] := 1 shl p;
     move(plano^[p],MEM[$A000:0000],8000);
     move(plano^[p],MEM[$A200:0000],8000);
    end;
   port[$3C4] := 2;
   port[$3C5] := 15;
  end;
  procedure leer_pantalla_de_disco(nombre_archivo : string);
  var
   archivo : file of tipo_plano;
  begin
   assign(archivo,nombre_archivo);
   reset(archivo);
   read(archivo,plano^);
   close(archivo);
   restaura_pantalla;
  end;
  procedure pelota;
  var
   i, j : byte;
  begin
   for j := 0 to 3 do
    punto_ega(x + 2 + j,y,10,bascula);
   for j := 0 to 5 do
    punto_ega(x + 1 + j,y + 1,10,bascula);
   for j := 2 to 5 do
    for i := 0 to 7 do
     punto_ega(x + i,y + j,10,bascula);
   for j := 0 to 5 do
    punto_ega(x + 1 + j,y + 6,10,bascula);
   for j := 0 to 3 do
    punto_ega(x + 2 + j,y + 7,10,bascula);
   activa_pagina(bascula);
   delay(12);
   if bascula = 0 then
    bascula := 1
```

```
  else
    bascula := 0;
end;
procedure restaura_fondo;
begin
 for p := 0 to 3 do
   begin
    port[$3C4] := 2;
    port[$3C5] := 1 shl p;
    move(plano^[p,40*(y-1) + x div 8],
    MEM[$A000+bascula*$200:40*(y-1) + x div 8],402);
   end;
 port[$3C4] := 2;
 port[$3C5] := 15;
end;
procedure escruta_teclado;
begin
 if keypressed then
   begin
    tocado := readkey;
    if tocado = #0 then
      tocado := readkey;
      case tocado of
       '1',#79: begin
                  dx := -1;
                  dy := 1;
                end;
       '2',#80: begin
                  dx := 0;
                  dy := 1;
                end;
       '3',#81: begin
                  dx := 1;
                  dy := 1;
                end;
       '4',#75: begin
                  dx := -1;
                  dy := 0;
```

```
                end;
            '6',#77: begin
                dx := 1;
                dy := 0;
               end;
            '7',#71: begin
                dx := -1;
                dy := -1;
               end;
            '8',#72: begin
                dx := 0;
                dy := -1;
               end;
            '9',#73: begin
                dx := 1;
                dy := -1;
               end;
          else    begin
                dx := 0;
                dy := 0;
               end;
        end;
      end;
     if (x < 1) or (x > 310) then
      dx := - dx;
     if (y < 1) or (y > 191) then
      dy := - dy;
     x := x + dx;
     y := y + dy;
    end;
    begin
     inicializar_modo(13);
     new(plano);
     leer_pantalla_de_disco('ALBERTO.EGA');
     x := 100;
     y := 100;
     dx := 1;
     dy := 1;
```

```
tocado := ' ';
bascula := 0;
repeat
 escruta_teclado;
 pelota;
 restaura_fondo;
until tocado = #27;
inicializar_modo(3);
end.
```

4.7 Diferencias entre el modo 13 y los modos EGA,14,16 y VGA 18

¿Porque se ha limitado este trabajo al estudio de los modos 14,16 y 18? Es porque son los únicos modos que permiten la visualización de 16 colores. El modo 15 es monocromático, el 17 solo soporta 2 colores lo que hace que sea poco atractivo. Y si se manejan los modos superiores, también se sabe manejar los modos simples.

Todos los modos son compatibles con el 13 que se trata en este cápitulo, pero solo el modo 18 es un modo VGA con una paleta de colores de 262,144.

En el modo 18, los procedimientos tanto para tomar color como para escribirlo son utilizables, sin ningún cambio, con la única limitación de que sólo están disponibles 16 colores que son muy pocos a comparacion de los 256 del modo 19.

Si se retoma la declaración de los procedimientos en orden cronológico se sabremos cuales son los cambios a efectuar para que sean utilizados en cualquier modo y cualquier lenguaje de programación.

Las modificaciones, por ejemplo al procedimiento -punto_ega- se indican en el mismo. La animación utilizando las ocho páginas de la pantalla en el modo 13, pueden realizarse solo con cuatro en el modo 14, y solo con dos en el modo 16, por lo que pierde interés. Es imposible realizar animaciones en el modo 18 ya que no existe una segunda página disponible. A continuación se enlistan y describen las diferencias entre los diferentes modos, pero ahora a nivel de declaración:

El modo 14, las declaraciones conservan la misma forma que en el modo 13, teniendo en cuenta el tamaño de los planos de bits. Por ejemplo:

```
type
 plano_ptr  = ^tipo_plano;
 tipo_plano  = array[0..3] of array[0..15999] of byte;
var
 plano :plano_ptr;
```

En los modos 16 y 18 extendiendo los indices de las tablas de (0..27999) para el modo 16 y de (0..38999) para elmodo 18, provocaria un mensaje de error en la compilación ya que este tipo de variable desborda los 64k máximos para un bloque de datos. Hay que declarar cada plano como una variable, por ejemplo para el modo 18, sería:

```
type
 plano_ptr  = ^tipo_plano;
 tipo_plano  = array[0..3] of array[0..15999] of byte;
var
plano0,plano1,plano2,plano3:plano_ptr;
```

Sobre todo, no hay que olvidar inicializar los cuatro punteros, mediante las siguientes instrucciones:

```
NEW(plano0);
NEW(plano1);
NEW(plano2);
NEW(plano3);
```

Para los proedimientos salva y restaura pantalla, en modo 14 sólo se reemplazan los 8 por los 16 bytes a transladar para cada plano. En base a las instrucciones de arriba para los modos 16 y 18, los procedimientos deberán tomar una nueva forma. Para el modo 18, por ejemplo:

```
Procedure salva_pantalla_modo_18;
begin
 port($3CE):= 4;
 port($3CE):= 0;
 move(mem[$A000:0000 , plano0^,38400);
 port($3CE):= 1;
 move(mem[$A000:0000 , plano1^,38400);
 port($3CE):= 2;
```

```
    move(mem[$A000:0000 , plano2^,38400);
    port($3CE):= 3;
    move(mem[$A000:0000 , plano3^,38400);
    port($3CE):= 0;
  end;
  Procedure restaura_pantalla_modo_18;
  begin
    PORT[$3C5 := 2;
    PORT[$3C5 := 1;
    move(plano0^,MEM[$A000:0000 ,38400);
    PORT[$3C5 := 2;
    move(plano1^,MEM[$A000:0000 ,38400);
    PORT[$3C5 := 4;
    move(plano2^,MEM[$A000:0000 ,38400);
    PORT[$3C5 := 8;
    move(plano3^,MEM[$A000:0000 ,38400);
    PORT[$3C$ := 15;
  end;
```


En el procedimiento ***punto_virtual*** si reemplazamos 40*y por 80*y para tener en cuenta la nueva anchura de la pantalla de estos tres modos. Para los modos 16 y 18, hay que sustituir el bucle de los cuatro valores de p, por cuatro instrucciones, identificando cada una a un puntero en particular, siguiendo el ejemplo anterior.

Los procedimientos ***leer_imagen_de_disco*** sirven para el modo 14, en los modos restantes habrá que sustituir este procedimiento por:

```
procedure leer_imagen_de_disco(nombre:string);
var
  archivo:FILE OF tipo_plano;
begin
  assing(archivo,nombre);
  reset(archivo);
  read(archivo,plano0^);
  read(archivo,plano1^);
  read(archivo,plano2^);
  read(archivo,plano3^);
```

```
    close(archivo);
    restaura_pantalla;
  end;
  procedure grabar_imagen_en_disco(nombre:string);
  var
    archivo:FILE OF tipo_plano;
  begin
    salva_pantalla;
    assing(archivo,nombre);
    rewrite(archivo);
    write(archivo,plano0^);
    write(archivo,plano1^);
    write(archivo,plano2^);
    write(archivo,plano3^);
    close(archivo);
  end;
```


En lo relacionado a la compactación de archivos de imágenes, en el modo 14 basta con duplicar el tamaño de los punteros y los índices que se recogen. Para otros modos, habrá que dividir en varios los punteros.

En el procedimiento ***color_pantalla***, hay que modificar el número de bytes a duplicar para llenar la pantalla con 16000, 28000 y 38400 para los modos 14,16 y 18 respectivamente.

Los procedimientos ***rectangulo, salva_imagen y restaura_imagen*** corren sin ningún problema en los tres modos, si sustituimos **40*linea** por **80*linea**.

Partiendo de estos elementos, comprobamos que las modificaciones son menores. Ya se conoce el camino, no tendremos ningún problema cuando se desarrollen programas gráficos.

Antes de escoger un modo de desarrollo, tendremos que verificar las características de la pantalla (anchura y largo), además de lo que implicaría guardar las imágenes en disco y el direccionamiento de los punteros. **Debemos medir, hasta que punto es difícil la compatibilidad con la optimización de los programas.**

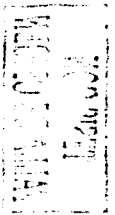
Lo que no debe olvidarse...

Aunque aprovechamos al máximo la capacidad esta tarjeta y su programación, la animación se realiza más lenta que con la tarjeta VGA. Se recomienda, que antes de realizar una animación, debemos seleccionar una tarjeta y su modo de programación. La lentitud es debida a la manipulación de máscaras de bits. En caso contrario, si la aplicación programada necesita desplazamientos y visualización de bytes, se deberán tomar en cuenta lo siguiente:

- Al leer la imágen de disco, guardarla en un puntero. Se vuelve a restaurar con un procedimiento especial en las primeras dos páginas de la memoria de video en las direcciones A0000h y A2000h.
- El objeto deberá dibujarse pixel a pixel en la página inactiva, según el valor de una variable que tenga los valores 0 y 1. Esta es la tarea más lenta del bucle. Y aunque las líneas comprendidas entre la 2 y la 5, posean justamente un byte de longitud, no se podrá dibujar mediante la visualización por byte. Esto sería un poco más rápido, ya que no se tendría que calcular la máscara de bits. Pero estas líneas solo comienzan una vez en 8 en dirección de un byte sobre la pantalla, que descarta por completo esta salida.
- Tan rápido como ya se dibuje el objeto, se vuelve a activar la página donde se encuentra.
- Al momento de borrar el objeto, la página se vuelve inactiva. Este borrado se realiza restaurando toda la banda horizontal de la pantalla, diez líneas más de dos bytes. **¡Es vital que esta rutina se corra con mas velocidad que la que dibuje el objeto!.** Debemos incluir una instrucción que lo detenga para dejar tiempo a la página para "desactivarse", y así cumplir el borrado del objeto en la parte baja de la pantalla con la sensación de parpadeo.
- El ejemplo perpecto de rapidez entre pixeles y bytes es el programa -CUABLEGA.PAS-, donde la animación se ha vuelto lo suficientemente rápida para reestablecer la opción de modificación

de la velocidad mediante la pulsación de las teclas "+" y "-" del teclado. La visualización del cuadrado es un procedimiento en el que se utiliza el color blanco. La restauración de la imagen el fondo, después del desplazamiento llama a un procedimiento -MOVE- que transfiere una línea de bytes. Se busca entre estos bytes de la imagen de fondo guardada con una variable tipo puntero después de la lectura en el disco. La línea transferida es la superior o la inferior del cuadrado según si asciende o desciende la pelota.

Para concluir, el capítulo 5 aborda la conversión de imágenes y portabilidad de las mismas de un modo a otro; en él se podrá aplicar todo lo que en los primeros cuatro capítulos afinando y aplicando todo lo que se ha descrito.



CAPÍTULO 5

CONVERSIÓN DE IMÁGENES Y PORTABILIDAD DE UN MODO A OTRO

En este Capítulo abordaremos:

- ***Los formatos gráficos más importantes BMP, PCX, GIF, JPG. y su codificación en un lenguaje de alto nivel.***
- ***Cómo visualizar un gráfico creado para VGA en las tarjetas CGA y EGA.***
- ***La lectura de una imagen de 320 x 200 pixeles en formato EGA de 640 x 200 pixeles.***
- ***Incluimos un programa que resultará útil para cualquier tipo de tarjeta gráfica que se emplee.***

5.1 Formatos de archivos gráficos BMP, PCX, GIF, JPG

Una imagen puede almacenarse en un archivo siguiendo diferentes formatos. Unos son más sencillos que otros, muchos utilizan compresión de datos, cada uno tiene sus ventajas y sus desventajas, pero todos ellos tienen algunas características en común: Siempre se utiliza una cabecera en el archivo que identifica el tipo de archivo del que se trata, y contiene información necesaria para interpretar el archivo, como el tamaño de la imagen o el número de colores. A continuación se encuentran, generalmente comprimidos con un algoritmo específico de ese formato, los datos de la imagen. Una vez descomprimidos, estos datos indican el color de cada pixel (punto) de la imagen.

La imagen puede tener más o menos colores, y en función del número de colores serán necesarios más o menos bits por pixel para indicar el color del que se trata. Cuantos más colores, mejor calidad tendrá la imagen, pero más ocupará el archivo. Normalmente el número de colores es 16, 256 ó 16 millones, lo que requiere 4, 16 ó 24 bits por pixel. En el caso de utilizar 16 ó 256 colores, debe especificarse a que color real corresponde cada uno de esos colores, es decir, que cantidades de Rojo, Verde y Azul serán utilizadas para representar el color en la pantalla. La tabla que asocia a cada color las correspondientes cantidades de Rojo, Verde y Azul se llama paleta de colores. Puede ser modificada en función de la imagen, por lo que es necesario guardarla en el archivo. En el caso particular de utilizar 16 millones de colores, no se utiliza paleta de colores, pues la relación entre número de color y cantidad de Rojo, Verde y Azul es implícita: De los 24 bits por pixel, 8 especifican la cantidad de Rojo, otros 8 la cantidad de Verde, y otros 8 la de Azul.

Los archivos gráficos contendrán pues una cabecera, los datos de los pixeles y la paleta de colores (salvo si se usan 24 bits por pixel).

Veamos a continuación cuales son los formatos más utilizados de archivos gráficos.

BMP

El formato BMP (Windows BitMaP) es probablemente el formato de archivo para imágenes más simple que existe. **Aunque teóricamente permite compresión, en la práctica nunca se usa,** y consiste simplemente en una cabecera y a continuación los valores de cada pixel, comenzando por la línea de más abajo y terminando por la superior, pixel a pixel de izquierda a derecha. Parece ser el formato preferido por Bill Gates. Su única ventaja es su sencillez. Su gran desventaja es el enorme tamaño de los archivos. Un ejemplo de programa en Turbo Pascal para visualizar archivos BMP es VERBMP.PAS. Se trata de un programa para visualizar archivos BMP de 16 ó 256 colores de cualquier tamaño. Si la imagen es tan grande que no cabe en la pantalla, solo se visualiza la esquina superior izquierda. La estructura exacta del formato BMP puede entenderse facilmente observando el programa VERBMP.PAS.

PCX

En el formato PCX (de PC Paintbrush), los datos están comprimidos con un algoritmo llamado RLE. Suponiendo que estemos utilizando 256 colores (un byte por pixel), consiste simplemente en reemplazar las secuencias de N pixels consecutivos del mismo color por dos bytes, de forma que el primero indique el número N de repeticiones y el segundo indique el color. Para saber si un byte indica simplemente el color de un pixel o el número de repeticiones de una secuencia de pixels, se utiliza el siguiente criterio: si el byte es inferior o igual a 192, corresponde al color de un único pixel, pero si el byte es superior a 192, poniendo a 0 los 2 bits más significativos indicará el número de pixels repetidos a continuación, que tendrán el color indicado por el byte siguiente (aunque éste sea superior a 192). Este algoritmo permite reducir el tamaño del archivo cuando la imagen sea un dibujo, pues serán muchos los pixels consecutivos del mismo color. Sin embargo cuando haya algún pixel aislado de un color X>192, éste se reemplazará por los bytes 193, X, indicando que se trata de una secuencia de un pixel de color X. Por ese motiv o

es posible que en algunos casos llegue a aumentar el tamaño del archivo. Esto suele pasar con imagenes escaneadas, pues es improbable que varios pixels consecutivos tengan exactamente el mismo color. Un ejemplo de programa en Turbo Pascal para visualizar archivos PCX es VERPCX.PAS. Se trata de un programa para visualizar archivos PCX de 256 colores y de cualquier tamaño, pero si no cabe en pantalla solo se visualiza la esquina superior izquierda. La estructura exacta del formato PCX está explicada en el archivo VERPCX.PAS. Otros ejemplos para decodificar archivos PCX se encuentran en los SWAG.

GIF

El formato GIF (Graphic Interchange Format) fue inventado por Compuserve. Utiliza un algoritmo de compresión LZW modificado (el mismo en el que se basan los programas de compresión convencionales), mucho más complejo que el RLE. Consiste en no detectar solo las repeticiones de un color, sino en detectar las repeticiones de ciertas secuencias, mediante un diccionario que se va construyendo. Al igual que el formato PCX, funciona especialmente bien con dibujos, en los que muchos puntos consecutivos tienen el mismo color, o se repiten secuencias de colores, pero también funciona bien con fotografías escaneadas o cualquier otra imagen. Además permite definir un color transparente, por lo que es muy util para páginas Web. Una explicación detallada del formato GIF se encuentra en pcgpe (PC Game Programmers Encyclopaedia). Dicho archivo también incluye explicaciones de formatos de audio, como implementar sprites, programación de la VGA, y código optimizado para juegos.

JPG

El formato JPG utiliza un complejo algoritmo de compresión con pérdida de información. Es decir, el algoritmo modifica ligeramente los datos de forma que la imagen puede comprimirse mucho más. Gracias a esto el formato JPG es uno de los que más comprimen. Las modificaciones realizadas son inapreciables si se trata de

una fotografía escaneada. Sin embargo, si se trata de un dibujo aparecen puntos visibles en la imagen. Por esta razón este formato es muy util para fotografías escaneadas, pero no para dibujos, y por eso no permite definir colores como transparentes. Existe una unit llamada PASJPG10.ZIP en Pascal para decodificar JPG.

5.2 ¿Qué camino tomar para la compatibilidad entre los distintos formatos de imágenes?

Dentro de los programas gráficos, existen algunos que permiten almacenar archivos de imágenes en disco, mediante una serie de pasos que le permiten al programa tener un canal de comunicación con otros programas, digitalizadores, u otros elementos de uso para los programas y archivos con imágenes. Todas las herramientas software y hardware tienen sus propias reglas para codificar imágenes. Intentar decodificarlas, puede resultar una tarea un poco difícil de terminar y más cuando éstas se encuentran compactadas a no ser de que se conozca el algoritmo de descompactación.

Por ejemplo, si revisamos el tamaño de los archivos gráficos generados en los capítulos anteriores, veríamos que todos tienen los siguientes tamaños, esto se ilustra en la tabla 9:

Tipo de archivo	Tamaño en bytes
CGA	16,384 bytes
EGA	32,000 bytes
VGA (MCGA)	64,000 bytes

Tabla 9. Tamaños de los archivos generados en diversas tarjetas

Lo anterior es un error, ya que corresponden todos a formatos de 320x200 pixeles. Hay que notar que los CGA de 16,000 bytes serían suficientes para guardar toda la imagen. De cualquier forma debemos considerar lo siguiente:

1. Para tener un espacio vacío de 192 bytes que existen en el último byte de líneas pares y el primer byte de líneas impar, y

2. Para guardar los bytes en un archivo con un solo elemento, debería grabarse en un bloque de 16,192 bytes.

5.3 Como ver un gráfico creado para VGA en tarjetas CGA y EGA

La posibilidad de leer en una computadora con una tarjeta EGA o CGA una imagen creada en modo 19 VGA representa un gran interés. Un programa gráfico creado según este principio podrá funcionar con seguridad en cualquier tarjeta gráfica.

El principio es bastante simple y se basa en la norma de 40 bytes. En una imagen VGA en modo 19, el programa lee ocho líneas para poder construir cada pixel. para visualizar en modo EGA o CA, el programa utiliza las cuatro líneas en el primer caso, y sólo dos primeras en el segundo.

El siguiente código realiza la operación de conversión de un archivo imagen hecho en modo 19 VGA, al formato de otra tarjeta.

```
program cambiar_formato_de_imagen;
uses
 crt, dos;
var
 modo          : byte;
 nombre_archivo : string;
 scr           : integer;
 res           : registers;
 tocado        : char;
 ok,ega          : boolean;

procedure inicializar_modo(valor : byte);
begin
 res.ah := 0;
 res.al := valor;
 intr($10,res);
end;

function Ver_modo: byte;
```

```
var
  tarjeta: byte;
begin
  with res do
    begin
      ah:= $F;
      intr($10,res);
      tarjeta:= al;
    end;
  ver_modo:= tarjeta;
  if (tarjeta>= 13) and (MEM[$40:$88] = 0) then
    ver_modo:= 0;
end;

function ver_ega: boolean;
begin
  inicializar_modo(13);
  if ver_modo = 13 then
    ver_ega:= true
  else
    ver_ega:= false;
  inicializar_modo(3);
end;

function detectar : byte;
var
  i,  j,  tarjeta: byte;
begin
  for i := 1 to 3 do
    case i of
      1 : begin
      inicializar_modo(19);
          if ver_modo= 19 then
            begin
              detectar := 19;
              exit;
```

```
            end;
          end;
        2 : begin
            inicializar_modo(13);
            if ver_modo= 13 then
             begin
              detectar := 13;
              exit;
             end;
          end;
        3 : begin
        inicializar_modo(4);
            if ver_modo= 4 then
             begin
              detectar := 4;
              exit;
             end
            else
             begin
              write('no he detectado tarjeta gr fica');
              halt;
             end;
          end;
    end;
end;

function longitud(nombre : string) : longint;
var
 f : file of byte;
begin
  assign (f,nombre);
  reset (f);
  longitud:= filesize(f);
  close (f);
end;
```

```
function existe(archivo: string) : boolean;
var
 fich : file of byte;
begin
 assign(fich,archivo);
 {$I-}
  reset(fich);
  close(fich);
 {$I+}
 existe := ioresult = 0;
end;

procedure cambiar_formato;
type
 ELB  = ^ELBM;
 ELBM = array[1..64824] of byte;
var
 pantalla_ibm   : ELB;
 archivo_b    : file of ELBM;
 a, b,c       : byte;
 i,  j, k, byte     : byte;
 direccion: word;
begin
 new(pantalla_ibm);
 assign(archivo_b,nombre_archivo);
 reset (archivo_b);
 read(archivo_b,pantalla_ibm^);
 if EGA then
  begin
   inicializar_modo(13);
   for k := 0 to 3 do
    begin
  port[$03C4] := 2;
  port[$03C5] := 1 shl k;
      for j := 0 to 199 do
   move(pantalla_ibm^[40*(8*j + k)+825],MEM[$A000:(j*40)],40) ;
```

```
      end;
    port[$03C4] := 2;
    port[$03C5] := 15;
     end
    else
     begin
      inicializar_modo(4);
      for j := 0 to 199 do
       for i := 0 to 39 do
        for k := 0 to 7 do
         begin
          direccion:= 320*j + i + 825;
          a := pantalla_ibm^[direccion] div (1 shl (7 - k)) mod 2;
          b := pantalla_ibm^[direccion+ 40] div (1 shl (7 - k)) mod 2;
          c := (a shl 1) + b;
          byte := c shl (2*(3 - k mod 4));
          if j mod 2 = 0 then
           inc(MEM[$B800:40*j + 2*i + k div 4],byte)
          else
           inc(MEM[$BA00:80*(j div 2) + 2*i + k div 4],byte);
         end;
     end;
   close(archivo_b);
   dispose(pantalla_ibm);
end;

function comprobar: boolean;
begin
 if longitud(nombre_archivo) = 64824 then
  comprobar:= true
 else
  comprobar:= false;
end;

procedure confirmar;
begin
```

```
  repeat
   tocado:= readkey;
   tocado:= upcase(tocado);
  until tocado in ['S','N',#13,#27];
end;

begin
 scr := -1;
 repeat
  ok := false;
  inc(scr);
  str(scr,nombre_archivo);
  if scr < 10 then
   nombre_archivo := '0' + nombre_archivo;
  nombre_archivo:= 'SCREEN'+nombre_archivo+'.LBM';
  if existe(nombre_archivo) and comprobar then
   begin
    writeln('archivo a convertir : ',nombre_archivo + ' ?');
    confirmar;
    if tocado in [#13,'S'] then
     ok := TRUE;
   end;
 until ok or (scr = 99);
 if SCR = 99 then
  begin
   writeln('paso del archivo a las normas MCGA/VGA..');
   halt;
  end;
 modo := detectar;
 inicializar_modo(3);
 if (modo >= 13) and ver_ega then
  begin
   writeln('convertir a formato EGA ?');
   confirmar;
   if tocado in ['S',#13] then
    ega:= true
```

```
    else
      ega:= false;
   end
  else
   ega:= false;
  if ega then
   writeln('-> EGA')
  else
   writeln('-> CGA');
  cambiar_formato;
end.
```


Todo el programa es una recapitulación de los conceptos vistos con anterioridad. La rutina comprobar_EGA, asegura de que se pueda emularse en EGA ya que casi todas las computadoras en nuestro tiempo ya tienen tarjetas VGA y posteriores.

Si usaremos este programa sobre una imagen VGA dibujada en blanco (valor 15 en el registro) sobre negro (0 del registro) la conversión se apegará a los colores, como se lustra en la tabla 10:

Pixel	Color	Tarjeta
00001111 (no truncado)	Blanco	VGA
1111 (truncado a 4)	Blanco	EGA
11 (truncado a 2)	Blanco	CGA

Tabla 10. Tabla de colores de la tarjeta VGA

Comprobaremos lo anterior, si revisamos la imagen SCREEN00.LBM incluido en los programas finales en disco. Pero este caso no es general. En efecto, supongamos que cada pixel de la imagen VGA es un matiz de gris. El número de registro del color correspondiente estará entre 16 y 31 si no se ha modificado la paleta del sistema. Tomemos el 26, examinemos la correspondencia entre el color y el juego de bits de menor peso correspondiente según esta forma de codificación:

Pixel	Color	Tarjeta
00011010 (no truncado)	gris	VGA
1010 (truncado a 4)	verde	EGA
10 (truncado a 2)	rojo (magenta)	CGA

Tabla 11. Correspondencia entre color y juego bits

Aunque el trazado de la imagen queda preservado, por otra parte, verificamos que los colores no. En el archivo SCREEN06.LBM el fin es un ejemplo de esta situación. El archivo original VGA está dibujado con una degradación de grises. La transformación en EGA o CGA provoca una deformación de la imagen original.

Con respecto a los colores, en función de la conversión en otros modos, habrá que seguir este método, crear una imagen en VGA con reglas estrictas y bien determinadas.

5.4 Algoritmo de conversión para imágenes VGA para ser vistas en EGA y CGA

El principio de conversión es el siguiente: Cada color EGA corresponde a un grupo de tres elementos RGB con 2 matices de intensidad por cada color, además del nivel cero. También cada color VGA se compone de un juego de componentes RGB pero con 63 matices de intensidad. El programa PALETVGA , permite ver estos niveles de RGB. El algoritmo siguiente, reducirá los 63 matices RGB de VGA a tres (3) solamente en EGA. El usuario seleccionará tres umbrales, podrá modificarlos al gusto y así reducir el rendimiento de la imagen. Debajo del primer umbral situado bastante bajo, ordinariamente 8 de los 63 son para los tres componentes, el pixel EGA queda en negro. Entre este umbral y el posterior, ordinariamente 28 de los 63 matices se toman los matices obscuros de los componentes RGB EGA.

Si se encuentran entre estos dos umbrales, el pixel será gris obscuro. Entre el umbral obscuro (28 de los 63) y el claro (situado en el 56 de los 63)

aproximadamente, se envían a la pantalla los matices claros de los componentes RGB EGA.

Si los tres componentes se sitúan entre estos dos umbrales, el pixel EGA, será gris claro.

En forma general, el pixel puesto en pantalla será el resultado de tres componentes para cada uno de los tres niveles: cero, matiz obscuro, matiz claro. Si los tres componentes RGB VGA son superiores al umbral claro, el pixel será de color claro. El programa ideal para ejemplificar lo anterior es CONVEREGA, que permite además, guardar la imagen obtenida en formato EGA.

La operación de conversión de la imagen EGA en CGA es mucho más simple. En este estado, no es posible respetar los colores, pero por lo menos habrá que respetar el trazado de la imagen. Se toma este principio de conversión basado en las temperaturas de los colores que se ven en el nivel de intensidad de los pixeles CGA de un monitor monocromático.

La tabla 12 muestra las equivalencias de algunos colores en los diferentes formatos tarjetas gráficas.

COLOR EGA	PALETA CGA 0	PALETA CGA 1	REGISTRO
Blanco o amarillo	Blanco	Amarillo	3
Magenta o rojo	magenta	Rojo	2
Cyan o verde	cyan	Verde	1
Azul o negro	negro	Negro	0

Tabla 12. Colores de los diferentes formatos de tarjetas gráficas

Hay que decir lo siguiente: los matices claros u obscuros de los colores EGA, se pierden. La imagen se degrada considerablemente, pero teniendo en cuenta la gran cantidad de computadoras personales de hoy con tarjetas gráficas VGA y posteriores, este algoritmo puede resultar de gran uso. El código completo en Turbo Pascal se presenta a continuación:

```
program convega;
uses
  crt, dos, iniciar;
```

```
type
 pantalla_ptr = ^tipo_pantalla;
 tipo_pantalla = array[0..63999] of byte;
var
 pantalla          : array[0..199,0..319] of byte absolute $A000:0;
 panta             : tipo_pantalla absolute $A000:0;
 salva_pantalla_ptr : pantalla_ptr;
 res               : registers;
 tocado            : char;

procedure leer_pantalla_de_disco(nombre_archivo : String);
var
 archivo : file of tipo_pantalla;
begin
 assign(archivo,nombre_archivo);
 reset(archivo);
 read(archivo,panta);
 close(archivo);
end;

procedure salvar_pantalla;
begin
 move(pantalla,salva_pantalla_ptr^,64000);
end;

procedure restaura_pantalla;
begin
 move(salva_pantalla_ptr^,pantalla,64000);
end;

procedure borrar_linea;
begin
 gotoxy(1,24);
 fillchar(pantalla[183,0],5440,0);
end;
```

```
procedure transforma_vga_en_ega;
type
 plano_ptr  = ^tipo_plano;
 tipo_plano  = array[0..3,0..7999] of byte;
var
 rvb0, r1, r2, b1, b2, v1, v2, rojo, verde, azul : byte;
 bidon, valor : integer;
 tabla : array[0..767] of byte;
 y, x, i : word;
 plano : plano_ptr;
 fplano : file of tipo_plano;
 nombre_archivo : string;

procedure coger_tabla_colores(numero,nombre : byte);
begin
 with res do
  begin
   ah := $10;
   al := $17;
   bx := numero;
   cx := nombre;
   es := Seg(tabla);
   dx := Ofs(tabla);
   intr($10,res);
  end;
end;
function escribir_bit(ea,plano,posicion,color : byte) : byte;
const
 colores : array[0..15] of array[0..3] of byte = ((0,0,0,0),
(1,0,0,0),(0,1,0,0),(1,1,0,0),(0,0,1,0),

 (1,0,1,0),(0,1,1,0),(1,1,1,0),(0,0,0,1),(1,0,0,1),(0,1,0,1),
                (1,1,0,1),(0,0,1,1),(1,0,1,1),(0,1,1,1),(1,1,1,1));
var
 ci, po : byte;
begin
```

```
  cl := colores[color,plano];
  po := 7 - posicion;
  if cl = 1 then
    escribir_bit := aa OR 1 shl po
  else
    escribir_bit := aa AND NOT ( 1 shl po );
end;

begin
  coger_tabla_colores(0,255);
  borrar_linea;
  write('umbral negro ( próximo a 8 ) -> ');
  readln(rvb0);
  borrar_linea;
  write('umbral sombra ( próximo a 28 ) -> ');
  readln(valor);
  r1 := valor;  v1 := valor;  b1 := valor;
  borrar_linea;
  write('umbral claro  ( próximo a 56 ) -> ');
  readln(valor);
  r2 := valor;  v2 := valor;  b2 := valor;
  restaura_pantalla;
  for y := 0 to 199 do
   for x := 0 to 319 do
     if pantalla[y,x] > 0 then
       begin
        rojo := tabla[3*pantalla[y,x]];
        verde  := tabla[3*pantalla[y,x] + 1];
        azul  := tabla[3*pantalla[y,x] + 2];
        if (rojo < rvb0) and (azul < rvb0) and (verde < rvb0) then
          pantalla[y,x] := 0
        else
          if rojo < r1 then
            begin
              if azul < b1 then
                begin
```

```
        if verde < v1 then
         pantalla[y,x] := 8
        else
         if verde < v2 then
          pantalla[y,x] := 2
         else
          pantalla[y,x] := 10;
       end
      else
       if azul < b2 then
        begin
         if verde < v1 then
          pantalla[y,x] := 1
         else
          if verde < v2 then
           pantalla[y,x] := 3
          else
           pantalla[y,x] := 10;
        end
       else
        begin
         if verde < v1 then
          pantalla[y,x] := 9
         else
          if verde < v2 then
           pantalla[y,x] := 11
          else
           pantalla[y,x] := 11;
        end;
  end
 else
  if rojo < r2 then
   begin
    if azul < b1 then
     begin
      if verde < v1 then
```

```
          pantalla[y,x] := 4
         else
          if verde < v2 then
           pantalla[y,x] := 6
          else
           pantalla[y,x] := 10;
        end
       else
        if azul < b2 then
         begin
          if verde < v1 then
           pantalla[y,x] := 5
          else
           if verde < v2 then
            pantalla[y,x] := 7
           else
            pantalla[y,x] := 10;
         end
        else
         begin
          if verde < v1 then
           pantalla[y,x] := 9
          else
           if verde < v2 then
            pantalla[y,x] := 11
           else
            pantalla[y,x] := 11;
         end;
    end
    else
    begin
       if azul < b1 then
        begin
         if verde < v1 then pantalla[y,x] := 12
         else
         if verde < v2 then pantalla[y,x] := 12
```

```
            else pantalla[y,x] := 14;
          end
          else
          if azul < b2 then
          begin
            if verde < v1 then pantalla[y,x] := 12
            else
            if verde < v2 then pantalla[y,x] := 13
            else pantalla[y,x] := 14;
          end
          else
          begin
            if verde < v1 then pantalla[y,x] := 13
            else
            if verde < v2 then pantalla[y,x] := 13
            else pantalla[y,x] := 15;
          end;
      end;
    end;
  salvar_pantalla;
  borrar_linea;
  write('Salvaguardar en formato EGA ?');
  tocado := readkey;
  if tocado in ['s','S',#13] then
   begin
   new(plano);
   borrar_linea;
   write('nombre del archivo de destino : ');
   readln(nombre_archivo);
   borrar_linea;
   write('un minuto de paciencia..');
   delay(1000);
  restaura_pantalla;

  for y := 0 to 199 do
  for x := 0 to 319 do
```

```
        for i := 0 to 3 do
         plano^[i,40*y + x div 8] :=escribir_bit(plano^[i,40*y + x div 8],i,x mod
8 ,pantalla[y,x]);
        assign(fplano,nombre_archivo);
        rewrite(fplano);
        write(fplano,plano^);
        close(fplano);
        dispose(plano);
         end
        else
        restaura_pantalla;
        end;

procedure transforma_EGA_en_CGA;
type
     pantalla_cga_ptr  = ^pantalla_cga_tipo;
     pantalla_cga_tipo = array[0..16383] of byte;
var
     y,x,i : word;
     salva_cga_ptr : pantalla_cga_ptr;
     fplano : file of pantalla_cga_tipo;
     nombre_archivo : string;
       begin
        for y := 0 to 199 do
         for x := 0 to 319 do
          if pantalla[y,x] > 0 then
          case pantalla[y,x] of
          14,15,6,7 : pantalla[y,x] := 15;
          12,13,4,5 : pantalla[y,x] := 13;
          10,11,2,3 : pantalla[y,x] := 11;
           8, 9, 1  : pantalla[y,x] := 0;
          end;
          salvar_pantalla;

       borrar_lineas;
       write('Salvaguardar en formato CGA ?');
```

```
        tocado := readkey;
        if tocado in ['s','S',#13] then
         begin
         new(salva_cga_ptr);
         borrar_linea;
         write('nombre del archivo de destino : ');
         readln(nombre_archivo);
         borrar_linea;
         write('un minuto de paciencia..');
         delay(1000);
         restaura_pantalla;

       for y := 0 to 99 do
       for x := 0 to 319 do
        salva_cga_ptr^[80*y + x div 4] := ((pantalla[2*y,x] DIV 2) MOD 4) SHL
6
          + ((pantalla[2*y,x+1] DIV 2) MOD 4) SHL 4+ ((pantalla[2*y,x+2] DIV
2) MOD 4) SHL 2
          + (pantalla[2*y,x+3] DIV 2) MOD 4;

       for y := 0 to 99 do
       for x := 0 to 319 do
        salva_cga_ptr^[80*y + x div 4 + 8192] := ((pantalla[2*y + 1,x] DIV 2)
MOD 4) SHL 6
          + ((pantalla[2*y + 1,x+1] DIV 2) MOD 4) SHL 4+((pantalla[2*y+ 1,x+2]
DIV 2) MOD 4) SHL 2
          + (pantalla[2*y + 1,x+3] DIV 2) MOD 4;
       assign(fplano,nombre_archivo);
       rewrite(fplano);
       write(fplano,salva_cga_ptr^);
       close(fplano);
       dispose(salva_cga_ptr);
        end
       else
       restaura_pantalla;
      end;
```

```
begin
 inicializar_modo(19);
  directvideo := false;
  new(salva_pantalla_ptr);
  leer_pantalla_de_disco('PALETA.VGA');
  salvar_pantalla;
  transforma_VGA_en_EGA;
  transforma_EGA_en_CGA;
end.
```

El código del programa es extenso, pero no se incluye nada nuevo. La función **escribir_bit**, calcula los bits con los que hay que cargar los cuatro planos en el formato EGA, a partir de la descomposición de los pixeles de la pantalla.

Cuando la pantalla esta almacenada en formato de imágen EGA, los pixeles son reducidos a los valores del 0 al 15. La transcripción es bastante rápida y su almacenamiento también.

5.5 *Programa para cualquier tipo de tarjeta gráfica*

Anteriormente, se ha visto el enfoque de todo lo concerniente a los formatos de archivos gráficos y de las características de conversión y visualización con una tarjeta diferente de la utilizada para crear el archivo gráfico. El siguiente programa esta dividido en tres partes, las cuales se describen a continuación:

1. Detecta la tarjeta gráfica e instala el modo 320x200 pixeles que corresponda al mayor número de colores.
2. Carga la imagen correspondiente al modo detectado. Lógicamente habrá que cargar una imagen VGA en formato LBM y restituir la imagen, tomando las 2, 4 u 8 líneas de bits, según el modo identificado.
3. Escribe y utiliza dos rutinas INVERTIR_IMAGEN Y SIMETRÍA, ambas están basadas en la llamada a la función 13 de la interrupción de video que permite leer el color de un punto puesto en pantalla. La primera, invierte la imagen y la segunda simetriza la imagen con respecto al punto central de la pantalla.

El código se muestra a continuación:

```
program simetria;
uses
 crt,  dos;
var
 nombre_archivo   : string;
 modo : byte;
 res : registers;
procedure inicializar_modo(valor : byte);
begin
 res.ah := 0;
 res.al := valor;
 intr($10,res);
end;
function detectar : byte;
function ver_modo : byte;
var
 tarjeta : byte;
begin
 with res do
  begin
   ah := $F;
   intr($10,res);
   tarjeta := al;
  end;
 ver_modo := tarjeta;
 if (tarjeta >= 13) and (MEM[$40:$88] = 0) then
  ver_modo := 0;
end;
var
 i,  j,  tarjeta : byte;
begin
 for i := 1 to 3 do
  case i of
   1 : begin
       inicializar_modo(19);
```

```
                if var_modo = 19 then
                 begin
                  detectar := 19;
                  exit;
                 end;
               end;
              2 : begin
              inicializar_modo(13);
              if var_modo = 13 then
                   begin
                    detectar := 13;
                    exit;
                   end;
             end;
              3 : begin
              inicializar_modo(4);
              if var_modo = 14 then
                   begin
                    detectar := 4;
                    exit;
                   end
                  else
                   begin
                    write('no he detectado tarjeta grafica');
                    halt;
                   end;
                end;
            end;
          end;
          procedure coger_a_Alberto;
          type
           plano_ptr = ^tipo_plano;
           tipo_plano = array[0..3,0..7999] of byte;
           pantalla_cga = array[0..16383] of byte;
           pantalla_virtual = array[0..63999] of byte;
          var
```

```
 plano : plano_ptr;
 archivo_ega : file of tipo_plano;
 panta_cga : pantalla_cga absolute $B800:$0000;
 archivo_cga : file of pantalla_cga;
 panta_vga : pantalla_virtual absolute $A000:$0000;
 archivo_vga : file of pantalla_virtual;
 p : byte;
begin
 case modo of
  4 : begin
       assign(archivo_cga,nombre_archivo);
       reset(archivo_cga);
       read(archivo_cga,panta_cga);
       close(archivo_cga);
      end;
 13: begin
      assign(archivo_ega,nombre_archivo);
      reset(archivo_ega);
      new(plano);
      read(archivo_ega,plano^);
      close(archivo_ega);
      for p := 0 to 3 do
       begin
        port[$3C4] := 2;
        port[$3C5] := 1 shl p;
    move(plano^[p],MEM[$A000:0000],8000);
       end;
port[$3C4] := 2;
port[$3C5] := 15;
    dispose(plano);
  end;
19: begin
    assign(archivo_vga,nombre_archivo);
    reset(archivo_vga);
    read(archivo_vga,panta_vga);
    close(archivo_vga);
```

```
            end;
          end;
        end;
        function leer_punto(x,y : integer) : byte;
        begin
         with res do
          begin
           ah := $D;
           dx := y;
           cx := x;
           bh := 0;
           intr($10,res);
           leer_punto := al;
          end;
       end;
       procedure escribir_punto(x,y,col : integer);
       begin
        with res do
         begin
          ah := $C;
          dx := y;
          cx := x;
          al := col;
          bh := 0;
          intr($10,res);
         end;
      end;
      procedure invertir_imagen;
      var
       x,
       y,
       i : byte;
      begin
       for y := 0 to 199 do
        for x := 0 to 159 do
         begin
```

```
        i := leer_punto(x,y);
        escribir_punto(x,y,leer_punto(319 - x,y));
        escribir_punto(319 - x,y,i);
      end;
  end;
  procedure simetrica;
  var
  x,  y : byte;
  begin
   for y := 0 to 199 do
     for x := 0 to 159 do
       escribir_punto(x,y,leer_punto(319 - x,y));
  end;
  begin
   modo := detectar;
   inicializar_modo(modo);
   case modo of
     4: nombre_archivo := '.CGA';
    13: nombre_archivo := '.EGA';
    19: nombre_archivo := '.MCG';
  end;
  nombre_archivo := 'ALBERTO' + nombre_archivo;
  coger_a_Alberto;
  invertir_imagen;
  simetrica;
end.
```

La imagen se ha transformado de tal forma que parece extraterrestre. Dado el número de llamadas a las funciones BIOS, el programa se ejecuta lentamente. Con una tarjeta VGA, se puede comprobar las velocidades en los tres modos. Basta con escribir desde el principio el modo que se desea en vez de identificarlo. Cuando es pasada por el BIOS, la visualización en modo 19 es más lenta ya que trabaja al nivel de bits y tiene la ventaja de manipular la imagen con 256 colores; esto es por regla general. Si se comprueba una mayor velocidad de visualización para los programas en modo 19 VGA, en particular y con respecto al modo EGA, es porque ya hemos acelerado el procedimiento mejorando al nivel del método. El punto opuesto de esta mejora, es la necesidad de crear rutinas básicas para cada tarjeta.

Este último capítulo no solo complementa los cuatro anteriores, sino que pone las bases para futuras creaciones software donde el elemento principal tanto de comunicación y procesamiento es visual. Siempre es importante recordarlo, UNA IMÁGEN DICE MAS QUE MIL PALABRAS...

CONCLUSIONES

Los archivos gráficos y las tarjetas que nos permiten mostrarlos van de la mano al momento de programar aunque sea de forma elemental. Este trabajo deja las bases para poder manipular estos elementos visuales por medio de las más elementales tarjetas gráficas. Inicia por conceptos básicos y a medida que avanza el desarrollo de los temas, los códigos y las explicaciones que acompañan a los mismos nos permiten asimilar tanto la forma y manera de programarlos con herramientas de desarrollo ampliamente utilizadas: **Turbo Pascal, ensamblador y C++.**

Este es solo el principio del largo recorrido que tendrán que pasar los usuarios deseosos de generar aplicaciones propias y poderosas. Aquí se mostró la forma de cómo manipular las arribos gráficos (ya sea creados por uno mismo o digitalizados) empleando la tarjeta VGA, CGA y EGA. Además se comparan las ventajas y desventajas de uno y otra tarjeta confeccionando códigos que realizarán las mismas acciones pero con diferentes criterios a fin de resaltar sus diferencias.

Cuando se trataba cada tarjeta se inicia con los conceptos elementales para la creación de animaciones rudimentarias, compactación y almacenamiento de archivos gráficos cada tema acompañado de un código listo para capturarse facilitando la asimilación de lo expuesto.

Lo que se puede generar después depende de la capacidad y destreza de quienes tomen en cuenta esta compilación de programas entre los que puede mencionar: la creación de juegos cada vez más atractivos a nuestras costumbres y necesidades, confección de sitios WEB que toman como base para todas sus operaciones a la tarjeta VGA, construcción de graficadores y todas aquellas aplicaciones donde intervengan imágenes contenidas en dispositivos de almacenamiento.

Francisco Cruz Mateos
UNAM-ARAGON 2002

Final

Si en los centros de formación y capacitación se integraran grupos de trabajo en la investigación, construcción de códigos y confección de aplicaciones donde todos los usuarios interesados pudieran extraer y aportar sus conocimientos, programas; creando un sitio en LA RED sería sensacional, puesto que no habría fronteras para el desarrollo de cualquier solución que se nos requiera. A cualquier hora y lugar podría el interesado aprender e intercambiar sus códigos a fin de conformar una gran biblioteca de poderosas aplicaciones basadas en lenguajes de desarrollo aparentemente ya pasados de moda pero que cobran vigencia cuando se requieren realizar "parches" a esos grandes programas de aplicación específica.

No se descubre el hilo negro, se abordan conceptos fundamentalísimos que hoy en día no todos manejan y toman en cuenta cuando se diseñan programas donde el elemento principal es visual (los gráficos).

Los cuatro objetivos que se plantearon al inicio de este trabajo han sido ampliamente cubiertos, tratando de ser lo más explícitamente claros en el lenguaje, algoritmos, códigos en lenguaje de alto nivel y ensamblador pero sobre todo en dotar al que consulta este trabajo de herramientas poderosas en su largo camino hacia aplicaciones visuales con gran aceptación por los elementos gráficos que se le incorporen.

BIBLIOGRAFÍA

DUNTEMANN, Jeff
"La Biblia del Turbo Pascal"
Editorial Anaya-Multimedia
Madrid, España 1990

HERGERT, Douglas
"Turbo Pascal 6.0 for PC"
Editorial SAMS
United States of America, 1991

HENNEFELD, Juliend
"Turbo Pascal con Aplicaciones 4.0-6.0"
Grupo Editorial Iberoamericana
Segunda Edición
México, D.F. 1992.

SHNEIDER, G.M. and C. Bruell Steven
"Programación avanzada y resolución de problemas con Turbo Pascal"
Editorial Anaya-Multimedia
Primera Edición
Madrid, España 1993.

RUGG, Tom and Feldman Phil
"Turbo Pascal (Biblioteca de Programas)"
Editorial Anaya-Multimedia
Segunda Edición
Madrid, España 1990.

TISHER, Michel
"Turbo Pascal 6.0"
Editorial Abacus
United States of America 1990.

JOEL, Goldstein Larry

"IBM PC y Compatibles"
Editorial Prentice Hall- Iberoamericana
Cuarta Edición
México, D.F. 1993

TISHER, Michael and JENDRICH, Bruno
"PC Interno 5.0"
Editorial Alfa-Omega
Primera Edición
Madrid, España 1996

CAIRÓ, Oswaldo
"Metodología de la Programación" Tomos I y II
Editorial Alfa-Omega
México, D.F. 1996

SCOTT, D. Palmer
"Domine Turbo Pascal "
Ediciones Ventura
México, D.F. 1992

PAREJA, Cristóbal y OJEDA, Manuel
"Desarrollo de algoritmos y técnicas de programación en Turbo Pascal"
Editorial RA-MA
Madrid, España 1997

R. BAJAR, Victoria
"Lenguaje Pascal"
Editorial Limusa
México, D.F. 1989

AMO, Alfonso y MORALES, Lozano
"Problemas de programación"
Editorial Paraninfo
Madrid, España 1990

EZZEL, Ben
"Using Turbo Pascal 6.0"
Editorial Addison-Wesley

United States of America 1990.

GERVAIS, Francois
"Programación de las tarjetas gráficas CGA, EGA, VGA"
Editorial Addison-Wesley Iberoamericana
Segunda Edición
United States of America 1995

WEISKAMP and HEINY
"Graficas poderosas con Turbo C++"
Editorial Megabyte Wiley Noriega editores
México, D.F. 1995