Chapter 33

Yogi Bear and Eurythmics Confront VGA Colors

Chapter 33

The Basics of VGA Color Generation

Kevin Mangis wants to know about the VGA's 4bit to 8-bit to 18-bit color translation. Mansur Loloyan would like to find out how to generate a look-up table containing 256 colors and how to change the default color palette. And surely they are only the tip of the iceberg; hordes of screaming programmers from every corner of the planet are no doubt tearing the place up looking for a discussion of VGA color, and venting their frustration at my mailbox. *Let's have* it, they've said, clearly and in considerable numbers. As Eurythmics might say, who is this humble writer to disagree?

On the other hand, I hope you all know what you're getting into. To paraphrase Yogi, the VGA is smarter (and more confusing) than the average board. There's the basic 8-bit to 18-bit translation, there's the EGA-compatible 4bit to 6-bit translation, there's the 2- or 4bit color paging register that's used to pad 6- or 4bit pixel values out to 8 bits, and then there's 256-color mode. Fear not, it will all make sense in the end, but it may take us a couple of additional chapters to get there—so let's get started.

Before we begin, though, I must refer you to Michael Covington's excellent article, "Color Vision and the VGA," in the June/July 1990 issue of *PC TECHNIQUES*. Michael, one of the most brilliant people it has ever been my pleasure to meet, is an expert in many areas I know nothing about, including linguistics and artificial intelligence. Add to that list the topic of color perception, for his article superbly describes the mechanisms by which we perceive color and ties that information to the VGA's capabilities. After reading Michael's article, you'll understand what colors the VGA is capable of generating, and why.

Our topic in this chapter complements Michael's article nicely. Where he focused on color perception, we'll focus on color generation; that is, the ways in which the VGA can be programmed to generate those colors that lie within its capabilities. To find out why a VGA can't generate as pure a red as an LED, read Michael's article. If you want to find out how to flip between 16 different sets of 16 colors, though, don't touch that dial!

I would be remiss if I didn't point you in the direction of two more articles, these in the July 1990 issue of *Dr. Dobb's Journal.* "Super VGA Programming," by Chris Howard, provides a good deal of useful information about SuperVGA chipsets, modes, and programming. "Circles and the Digital Differential Analyzer," by Tim Paterson, is a good article about fast circle drawing, a topic we'll tackle soon. All in all, the dog days of 1990 were good times for graphics.

VGA Color Basics

Briefly put, the VGA color translation circuitry takes in one 4- or 8-bit pixel value at a time and translates it into three 6-bit values, one each of red, green, and blue, that are converted to corresponding analog levels and sent to the monitor. Seems simple enough, doesn't it? Unfortunately, nothing is ever that simple on the VGA, and color translation is no exception.

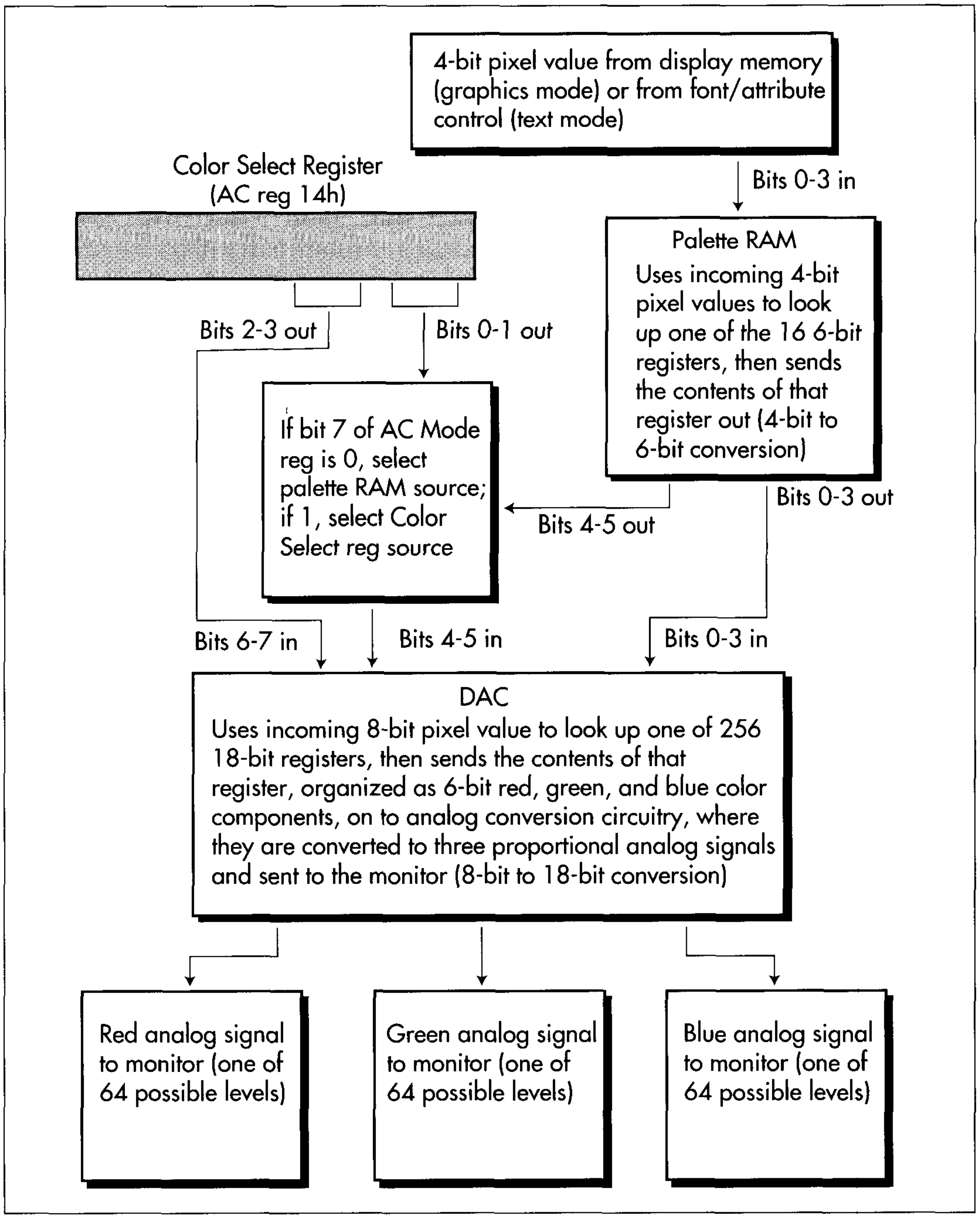
The Palette RAM

The color path in the VGA involves two stages, as shown in Figure 33.1. The first stage fetches a 4-bit pixel from display memory and feeds it into the EGA-compatible palette RAM (so called because it is functionally equivalent to the palette RAM color translation circuitry of the EGA), which translates it into a 6-bit value and sends it on to the DAC. The translation involves nothing more complex than the 4-bit value of a pixel being used as the address of one of the 16 palette RAM registers; a pixel value of 0 selects the contents of palette RAM register 0, a pixel value of 1 selects register 1, and so on. Each palette RAM register stores 6 bits, so each time a palette RAM register is selected by an incoming 4-bit pixel value, 6 bits of information are sent out by the palette RAM. (The operation of the palette RAM was described back in Chapter 29.)

The process is much the same in text mode, except that in text mode each 4-bit pixel value is generated based on the character's font pattern and attribute. In 256-color mode, which we'll get to eventually, the palette RAM is not a factor from the programmer's perspective and should be left alone.

The DAC

Once the EGA-compatible palette RAM has fulfilled its karma and performed 4-bit to 6-bit translation on a pixel, the resulting value is sent to the DAC (Digital/Analog Converter). The DAC performs an 8-bit to 18-bit conversion in much the same manner as the palette RAM, converts the 18-bit result to analog red, green, and blue


4-bit pixel value from display memory (graphics mode) or from font/attribute control (text mode)
Bits 0-3 in
Color Select Register (AC reg 14h)
Palette RAM
Uses incoming 4-bit pixel values to look up one of the 16 6-bit registers, then sends the contents of that register out (4-bit to 6-bit conversion)
Bits 2-3 out
Bits 0-1 out
If bit 7 of AC Mode reg is 0, select palette RAM source; if 1, select Color Select reg source
Bits 4-5 out
Bits 0-3 out
Bits 6-7 in
Bits 4-5 in
Bits 0-3 in
DAC
Uses incoming 8-bit pixel value to look up one of 256 18-bit registers, then sends the contents of that register, organized as 6-bit red, green, and blue color components, on to analog conversion circuitry, where they are converted to three proportional analog signals and sent to the monitor (8-bit to 18-bit conversion)
Red analog signal to monitor (one of 64 possible levels)
Green analog signal to monitor (one of 64 possible levels)
Blue analog signal to monitor (one of 64 possible levels)

The VGA color generation path.
Figure 33.1

signals (6 bits for each signal), and sends the three analog signals to the monitor. The DAC is a separate chip, external to the VGA chip, but it's an integral part of the VGA standard and is present on every VGA.

(I'd like to take a moment to point out that you can't speak of "color" at any point in the color translation process until the output stage of the DAC. The 4-bit pixel values in memory, 6-bit values in the palette RAM, and 8-bit values sent to the DAC are all attributes, not colors, because they're subject to translation by a later stage. For example, a pixel with a 4-bit value of 0 isn't black, it's attribute 0. It will be translated to 3FH if palette RAM register 0 is set to 3FH, but that's not the color white, just another attribute. The value 3FH coming into the DAC isn't white either, and if the value stored in DAC register 63 is red=7, green=0, and blue=0, the actual *color* displayed for that pixel that was 0 in display memory will be dim red. It isn't color until the DAC says it's color.)

The DAC contains 256 18-bit storage registers, used to translate one of 256 possible 8-bit values into one of 256K (262,144, to be precise) 18-bit values. The 18-bit values are actually composed of three 6-bit values, one each for red, green, and blue; for each color component, the higher the number, the brighter the color, with 0 turning that color off in the pixel and 63 (3FH) making that color maximum brightness. Got all that?

Color Paging with the Color Select Register

"Wait a minute," you say bemusedly. "Aren't you missing some bits between the palette RAM and the DAC?" Indeed I am. The palette RAM puts out 6 bits at a time, and the DAC takes in 8 bits at a time. The two missing bits—bits 6 and 7 going into the DAC—are supplied by bits 2 and 3 of the Color Select register (Attribute Controller register 14H). This has intriguing implications. In 16-color modes, pixel data can select only one of 16 attributes, which the EGA palette RAM translates into one of 64 attributes. Normally, those 64 attributes look up colors from registers 0 through 63 in the DAC, because bits 2 and 3 of the Color Select register are both zero. By changing the Color Select register, however, one of three other 64 color sets can be selected instantly. I'll refer to the process of flipping through color sets in this manner as *color paging.*

That's interesting, but frankly it seems somewhat half-baked; why bother expanding 16 attributes to 64 attributes before looking up the colors in the DAC? What we'd *really* like is to map the 16 attributes straight through the palette RAM without changing them and supply the upper *4* bits going to the DAC from a register, giving us 16 color pages. As it happens, all we have to do to make that happen is set bit 7 of the Attribute Controller Mode register (register 10H) to 1. Once that's done, bits 0 through 3 of the Color Select register go straight to bits 4 through 7 of the DAC, and only bits 3 through 0 coming out of the palette RAM are used; bits 4 and 5 from the palette RAM are ignored. In this mode, the palette RAM effectively contains 4-bit, rather than 6-bit, registers, but that's no problem because the palette RAM will be programmed to pass pixel values through unchanged by having register 0 set to 0, register 1 set to 1, and so on, a configuration in which the upper two bits of all the palette RAM registers are the same (zero) and therefore irrelevant. As a matter of fact, you'll generally want to set the palette RAM to this pass-through state when working with VGA color, whether you're using color paging or not.

Why is it a good idea to set the palette RAM to a pass-through state? It's a good idea because the palette RAM is programmed by the BIOS to EGA-compatible settings and the first 64 DAC registers are programmed to emulate the 64 colors that an EGA can display during mode sets for 16-color modes. This is done for compatibility with EGA programs, and it's useless if you're going to tinker with the VGA's colors. As a VGA programmer, you want to take a 4-bit pixel value and turn it into an 18-bit RGB value; you can do that without any help from the palette RAM, and setting the palette RAM to pass-through values effectively takes it out of the circuit and simplifies life something wonderful. The palette RAM exists solely for EGA compatibility, and serves no useful purpose that I know of for VGA-only color programming.

256-Color Mode

So far I've spoken only of 16-color modes; what of 256-color modes?

The rule in 256-color modes is: *Don't tinker with the VGA palette.* Period. You can select any colors you want by reprogramming the DAC, and there's no guarantee as to what will happen if you mess around with the palette RAM. There's no benefit that I know of to changing the palette RAM in 256-color mode, and the effect may vary from VGA to VGA. So don't do it unless you know something I don't.

On the other hand, feel free to alter the DAC settings to your heart's content in 256-color mode, all the more so because this is the only mode in which all 256 DAC settings can be displayed simultaneously. By the way, the Color Select register and bit 7 of the Attribute Controller Mode register are ignored in 256-color mode; all 8 bits sent from the VGA chip to the DAC come from display memory. Therefore, there is no color paging in 256-color mode. Of course, that makes sense given that all 256 DAC registers are simultaneously in use in 256-color mode.

Setting the Palette RAM

The palette RAM can be programmed either directly or through BIOS interrupt 10H, function 10H. I strongly recommend using the BIOS interrupt; a clone BIOS may mask incompatibilities with genuine IBM silicon. Such incompatibilities could include anything from flicker to trashing the palette RAM; or they may not exist at all, but why find out the hard way? My policy is to use the BIOS unless there's a clear reason not to do so, and there's no such reason that I know of in this case.

When programming specifically for the VGA, the palette RAM needs to be loaded only once, to store the pass-through values 0 through 15 in palette RAM registers 0 through 15. Setting the entire palette RAM is accomplished easily enough with subfunction 2 (AL=2) of function 10H (AH=10H) of interrupt 10H. A single call to this subfunction sets all 16 palette RAM registers (and the Overscan register) from a block of 17 bytes pointed to by ES:DX, with ES:DX pointing to the value for register 0, ES:DX+1 pointing to the value for register 1, and so on up to ES:DX+16, which points to the overscan value. The palette RAM registers store 6 bits each, so only the lower 6 bits of each of the first 16 bytes in the 17-byte block are significant. (The Overscan register, which specifies what's displayed between the area of the screen that's controlled by the values in display memory and the blanked region at the edges of the screen, is an 8-bit register, however.)

Alternatively, any one palette RAM register can be set via subfunction 0 (AL=0) of function 10H (AH=10H) of interrupt 10H. For this subfunction, BL contains the number of the palette RAM register to set and the lower 6 bits of BH contain the value to which to set that register.

Having said that, let's leave the palette RAM behind (presumably in a pass-through state) and move on to the DAC, which is the right place to do color translation on the VGA.

Setting the DAC

Like the palette RAM, the DAC registers can be set either directly or through the BIOS. Again, the BIOS should be used whenever possible, but there are a few complications here. My experience is that varying degrees of flicker and screen bounce occur on many VGAs when a large block of DAC registers is set through the BIOS. That's not a problem when the DAC is loaded just once and then left that way, as is the case in Listing 33.1, which we'll get to shortly, but it can be a serious problem when the color set is changed rapidly ("cycled") to produce on-screen effects such as rippling colors. My (limited) experience is that it's necessary to program the DAC directly in order to cycle colors cleanly, although input from readers who have worked extensively with VGA color is welcome.

At any rate, the code in this chapter will use the BIOS to set the DAC, so I'll describe the BIOS DAC-setting functions next. Later, I'll briefly describe how to set both the palette RAM and DAC registers directly, and I'll return to the topic in detail in an upcoming chapter when we discuss color cycling.

An individual DAC register can be set by interrupt 10H, function 10H (AH=10), subfunction 10H (AL=10H), with BX indicating the register to be set and the color to which that register is to be set stored in DH (6-bit red component), CH (6-bit green component), and CL (6-bit blue component).

A block of sequential DAC registers ranging in size from one register up to all 256 can be set via subfunction 12H (AL=12H) of interrupt 10H, function 10H (AH=10H). In this case, BX contains the number of the first register to set, CX contains the number of registers to set, and ES:DX contains the address of a table of color entries to which DAC registers BX through BX+CX-1 are to be set. The color entry for each

DAC register consists of three bytes; the first byte is a 6-bit red component, the second byte is a 6-bit green component, and the third byte is a 6-bit blue component, as illustrated by Listing 33.1.

If You Can't Call the BIOS, Who Ya Gonna Call?

Although the palette RAM and DAC registers should be set through the BIOS whenever possible, there are times when the BIOS is not the best choice or even a choice at all; for example, a protected-mode program may not have access to the BIOS. Also, as mentioned earlier, it may be necessary to program the DAC directly when performing color cycling. Therefore, I'll briefly describe how to set the palette RAM and DAC registers directly; in Chapter A on the companion CD-ROM I'll discuss programming the DAC directly in more detail.

The palette RAM registers are Attribute Controller registers 0 through 15. They are set by first reading the Input Status 1 register (at 3DAH in color mode or 3BAH in monochrome mode) to reset the Attribute Controller toggle to index mode, then loading the Attribute Controller Index register (at 3C0H) with the number (0 through 15) of the register to be loaded. Do *not* set bit 5 of the Index register to 1, as you normally would, but rather set bit 5 to 0. Setting bit 5 to 0 allows values to be written to the palette RAM registers, but it also causes the screen to blank, so you should wait for the start of vertical retrace before loading palette RAM registers if you don't want the screen to flicker. (Do you see why it's easier to go through the BIOS?) Then, write the desired register value to 3C0H, which has now toggled to become the Attribute Controller Data register. Write any desired number of additional register number/register data pairs to 3C0H, then write 20H to 3C0H to unblank the screen.

The process of loading the palette RAM registers depends heavily on the proper sequence being followed; if the Attribute Controller Index register or index/data toggle data gets changed in the middle of the loading process, you'll probably end up with a hideous display, or no display at all. Consequently, for maximum safety you may want to disable interrupts while you load the palette RAM, to prevent any sort of interference from a TSR or the like that alters the state of the Attribute Controller in the middle of the loading sequence.

The DAC registers are set by writing the number of the first register to set to the DAC Write Index register at 3C8H, then writing three bytes—the 6-bit red component, the 6-bit green component, and the 6-bit blue component, in that order—to the DAC Data register at 3C9H. The DAC Write Index register then autoincrements, so if you write another three-byte RGB value to the DAC Data register, it'll go to the next DAC register, and so on indefinitely; you can set all 256 registers by sending 256*3 = 768 bytes to the DAC Data Register.

Loading the DAC is just as sequence-dependent and potentially susceptible to interference as is loading the palette, so my personal inclination is to go through the whole process of disabling interrupts, loading the DAC Write Index, and writing a three-byte RGB value separately for each DAC register; although that doesn't take advantage of the autoincrementing feature, it seems to me to be least susceptible to outside influences. (It would be even better to disable interrupts for the entire duration of DAC register loading, but that's much too long a time to leave interrupts off.) However, I have no hard evidence to offer in support of my conservative approach to setting the DAC, just an uneasy feeling, so I'd be most interested in hearing from any readers.

A final point is that the process of loading both the palette RAM and DAC registers involves performing multiple **OUT**s to the same register. Many people whose opinions I respect recommend delaying between I/O accesses to the same port by performing a **JMP $+2** (jumping flushes the prefetch queue and forces a memory access—or at least a cache access—to fetch the next instruction byte). In fact, some people recommend two **JMP $+2** instructions between I/O accesses to the same port, and *three* jumps between I/O accesses to the same port that go in opposite directions (**OUT** followed by **IN** or **IN** followed by **OUT**). This is clearly necessary when accessing some motherboard chips, but I don't know how applicable it is when accessing VGAs, so make of it what you will. Input from knowledgeable readers is eagerly solicited.

In the meantime, if you can use the BIOS to set the DAC, do so; then you won't have to worry about the real and potential complications of setting the DAC directly.

An Example of Setting the DAC

This chapter has gotten about as big as a chapter really ought to be; the VGA color saga will continue in the next few. Quickly, then, Listing 33.1 is a simple example of setting the DAC that gives you a taste of the spectacular effects that color translation makes possible. There's nothing particularly complex about Listing 33.1; it just selects 256-color mode, fills the screen with one-pixel-wide concentric diamonds drawn with sequential attributes, and sets the DAC to produce a smooth gradient of each of the three primary colors and of a mix of red and blue. Run the program; I suspect you'll be surprised at the stunning display this short program produces. Clever color manipulation is perhaps the easiest way to produce truly eye-catching effects on the PC.

LISTING 33.1 L33-1.ASM

```
; Program to demonstrate use of the DAC registers by selecting a
; smoothly contiguous set of 256 colors, then filling the screen
; with concentric diamonds in all 256 colors so that they blend
; into one another to form a continuum of color.
;
     .model     small
     .stack     200h
     .data

; Table used to set all 256 DAC entries.
;
; Table format:
;    Byte 0: DAC register 0 red value
;    Byte 1: DAC register 0 green value
;     Byte 2: DAC register 0 blue value
;     Byte 3: DAC register 1 red value
;     Byte 4: DAC register 1 green value
;     Byte 5: DAC register 1 blue value
;     :
;     Byte 765: DAC register 255 red value
;     Byte 766: DAC register 255 green value
;     Byte 767: DAC register 255 blue value

ColorTable  label byte

; The first 64 entries are increasingly dim pure green.
X=0
      REPT  64
      db    0,63-X,0
X=X+1
      ENDM

; The next 64 entries are increasingly strong pure blue.
X=0
      REPT  64
      db    0,0,X
X=X+1
      ENDM

; The next 64 entries fade through violet to red.
X=0
      REPT  64
      db    X,0,63-X
X=X+1
      ENDM

; The last 64 entries are increasingly dim pure red.
X=0
      REPT  64
      db    63-X,0,0
X=X+1
      ENDM

        .code
Start:
      mov   ax,0013h                  ;AH=0 selects set mode function,
                                      ; AL=13h selects 320x200 256-color
      int   10h                       ; mode

                                      ;load the DAC registers with the
                                      ; color settings
      mov   ax,@data                  ;point ES to the default
      mov   es,ax                     ; data segment
      mov   dx,offset ColorTable
                                      ;point ES:DX to the start of the
                                      ; block of RGB three-byte values
                                      ; to load into the DAC registers
      mov   ax,1012h                  ;AH=10h selects set color function,
                                      ; AL=12h selects set block of DAC
                                      ; registers subfunction
      sub   bx,bx                     ;load the block of registers
                                      ; starting at DAC register #0
      mov   cx,100h                   ;set all 256 registers
      int   10h                       ;load the DAC registers
```

```
                                        ;now fill the screen with
                                        ; concentric diamonds in all 256
                                        ; color attributes
        mov     ax,0a000h               ;point DS to the display memory
        mov     ds,ax                   ; segment
                                        ;
                                        ;draw diagonal lines in the upper-
                                        ; left quarter of the screen
        mov     al,2                    ;start with color attribute #2
        mov     ah,-1                   ;cycle down through the colors
        mov     bx,320                  ;draw top to bottom (distance from
                                        ; one line to the next)
        mov     dx,160                  ;width of rectangle
        mov     si,100                  ;height of rectangle
        sub     di,di                   ;start at (0,0)
        mov     bp,1                    ;draw left to right (distance from
                                        ; one column to the next)
        call    FillBlock               ;draw it
                                        ;
                                        ;draw diagonal lines in the upper-
                                        ; right quarter of the screen
        mov     al,2                    ;start with color attribute #2
        mov     ah,-1                   ;cycle down through the colors
        mov     bx,320                  ;draw top to bottom (distance from
                                        ; one line to the next)
        mov     dx,160                  ;width of rectangle
        mov     si,100                  ;height of rectangle
        mov     di,319                  ;start at (319,0)
        mov     bp,-1                   ;draw right to left (distance from
                                        ; one column to the next)
        call    FillBlock               ;draw it

                                        ;draw diagonal lines in the lower-
                                        ; left quarter of the screen
        mov     al,2                    ;start with color attribute #2
        mov     ah,-1                   ;cycle down through the colors
        mov     bx,-320                 ;draw bottom to top (distance from
                                        ; one line to the next)
        mov     dx,160                  ;width of rectangle
        mov     si,100                  ;height of rectangle
        mov     di,199*320              ;start at (0,199)
        mov     bp,1                    ;draw left to right (distance from
                                        ; one column to the next)
        call    FillBlock               ;draw it
                                        ;
                                        ;draw diagonal lines in the lower-
                                        ; right quarter of the screen
        mov     al,2                    ;start with color attribute #2
        mov     ah,-1                   ;cycle down through the colors
        mov     bx,-320                 ;draw bottom to top (distance from
                                        ; one line to the next)
        mov     dx,160                  ;width of rectangle
        mov     si,100                  ;height of rectangle
        mov     di,199*320+319          ;start at (319,199)
        mov     bp,-1                   ;draw right to left (distance from
                                        ; one column to the next)
        call    FillBlock               ;draw it

        mov     ah,1                    ;wait for a key
        int     21h                     ;
        mov   ax,0003h                  ;return to text mode
        int   10h                       ;

        mov   ah,4ch                    ;done--return to DOS
        int   21h

; Fills the specified rectangular area of the screen with diagonal lines.
;
; Input:
;       AL = initial attribute with which to draw
;       AH = amount by which to advance the attribute from
;               one pixel to the next
;       BX = distance to advance from one pixel to the next
;       DX = width of rectangle to fill
;       SI = height of rectangle to fill
;       DS:DN = screen address of first pixel to draw
;       BP = offset from the start of one column to the start of
;               the next

FillBlock:
FillHorzLoop:
        push  di                        ;preserve pointer to top of column
        push  ax                        ;preserve initial attribute
        mov   cx,si                     ;column height
FillVertLoop:
        mov   [di],al                   ;set the pixel
        add   di,bx                     ;point to the next row in the column
        add   al,ah                     ;advance the attribute
        loop  FillVertLoop              ;
        pop   ax                        ;restore initial attribute
        add   al,ah                     ;advance to the next attribute to
                                        ; start the next column
        pop   di                        ;retrieve pointer to top of column
        add   di,bp                     ;point to next column
        dec   dx                        ;have we done all columns?
        jnz   FillHorzLoop              ;no, do the next column
        ret                             ;

          end Start
```

Note the jagged lines at the corners of the screen when you run Listing 33.1. This shows how coarse the 320×200 resolution of mode 13H actually is. Now look at how smoothly the colors blend together in the rest of the screen. This is an excellent example of how careful color selection can boost perceived resolution, as for example when drawing antialiased lines, as discussed in Chapter 42.

Finally, note that the border of the screen turns green when Listing 33.1 is run. Listing 33.1 reprograms DAC register 0 to green, and the border attribute (in the Overscan register) happens to be 0, so the border comes out green even though we haven't touched the Overscan register. Normally, attribute 0 is black, causing the border to vanish, but the border is an 8-bit attribute that has to pass through the DAC just like any other pixel value, and it's just as subject to DAC color translation as the pixels controlled by display memory. However, the border color is not affected by the palette RAM or by the Color Select register.

In this chapter, we traced the surprisingly complex path by which the VGA turns a pixel value into RGB analog signals headed for the monitor. In the next chapter and Chapter A on the companion CD-ROM, we'll look at some more code that plays with VGA color. We'll explore in more detail the process of reading and writing the palette RAM and DAC registers, and we'll observe color paging and cycling in action.