Programación Gráfica en C (Nivel Principiante)

Introducción

Este documento trata de explicar, a grandes rasgos, cómo funciona el dibujo de pixels bajo MS-DOS. En los tiempos que corren, es algo ya anticuado y prácticamente obsoleto, debido a que ya el 99% de los programas están hechos en Win95/98, y ahora con Windows 2000 y el próximo Windows ME, reducirán a 0 la programación bajo MS-DOS, pero, en mi opinión, y basándome en mi experiencia (poca), el manejo de gráficos es más "asequible" bajo MS-DOS. Esto es un curso de iniciación al manejo de memoria gráfica, pixels, RGB, etc., y no pretende ser exhaustivo, sino dar unas líneas generales sobre cómo se hace, con sus ventajas e inconvenientes respecto a Windows. Se dan por supuesto algunos conocimientos de C (y saber que es el Assembler), ya que los ejemplos serán fragmentos de código, no listados completos. El 95% de lo comentado aquí lo he comprobado yo mismo, y funciona, y se puede decir que es un pequeño recopilatorio de mis "andaduras" con la VGA desde que comenzé a programar en C.

Una nota más, emplearé abundantes tecnicismos en inglés (sin traducción ni acentos), asique lo siento por los amantes de la lengua de Cervantes :)

Espero que sirva de ayuda a todo aquel nostálgico, programador principiante o fanático del DOS. No soy un experto redactor, así que si algo no está bien explicado, mis disculpas por adelantado.

Un saludo,

- KaRt0nEs

Un poco de historia

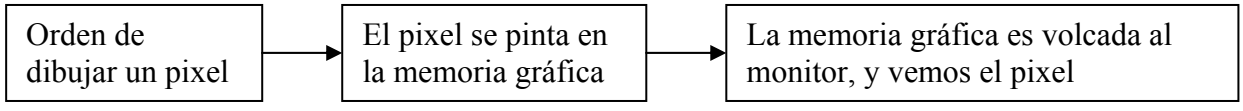
No puedo empezar sin hacer un breve resumen antes de la evolución de las tarjetas gráficas desde los primeros PCs.

Cuando salieron los primeros PCs (8086 y similares), los monitores eran monocromos o de escasos colores (4, 8 o a lo sumo 16). Eran los tiempos de las tarjetas CGA y EGA, lentas y muy simples. Con la llegada de los 286 y los 386, aparecieron las VGA, que soportaban 256 colores, algo más que de sobra para gráficos, juegos, etc. en esa época. Supusieron una revolución no sólo por esto, sino porque, aunque aparentemente soportaran pocos modos de vídeo (320x200 y 640x480 en modo gráfico son los más conocidos, aparte del clásico 80x25 del MS-DOS), los Hackers pronto descubrieron que podían manipularlas para conseguir modos gráficos no-nativos, como el famoso 320x240. Esto permitía jugar con muchos parámetros, como velocidades de refresco, colores, resoluciones, etc. Se generalizó el modo de 320x240 para los juegos, ya que si nos fijamos, esta resolución si mantiene el ratio (320x240, 640x480, 800x600,etc.) para que los pixels se vean cuadrados (en 320x200 se ven "achatados"). Debido a la velocidad de estos modos gráficos de baja resolución y a las optimizaciones que se han ido realizando, ha sido el estandar hasta la aparición de Windows 95.

Funcionamiento de la VGA. Comparación con el GUI de Windows

Cuando vemos una imagen en la pantalla, estamos viendo una copia de la imagen real. La imagen no se dibuja directamente en el CRT (Tubo de Rayos Catódicos) del monitor, sino que primero se pinta en la memoria de la tarjeta, y posteriormente se vuelca el contenido al monitor (a una velocidad que ni nos damos cuenta, a no ser que nuestros programas sean terriblemente lentos).

Esquema:



Hay que tener en cuenta otras variables, como el refresco del monitor, a la hora de conseguir velocidad, pero no entraremos en estos detalles (eso se lo dejo a los programadores de Demos y Juegos) salvo que entre en las optimizaciones que utilice como ejemplos.
La memoria gráfica se utiliza como la memoria normal, aunque antes hay que cambiar el modo de vídeo a la VGA. El tratamiento del modo gráfico es simple y tosco, ya que el MS-DOS no distingue entre píxels, caracteres o valores. Trata todo como datos, lo que nos da una ventaja, y es el poder manipularlos a nuestra voluntad sin restricciones, pero también nos obliga a tener en cuenta en cada momento lo que hacemos, y a llevar un control estricto de todas nuestras acciones, ya que podemos sobreescribir zonas de memoria que no pertenecen a la tarjeta gráfica.
En Windows, esto cambia en parte, ya que podemos simplemente indicar al GUI que queremos pintar un rectángulo con unas características determinadas, o dibujar una ventana que ocupe ¼ de la pantalla (en MS-DOS habría que "fabricar" un generador y manipulador de ventanas para hacer lo mismo). El problema de Windows es que su sistema de dibujo de primitivas se antoja terrible para un juego o demo, por lo que hay que echar mano de los DirectX (lo que supone aprender a manejar dichos controladores). En MS-DOS es un trabajo duro, ya que hay que partir desde cero prácticamente, pero tiene sus recompensas, como el hacer librerías gráficas personalizadas y optimizadas (¿a quien le importa el manejo de ventanas en un juego de conducción, o el usar primitivas de elipses, si lo que queremos es dibujar sprites lo más rápido posible?).

Algunos detalles previos

Direccionamiento de memoria:

Aquí no entro mucho en detalle, simplemente explico que se refiere a las direcciones según la nomenclatura SEGMENTO:OFFSET, siendo el SEGMENTO la "zona" de memoria (VGA, impresora, bios, etc.) y OFFSET las distintas "parcelas" o celdas de cada SEGMENTO.

Tipos de variables:

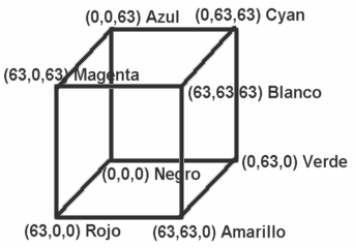
El lenguaje C dispone de tipos de variable enteros con y sin signo. Debido a que tanto el Assembler como el ordenador trabajan sin signo, todas las variables deben de ser definidas como unsigned int (WORD en ASM) y unsigned char (BYTE). Aquí hay una tabla de los diferentes valores que toman según uno u otro caso:

Con signo:	Sin signo:
Int: -32768 a 32767	unsigned int: 0 a 65535
Char: -128 a 127	unsigned char: 0 a 255

Como sabrás, el modo 320x200 tiene 256 colores, luego no nos sirve el char, y usar el int es gastar dos bolsas para cargar lo que cabe en una sola (además de que puede provocar problemas, ya que el registro en el que se almacena el color es de tipo BYTE). Como también supondrás, ya que 320x200 son 64000 y el unsigned int llega hasta 65535, éste último se convierte en el tipo de variable idóneo para almacenar el OFFSET.

RGB. Limitaciones de la VGA

Los colores se definen mediante 3 valores, los componentes Red, Green y Blue (Rojo, Verde y Azul) o comúnmente llamados RGB. En la VGA de 8 bits de color (o sea, 256 colores), la "pigmentación" de cada color es un BYTE, que tiene 256 combinaciones posibles. Como son tres componentes, pero los ordenadores funcionan con potencias de dos, 256/4 = 64 posibles valores para cada componente RGB (si nos fijamos, estamos perdiendo 64 valores para un hipotético cuarto componente). Con esto tenemos una paleta de 256 colores a elegir entre 262144 posibles, lo que para juegos no está mal, pero para imágenes se queda algo escaso (Windows y las tarjetas nuevas lo han remediado con los 16, 24 y 32 bits de color, que permiten millones de colores a la vez).



Los primeros pasos. Cambiar el modo de vídeo y pintar un pixel sin optimizaciones

El modo gráfico correspondiente a 320x200x256 en hexadecimal es 0x013 (usaré la notación hexadecimal del C, al menos hasta las optimizaciones en Assembler), y el del MS-DOS 0x03. Para cambiar el modo de vídeo es necesario utilizar las interrupciones de la BIOS, cuyo nº es el 0x010 (el de las interrupciones de MS-DOS es el 0x021).

En C, para llamar a las interrupciones, se utiliza la función int86, y antes hay que definir la estructura de datos con los registros necesarios para cambiar el modo de vídeo. Una típica función de cambio de modo de vídeo sería la siguiente:

```
Void modo_video (unsigned char MODO)
{
  union REGS regs;
  regs.h.ah = 0;
  regs.h.al = MODO;
  int86 (0x10, &regs, &regs);
}
```

A dicha función le pasaríamos como parámetro entre paréntesis el modo deseado (0x13 en este caso). El valor 0 en el registro AH indica a la BIOS que deseamos cambiar el modo de vídeo, y el registro AL debe almacenar el nuevo modo. int86 hace la llamada a la BIOS y le pasa los parámetros.

Una vez en la VGA estandar, para poder pintar pixels, hay que dibujarlos en la memoria de la VGA, cuya dirección comienza en A000:0000. Como el MS-DOS trata los pixels como valores de la memoria, no es posible de primeras el situar un pixel en la posición x=3 y=5. En realidad, esta posición sería la dirección A000:063D. ¿Lioso? Pues no pasa nada, porque para eso están las matemáticas. Veamos, si multiplicamos 320x200 nos da 64000 (FA00 en Hex.), luego las posiciones de memoria van desde A000:0000 hasta A000:FA00. Bien. Como sabemos que hay 320 columnas y 200 filas, con la fórmula Dir = (320 x Y) + X ya podemos saber a que OFFSET se corresponde la posición que deseemos.
Con la función poke podemos dibujar nuestros primeros pixels. Ejemplo de función:

```
Void pinta_pixel (int x, int y, unsigned char color)
{
  unsigned int DIR;
  DIR = (320*y)+x;
  Poke (A000, DIR, color);
}
```

Esta función pinta un pixel según las coordenadas X e Y y con el color desdeado.

Optimización de las rutinas. Otras primitivas para el manejo de píxeles (ya optimizadas)
Bueno, ya sabemos cómo se dibujan pixels en la VGA. Ahora intenta rellenar toda la pantalla de un color (con un bucle for, por ejemplo). ¿Qué, algo lento? Tanto si te lo parece como si no, te aseguro que es una forma muy "cutre" de diseñar un juego. Las instrucciones en C son lentas, muy lentas comparadas con las de Assembler (y algo lentas que las de Pascal, pero si quieres sencillez de manejo, algún precio tienes que pagar), pero todo tiene arreglo en esta vida. En C, podemos meter fragmentos de código escritos en ASM (Assembler), y, aunque no consigamos la misma velocidad, la mejora es impresionante (del orden de unas 8 veces más rápido). Se implementan añadiendo al principio de la línea asm (si vamos a poner varias líneas se pueden utilizar llaves para mayor comodidad). Pues bien, algún genio del Assembler le dio al coco y descubrió que era posible ahorrarse las multiplicaciones (operaciones que consumen bastantes ciclos de reloj), sustituyendolas por operaciones binarias de desplazamiento de Bits. El nº 7 en binario es 00000111, y como todos sabemos (¿o acaso no sabes nada de álgebra de boole, nºs binarios y potencias de base 2?), si desplazamos todos los bytes a la izquiera 1 posición es lo mismo que multiplicar por 2 (quedaría 00001110, que en decimal es 14), y si desplazamos a la derecha, dividimos por 2. Pues con esta culturilla general y algunos conocimientos más, aquí tenemos la función optimizada en ASM para dibujar un pixel en 320x200:

```
Void pixel (unsigned int COR_X, unsigned INT COR_Y, unsigned
char COLOR)
{
  asm {
  MOV AX, 0A000H
  MOV ES, AX
  MOV AL, COLOR
  MOV BX, COR_X
  MOV DX, COR_Y
  XCHG DH, DL
  MOV DI, DX
  SHR DI, 1
  SHR DI, 1
  ADD DI, DX
  ADD DI, BX
  MOV ES:[DI], AL
  }
}
```

Parece más larga, pero es muchísimo más rápida (las instrucciones utilizadas consumen poquísimos ciclos de reloj, y realiza algunas "trampas" matemáticas).

La función para seleccionar el modo de vídeo optimizada es bastante similar a la original, pero en ASM, para que vaya un poco más rápida:

```
Void modo_video (unsigned char MODO)
{
  asm MOV AH, 0
  asm MOV AL, MODO
  asm INT 10H
}
```

Ahora bien, te preguntarás: "muy bien, ya puedo pintar pixels, pero luego, ¿cómo se el color del pixel de la posición (310, 150)?", pues nada, alla vamos. La función sin implementar es sencilla. Simplemente hay que saber leer registros de la memoria y buscar en la dirección deseada el valor, que corresponderá al color del pixel (aquí lo dejo a los conocimientos de cada uno). La función optimizada (que también utiliza los truquitos anteriormente citados para evitar la multiplicación al hallar la posición de memoria) es la siguiente:

```
Unsigned char lee_pixel (unsigned int COR_X, unsigned char
COR_Y)
{
  unsigned char COLOR
  asm
  {
  MOV AX, 0A000H
  MOV ES, AX
  MOV BX, COR_X
  MOV DX, COR_Y
  XCHG DH, DL
  MOV DI, DX
  SHR DI, 1
  SHR DI, 1
  ADD DI, DX
  ADD DI, BX
  MOV COLOR, ES:[DI]
  }
  return (COLOR)
  }
```

La función retorna como valor el color del pixel.

Bueno, ya tenemos casi una librería para empezar. Sólo falta un último detalle. ¿Qué pasa si no nos gustan los 256 colores de la paleta que por defecto tiene el MS-DOS, o si queremos hacer un precioso mar con muchos tonos de azules y verdes?. Pues bien, como ya he explicado antes, podemos cambiar los componentes RGB para hacer nuevas paletas a nuestro gusto. La función para cambiar lo RGB de un color determinado es la siguiente:

```
Void color_rgb (unsigned char COLOR, unsigned char R, unsigned
char G, unsigned char B)
{
  asm {
  MOV AL, COLOR
  MOV DX, 3C8H
  OUT DX, AL
  INC DX
  MOV AL, R
  OUT DX, AL
  MOV AL, G
  OUT DX, AL
  MOV AL, B
  OUT DX, AL
  }
}
```

Conclusiones

Con esto termina mi “cursillo” rápido sobre programación gráfica en C. Con esto sólo he abierto las puertas del mundo que es el manejo de las tarjetas y los gráficos en los PCs. No he tocado nada de efectos como fundidos, dibujo en 3D y polígonos, manejo de sprites, importación de gráficos (GIFs, BMPs, PCXs, etc.) ni nada por el estilo. Esto es sólo una iniciación. Ahora es el turno de indagar e investigar como hacer una rutina para dibujar círculos o elipses, y cómo se puede optimizar para ir más rápido, ocupar menos, etc.

La base no es nada mala; las rutinas optimizadas son rutinas semi-profesionales, obtenidas de librerias muy completas y rápidas usadas en el mundo de la Demoscene.