Chapter 43

Bit-Plane Animation

Chapter 43

A Simple and Extremely Fast Animation Method for Limited Color

When it comes to computers, my first love is animation. There's nothing quite like the satisfaction of fooling the eye and creating a miniature reality simply by rearranging a few bytes of display memory. What makes animation particularly interesting is that it has to happen fast (as measured in human time), and without blinking and flickering, or else you risk destroying the illusion of motion and solidity. Those constraints make animation the toughest graphics challenge—and also the most rewarding.

It pains me to hear industry pundits rag on the PC when it comes to animation. Okay, I'll grant you that the PC isn't a Silicon Graphics workstation and never will be, but then neither is anything else on the market. The VGA offers good resolution and color, and while the hardware wasn't *designed* for animation, that doesn't mean we can't put it to work in that capacity. One lesson that any good PC graphics or assembly programmer learns quickly is that it's what the PC's hardware *can* do—not what it was intended to do—that's important. (By the way, if I were to pick one aspect of the PC to dump on, it would be sound, not animation. The PC's sound circuity really is lousy, and it's hard to understand why that should be, given that a cheap sound chip—which even the almost-forgotten PC*jr* had—would have changed everything. I guess IBM figured "serious" computer users would be put off by a computer that could make fun noises.)

Anyway, my point is that the PC's animation capabilities are pretty good. There's a trick, though: You can only push the VGA to its animation limits by stretching your mind a bit and using some unorthodox approaches to animation. In fact, stretching your mind is the key to producing good code for *any* task on the PC—that's the topic of the first part of this book. For most software, however, it's not fatal if your code isn't excellent—there's slow but functional software all over the place. When it comes to VGA animation, though, you won't get to first base without a clever approach.

So, what clever approaches do I have in mind? All sorts. The resources of the VGA (or even its now-ancient predecessor, the EGA) are many and varied, and can be applied and combined in hundreds of ways to produce effective animation. For example, refer back to Chapter 23 for an example of page flipping. Or look at the July 1986 issue of *PC Tech Journal*, which describes the basic block-move animation technique, or the August 1987 issue of *PC Tech Journal*, which shows a software-sprite scheme built around the EGA's vertical interrupt and the AND-OR image drawing technique. Or look over the rest of this book, which contains dozens of tips and tricks that can be applied to animation, including Mode X-based techniques starting in Chapter 47 that are the basis for many commercial games.

This chapter adds yet another sort of animation to the list. We're going to take advantage of the bit-plane architecture and color palette of the VGA to develop an animation architecture that can handle several overlapping images with terrific speed and with virtually perfect visual quality. This technique produces no overlap effects or flicker and allows us to use the fastest possible method to draw images—the **REP MOVS** instruction. It has its limitations, but unlike Mode X and some other animation techniques, the techniques I'll show you in this chapter will also work on the EGA, which may be important in some applications.

As with any technique on the PC, there are tradeoffs involved with bit-plane animation. While bit-plane animation is extremely attractive as far as performance and visual quality are concerned, it is somewhat limited. Bit-plane animation supports only four colors plus the background color at any one time, each image must consist of only one of the four colors, and it's preferable that images of the same color not intersect.

It doesn't much matter if bit-plane animation isn't perfect for all applications, though. The real point of showing you bit-plane animation is to bring home the reality that the VGA is a complex adapter with many resources, and that you can do remarkable things if you understand those resources and come up with creative ways to put them to work at specific tasks.

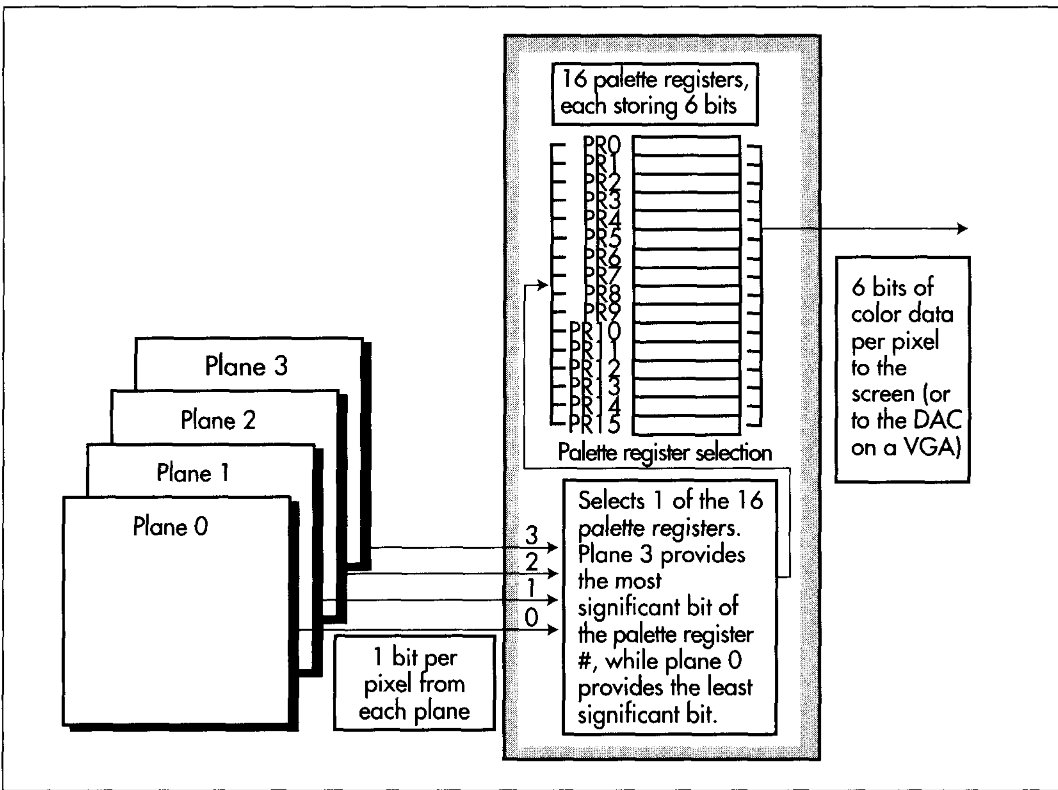
Bit-Planes: The Basics

The underlying principle of bit-plane animation is extremely simple. The VGA has four separate bit planes in modes 0DH, 0EH, 10H, and 12H. Plane 0 normally contains data for the blue component of pixel color, plane 1 normally contains green pixel

data, plane 2 red pixel data, and plane 3 intensity pixel data—but we're going to mix that up a bit in a moment, so we'll simply refer to them as planes 0, 1, 2, and 3 from now on.

Each bit plane can be written to independently. The contents of the four bit planes are used to generate pixels, with the four bits that control the color of each pixel coming from the four planes. However, the bits from the planes go through a look-up stage on the way to becoming pixels—they're used to look up a 6-bit color from one of the sixteen palette registers. Figure 43.1 shows how the bits from the four planes feed into the palette registers to select the color of each pixel. (On the VGA specifically, the output of the palette registers goes to the DAC for an additional look-up stage, as described in Chapters 33 and 34 and also Chapter A on the companion CD-ROM.)

Take a good look at Figure 43.1. Any light bulbs going on over your head yet? If not, consider this. The general problem with VGA animation is that it's complex and

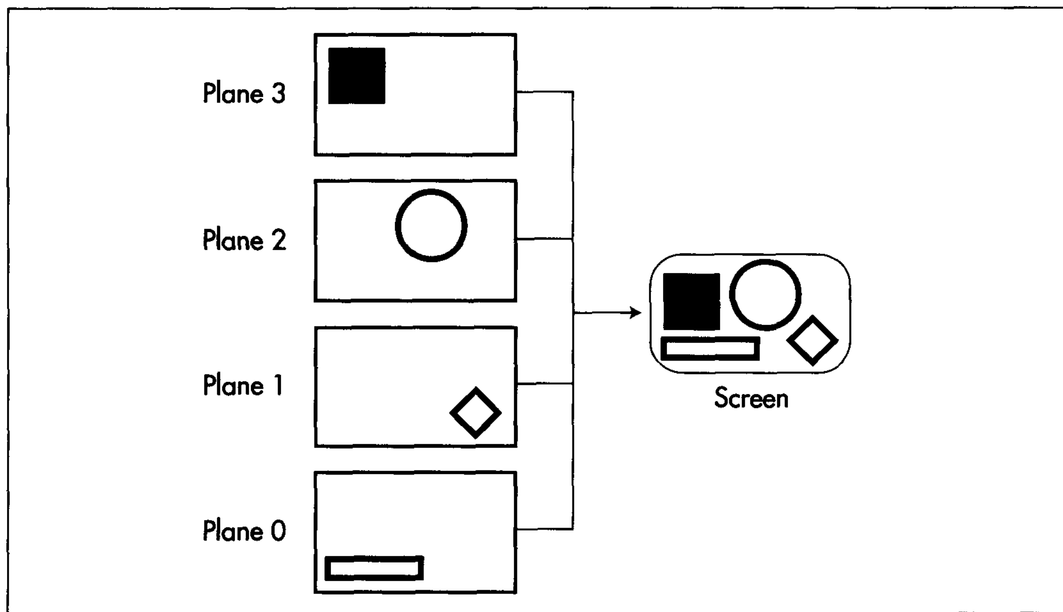


How 4 bits of video data become 6 bits of color.
Figure 43.1

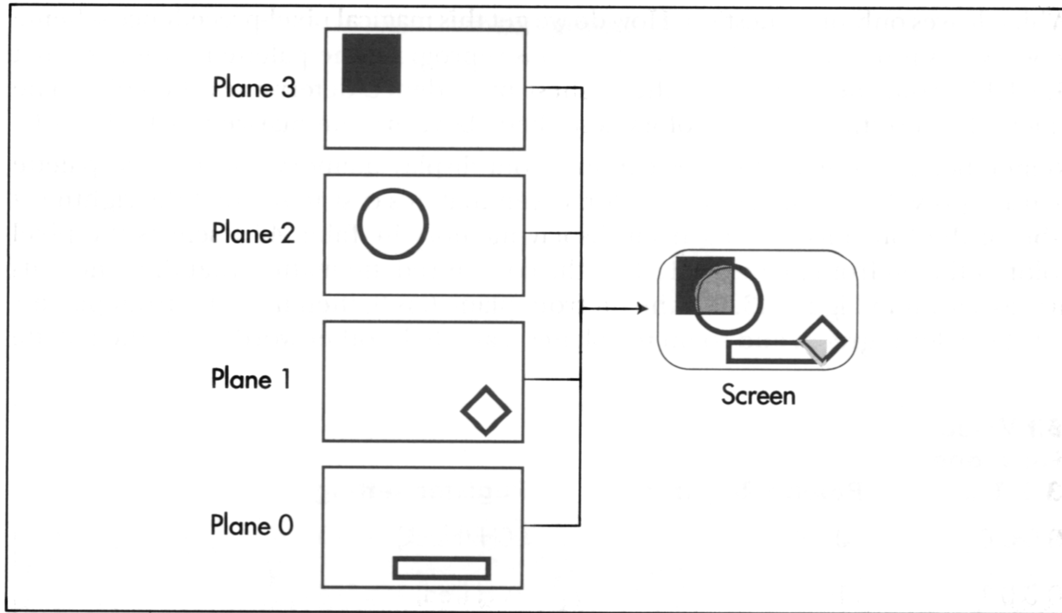
time-consuming to manipulate images that span the four planes (as most do), and that it's hard to avoid interference problems when images intersect, since those images share the same bits in display memory. Since the four bit planes can be written to and read from independently, it should be apparent that if we could come up with a way to display images from each plane independently of whatever images are stored in the other planes, we would have four sets of images that we could manipulate very easily. There would be no interference effects between images in different planes, because images in one plane wouldn't share bits with images in another plane. What's more, since all the bits for a given image would reside in a single plane, we could do away with the cumbersome programming of the VGA's complex hardware that is needed to manipulate images that span multiple planes.

All in all, it would be a good deal if we could store each image in a single plane, as shown in Figure 43.2. However, a problem arises when images in different planes overlap, as shown in Figure 43.3. The combined bits from overlapping images generate new colors, so the overlapping parts of the images don't look like they belong to either of the two images. What we really want, of course, is for one of the images to appear to be in front of the other. It would be better yet if the rearward image showed through any transparent (that is, background-colored) parts of the forward image. Can we do that?

You bet.



Storing images in separate planes.
Figure 43.2



The problem of overlapping colors.
Figure 43.3

Stacking the Palette Registers

Suppose that instead of viewing the four bits per pixel coming out of display memory as selecting one of sixteen colors,we view those bits as selecting one of *four* colors. If the bit from plane 0 is 1, that would select color 0 (say, red). The bit from plane 1 would select color 1 (say, green), the bit from plane 2 would select color 2 (say, blue), and the bit from plane 3 would select color 3 (say, white). Whenever more than 1 bit is 1, the 1 bit from the lowest-numbered plane would determine the color, and 1 bits from all other planes would be ignored. Finally, the absence of any 1 bits at all would select the background color (say, black).

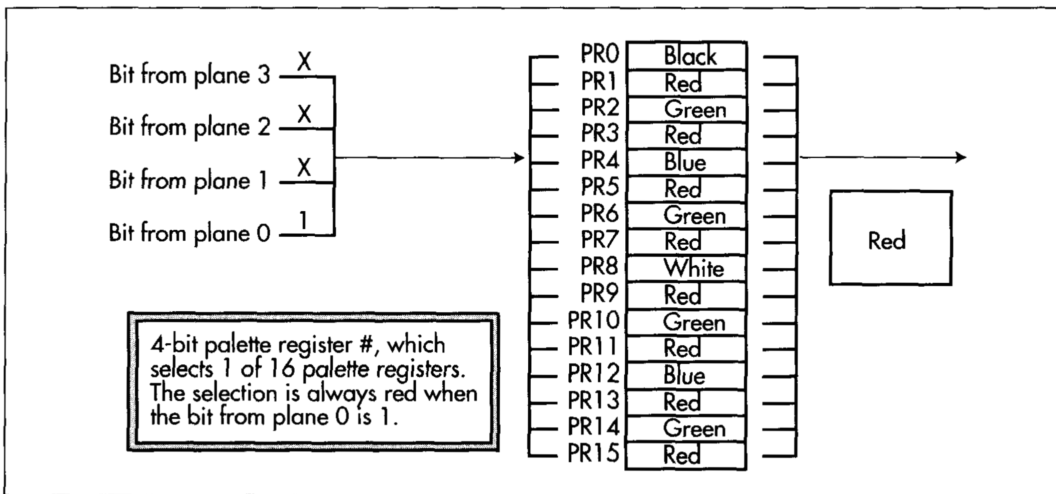
That would give us four colors and the background color. It would also give us nifty image precedence, with images in plane 0 appearing to be in front of images from the other planes, images in plane 1 appearing to be in front of images from planes 2 and 3, and so on. It would even give us transparency, where rearward images would show through holes within and around the edges of images in forward planes. Finally, and most importantly, it would meet all the criteria needed to allow us to store each image in a single plane, letting us manipulate the images very quickly and with no reprogramming of the VGA's hardware other than the few **OUT** instructions required to select the plane we want to write to.

Which leaves only one question: How do we get this magical pixel-precedence scheme to work? As it turns out, all we need to do is reprogram the palette registers so that the 1 bit from the plane with the highest precedence determines the color. The palette RAM settings for the colors described above are summarized in Table 43.1.

Remember that the 4-bit values coming from display memory select which palette register provides the actual pixel color. Given that, it's easy to see that the rightmost 1-bit of the four bits coming from display memory in Table 43.1 selects the pixel color. If the bit from plane 0 is 1, then the color is red, no matter what the other bits are, as shown in Figure 43.4. If the bit from plane 0 is 0, then if the bit from plane 1 is 1 the color is green, and so on for planes 2 and 3. In other words, with the palette

Bit Value For Plane 3 2 1 0	Palette Register	Register setting
0 0 0 0	0	00H (black)
0 0 0 1	1	3CH (red)
0 0 1 0	2	3AH (green)
0 0 1 1	3	3CH (red)
0 1 0 0	4	39H (blue)
0 1 0 1	5	3CH (red)
0 1 1 0	6	3AH (green)
0 1 1 1	7	3CH (red)
1 0 0 0	8	3FH (white)
1 0 0 1	9	3CH (red)
1 0 1 0	10	3AH (green)
1 0 1 1	11	3CH (red)
1 1 0 0	12	39H (blue)
1 1 0 1	13	3CH (red)
1 1 1 0	14	3AH (green)
1 1 1 1	15	3CH (red)

Table 43.1 Palette RAM settings for bit-plane animation.



How pixel precedence works.
Figure 43.4

register settings we instantly have exactly what we want, which is an approach that keeps images in one plane from interfering with images in other planes while providing precedence and transparency.

Seems almost too easy, doesn't it? Nonetheless, it works beautifully, as we'll see very shortly. First, though, I'd like to point out that there's nothing sacred about plane 0 having precedence. We could rearrange the palette register settings so that any plane had the highest precedence, followed by the other planes in any order. I've chosen to make plane 0 the highest precedence only because it seems simplest to think of plane 0 as appearing in front of plane 1, which is in front of plane 2, which is in front of plane 3.

Bit-Plane Animation in Action

Without further ado, Listing 43.1 shows bit-plane animation in action. Listing 43.1 animates 13 rather large images (each 32 pixels on a side) over a complex background at a good clip *even on a primordial 8088-based PC.* Five of the images move very quickly, while the other 8 bounce back and forth at a steady pace.

LISTING 43.1 L43-1.ASM

```
; Program to demonstrate bit-plane animation. Performs
; flicker-free animation with image transparency and
; image precedence across four distinct planes, with
; 13 32x32 images kept in motion at once.
;
;
```

```
; Set to higher values to slow down on faster computers.
; 0 is fine for a PC. 500 is a reasonable setting for an AT.
; Slowing animation further allows a good look at
; transparency and the lack of flicker and color effects
; when images cross.
;
SLOWDOWN    equ   10000
;
; Plane selects for the four colors we're using.
;
RED   equ   01h
GREEN equ   02h
BLUE  equ   04h
WHITE equ   08h
;
VGA_SEGMENT      equ   0a000h      ;mode 10h display memory
                                   ; segment
SC_INDEX         equ   3c4h        ;Sequence Controller Index
                                   ; register
MAP_MASK         equ   2           ;Map Mask register index in
                                   ; Sequence Controller
SCREEN_WIDTH     equ   80          ;# of bytes across screen
SCREEN_HEIGHT    equ   350         ;# of scan lines on screen
WORD_OUTS_OK     equ   1           ;set to 0 to assemble for
                                   ; computers that can't
                                   ; handle word outs to
                                   ; indexed VGA regs
;
stack segment para stack 'STACK'
      db      512 dup (?)
stack ends
;
; Complete info about one object that we're animating.
;
ObjectStructure  struc
Delay            dw    ?     ;used to delay for n passes
                             ; throught the loop to
                             ; control animation speed
BaseDelay        dw    ?     ;reset value for Delay
Image            dw    ?     ;pointer to drawing info
                             ; for object
XCoord           dw    ?     ;object X location in pixels
XInc             dw    ?     ;# of pixels to increment
                             ; location by in the X
                             ; direction on each move
XLeftLimit       dw    ?     ;left limit of X motion
XRightLimit      dw    ?     ;right limit of X motion
YCoord           dw    ?     ;object Y location in pixels
YInc             dw    ?     ;# of pixels to increment
                             ; location by in the Y
                             ; direction on each move
YTopLimit        dw    ?     ;top limit of Y motion
YBottomLimit     dw    ?     ;bottom limit of Y motion
PlaneSelect      db    ?     ;mask to select plane to
                             ; which object is drawn
                 db    ?     ;to make an even # of words
                             ; long, for better 286
                             ; performance (keeps the
                             ; following structure
                             ; word-aligned)
ObjectStructure  ends
```

```
;
Data  segment    word 'DATA'
;
; Palette settings to give plane 0 precedence, followed by
; planes 1, 2, and 3. Plane 3 has the lowest precedence (is
; obscured by any other plane), while plane 0 has the
; highest precedence (displays in front of any other plane).
;
Colors      db   000h       ;background color-black
            db   03ch       ;plane 0 only-red
            db   03ah       ;plane 1 only-green
            db   03ch       ;planes 0&1-red (plane 0 priority)
            db   039h       ;plane 2 only-blue
            db   03ch       ;planes 0&2-red (plane 0 priority)
            db   03ah       ;planes 1&2-green (plane 1 priority)
            db   03ch       ;planes 0&1&2-red (plane 0 priority)
            db   03fh       ;plane 3 only-white
            db   03ch       ;planes 0&3-red (plane 0 priority)
            db   03ah       ;planes 1&3-green (plane 1 priority)
            db   03ch       ;planes 0&1&3-red (plane 0 priority)
            db   039h       ;planes 2&3-blue (plane 2 priority)
            db   03ch       ;planes 0&2&3-red (plane 0 priority)
            db   03ah       ;planes 1&2&3-green (plane 1 priority)
            db   03ch       ;planes 0&1&2&3-red (plane 0 priority)
            db   000h       ;border color-black
;
; Image of a hollow square.
; There's an 8-pixel-wide blank border around all edges
; so that the image erases the old version of itself as
; it's moved and redrawn.
;
Square      label byte
      dw    48,6 ;height in pixels, width in bytes
      rept  8
      db    0,0,0,0,0,0 ;top blank border
      endm
      .radix     2
      db    0,11111111,11111111,11111111,11111111,0
      db    0,11111111,11111111,11111111,11111111,0
      db    0,11111111,11111111,11111111,11111111,0
      db    0,11111111,11111111,11111111,11111111,0
      db    0,11111111,11111111,11111111,11111111,0
      db    0,11111111,11111111,11111111,11111111,0
      db    0,11111111,11111111,11111111,11111111,0
      db    0,11111111,11111111,11111111,11111111,0
      db    0,11111111,00000000,00000000,11111111,0
      db    0,11111111,00000000,00000000,11111111,0
      db    0,11111111,00000000,00000000,11111111,0
      db    0,11111111,00000000,00000000,11111111,0
      db    0,11111111,00000000,00000000,11111111,0
      db    0,11111111,00000000,00000000,11111111,0
      db    0,11111111,00000000,00000000,11111111,0
      db    0,11111111,00000000,00000000,11111111,0
      db    0,11111111,00000000,00000000,11111111,0
      db    0,11111111,00000000,00000000,11111111,0
      db    0,11111111,00000000,00000000,11111111,0
      db    0,11111111,00000000,00000000,11111111,0
      db    0,11111111,00000000,00000000,11111111,0
      db    0,11111111,00000000,00000000,11111111,0
      db    0,11111111,00000000,00000000,11111111,0
      db    0,11111111,00000000,00000000,11111111,0
```

```
     db    0,11111111,11111111,11111111,11111111,0
     db    0,11111111,11111111,11111111,11111111,0
     db    0,11111111,11111111,11111111,11111111,0
     db    0,11111111,11111111,11111111,11111111,0
     db    0,11111111,11111111,11111111,11111111,0
     db    0,11111111,11111111,11111111,11111111,0
     db    0,11111111,11111111,11111111,11111111,0
     db    0,11111111,11111111,11111111,11111111,0
     .radix     10
     rept  8
     db    0,0,0,0,0,0 ;bottom blank border
     endm
;
; Image of a hollow diamond with a smaller diamond in the
; middle.
; There's an 8-pixel-wide blank border around all edges
; so that the image erases the old version of itself as
; it's moved and redrawn.
;
Diamond     label byte
     dw    48,6  ;height in pixels, width in bytes
     rept  8
     db    0,0,0,0,0,0 ;top blank border
     endm
     .radix     2
     db    0,00000000,00000001,10000000,00000000,0
     db    0,00000000,00000011,11000000,00000000,0
     db    0,00000000,00000111,11100000,00000000,0
     db    0,00000000,00001111,11110000,00000000,0
     db    0,00000000,00011111,11111000,00000000,0
     db    0,00000000,00111110,01111100,00000000,0
     db    0,00000000,01111100,00111110,00000000,0
     db    0,00000000,11111000,00011111,00000000,0
     db    0,00000001,11110000,00001111,10000000,0
     db    0,00000011,11100000,00000111,11000000,0
     db    0,00000111,11000000,00000011,11100000,0
     db    0,00001111,10000001,10000001,11110000,0
     db    0,00011111,00000011,11000000,11111000,0
     db    0,00111110,00000111,11100000,01111100,0
     db    0,01111100,00001111,11110000,00111110,0
     db    0,11111000,00011111,11111000,00011111,0
     db    0,11111000,00011111,11111000,00011111,0
     db    0,01111100,00001111,11110000,00111110,0
     db    0,00111110,00000111,11100000,01111100,0
     db    0,00011111,00000011,11000000,11111000,0
     db    0,00001111,10000001,10000001,11110000,0
     db    0,00000111,11000000,00000011,11100000,0
     db    0,00000011,11100000,00000111,11000000,0
     db    0,00000001,11110000,00001111,10000000,0
     db    0,00000000,11111000,00011111,00000000,0
     db    0,00000000,01111100,00111110,00000000,0
     db    0,00000000,00111110,01111100,00000000,0
     db    0,00000000,00011111,11111000,00000000,0
     db    0,00000000,00001111,11110000,00000000,0
     db    0,00000000,00000111,11100000,00000000,0
     db    0,00000000,00000011,11000000,00000000,0
     db    0,00000000,00000001,10000000,00000000,0
     .radix     10
     rept  8
     db    0,0,0,0,0,0 ;bottom blank border
     endm
```

```
;
; List of objects to animate.
;
     even  ;word-align for better 286 performance
;
ObjectList label ObjectStructure
 ObjectStructure <1,21,Diamond,88,8,80,512,16,0,0,350,RED>
 ObjectStructure <1,15,Square,296,8,112,480,144,0,0,350,RED>
 ObjectStructure <1,23,Diamond,88,8,80,512,256,0,0,350,RED>
 ObjectStructure <1,13,Square,120,0,0,640,144,4,0,280,BLUE>
 ObjectStructure <1,11,Diamond,208,0,0,640,144,4,0,280,BLUE>
 ObjectStructure <1,8,Square,296,0,0,640,144,4,0,288,BLUE>
 ObjectStructure <1,9,Diamond,384,0,0,640,144,4,0,288,BLUE>
 ObjectStructure <1,14,Square,472,0,0,640,144,4,0,280,BLUE>
 ObjectStructure <1,8,Diamond,200,8,0,576,48,6,0,280,GREEN>
 ObjectStructure <1,8,Square,248,8,0,576,96,6,0,280,GREEN>
 ObjectStructure <1,8,Diamond,296,8,0,576,144,6,0,280,GREEN>
 ObjectStructure <1,8,Square,344,8,0,576,192,6,0,280,GREEN>
 ObjectStructure <1,8,Diamond,392,8,0,576,240,6,0,280,GREEN>
ObjectListEnd    label ObjectStructure
;
Data  ends
;
; Macro to output a word value to a port.
;
OUT_WORD   macro
if WORD_OUTS_OK
     out   dx,ax
else
     out   dx,al
     inc   dx
     xchg  ah,al
     out   dx,al
     dec   dx
     xchg  ah,al
endif
     endm
;
; Macro to output a constant value to an indexed VGA
; register.
;
CONSTANT_TO_INDEXED_REGISTER macro ADDRESS, INDEX, VALUE
     mov   dx,ADDRESS
     mov   ax,(VALUE shl 8) + INDEX
     OUT_WORD
     endm
;
Code  segment
     assume     cs:Code, ds:Data
Start proc  near
     cld
     mov   ax,Data
     mov   ds,ax
;
; Set 640x350 16-color mode.
;
     mov   ax,0010h     ;AH=0 means select mode
                        ;AL=10h means select
                        ; mode 10h
     int   10h          ;BIOS video interrupt
```

```
;
; Set the palette up to provide bit-plane precedence. If
; planes 0 & 1 overlap, the plane 0 color will be shown;
; if planes 1 & 2 overlap, the plane 1 color will be
; shown; and so on.
;
     mov   ax,(10h shl 8) + 2     ;AH = 10h means
                                  ; set palette
                                  ; registers fn
                                  ;AL = 2 means set
                                  ; all palette
                                  ; registers
     push  ds                     ;ES:DX points to
     pop   es                     ; the palette
     mov   dx,offset Colors       ; settings
     int   10h                    ;call the BIOS to
                                  ; set the palette
;
; Draw the static backdrop in plane 3. All the moving images
; will appear to be in front of this backdrop, since plane 3
; has the lowest precedence the way the palette is set up.
;
     CONSTANT_TO_INDEXED_REGISTER SC_INDEX, MAP_MASK, 08h
                                  ;allow data to go to
                                  ; plane 3 only
;
; Point ES to display memory for the rest of the program.
;
     mov   ax,VGA_SEGMENT
     mov   es,ax
;
     sub   di,di
     mov   bp,SCREEN_HEIGHT/16    ;fill in the screen
                                  ; 16 lines at a time
BackdropBlockLoop:
     call  DrawGridCross          ;draw a cross piece
     call  DrawGridVert           ;draw the rest of a
                                  ; 15-high block
     dec   bp
     jnz   BackdropBlockLoop
     call  DrawGridCross          ;bottom line of grid
;
; Start animating!
;
AnimationLoop:
     mov   bx,offset ObjectList   ;point to the first
                                  ; object in the list
;
; For each object, see if it's time to move and draw that
; object.
;
ObjectLoop:
;
; See if it's time to move this object.
;
     dec   [bx+Delay]             ;count down delay
     jnz   DoNextObject           ;still delaying-don't move
     mov   ax,[bx+BaseDelay]
     mov   [bx+Delay],ax          ;reset delay for next time
```

```
;
; Select the plane that this object will be drawn in.
;
      mov   dx,SC_INDEX
      mov   ah,[bx+PlaneSelect]
      mov   al,MAP_MASK
      OUT_WORD
;
; Advance the X coordinate, reversing direction if either
; of the X margins has been reached.
;
      mov   cx,[bx+XCoord]          ;current X location
      cmp   cx,[bx+XLeftLimit]      ;at left limit?
      ja    CheckXRightLimit        ;no
      neg   [bx+XInc]               ;yes-reverse
CheckXRightLimit:
      cmp   cx,[bx+XRightLimit]     ;at right limit?
      jb    SetNewX                 ;no
      neg   [bx+XInc]               ;yes-reverse
SetNewX:
      add   cx,[bx+XInc]            ;move the X coord
      mov   [bx+XCoord],cx          ; & save it
;
; Advance the Y coordinate, reversing direction if either
; of the Y margins has been reached.
;
      mov   dx,[bx+YCoord]          ;current Y location
      cmp   dx,[bx+YTopLimit];at top limit?
      ja    CheckYBottomLimit;no
      neg   [bx+YInc]               ;yes-reverse
CheckYBottomLimit:
      cmp   dx,[bx+YBottomLimit]    ;at bottom limit?
      jb    SetNewY                 ;no
      neg   [bx+YInc]               ;yes-reverse
SetNewY:
      add   dx,[bx+YInc]            ;move the Y coord
      mov   [bx+YCoord],dx          ; & save it
;
; Draw at the new location. Because of the plane select
; above, only one plane will be affected.
;
      mov   si,[bx+Image]           ;point to the
                                    ; object's image
                                    ; info
      call  DrawObject
;
; Point to the next object in the list until we run out of
; objects.
;
DoNextObject:
      add   bx,size ObjectStructure
      cmp   bx,offset ObjectListEnd
      jb    ObjectLoop
;
; Delay as specified to slow things down.
;
if SLOWDOWN
      mov   cx,SLOWDOWN
DelayLoop:
      loop  DelayLoop
endif
```

```
;
; If a key's been pressed, we're done, otherwise animate
; again.
;
CheckKey:
      mov   ah,1
      int   16h                 ;is a key waiting?
      jz    AnimationLoop       ;no
      sub   ah,ah
      int   16h                 ;yes-clear the key & done
;
; Back to text mode.
;
      mov   ax,0003h            ;AL=03h means select
                                ; mode 03h
      int   10h
;
; Back to DOS.
;
      mov   ah,4ch              ;DOS terminate function
      int   21h                 ;done
;
Start endp
;
; Draws a single grid cross-element at the display memory
; location pointed to by ES:DI. 1 horizontal line is drawn
; across the screen.
;
; Input: ES:DI points to the address at which to draw
;
; Output: ES:DI points to the address following the
;          line drawn
;
; Registers altered: AX, CX, DI
;
DrawGridCross    proc  near
      mov   ax,0ffffh           ;draw a solid line
      mov   cx,SCREEN_WIDTH/2-1
      rep   stosw               ;draw all but the rightmost
                                ; edge
      mov   ax,0080h
      stosw                     ;draw the right edge of the
                                ; grid
      ret
DrawGridCross    endp
;
; Draws the non-cross part of the grid at the display memory
; location pointed to by ES:DI. 15 scan lines are filled.
;
; Input: ES:DI points to the address at which to draw
;
; Output: ES:DI points to the address following the
;          part of the grid drawn
;
; Registers altered: AX, CX, DX, DI
;
DrawGridVert     proc  near
      mov   ax,0080h            ;pattern for a vertical line
      mov   dx,15               ;draw 15 scan lines (all of
                                ; a grid block except the
                                ; solid cross line)
```

```
BackdropRowLoop:
      mov   cx,SCREEN_WIDTH/2
      rep   stosw                    ;draw this scan line's bit
                                     ; of all the vertical lines
                                     ; on the screen
      dec   dx
      jnz   BackdropRowLoop
      ret
DrawGridVert     endp
;
; Draw the specified image at the specified location.
; Images are drawn on byte boundaries horizontally, pixel
; boundaries vertically.
; The Map Mask register must already have been set to enable
; access to the desired plane.
;
; Input:
;     CX - X coordinate of upper left corner
;     DX - Y coordinate of upper left corner
;     DS:SI - pointer to draw info for image
;     ES - display memory segment
;
; Output: none
;
; Registers altered: AX, CX, DX, SI, DI, BP
;
DrawObject  proc  near
      mov   ax,SCREEN_WIDTH
      mul   dx                       ;calculate the start offset in
                                     ; display memory of the row the
                                     ; image will be drawn at
      shr   cx,1
      shr   cx,1
      shr   cx,1                     ;divide the X coordinate in pixels
                                     ; by 8 to get the X coordinate in
                                     ; bytes
      add   ax,cx                    ;destination offset in display
                                     ; memory for the image
      mov   di,ax                    ;point ES:DI to the address to
                                     ; which the image will be copied
                                     ; in display memory
      lodsw
      mov   dx,ax                    ;# of lines in the image
      lodsw                          ;# of bytes across the image
      mov   bp,SCREEN_WIDTH
      sub   bp,ax                    ;# of bytes to add to the display
                                     ; memory offset after copying a line
                                     ; of the image to display memory in
                                     ; order to point to the address
                                     ; where the next line of the image
                                     ; will go in display memory
DrawLoop:
      mov   cx,ax                    ;width of the image
      rep   movsb                    ;copy the next line of the image
                                     ; into display memory
      add   di,bp                    ;point to the address at which the
                                     ; next line will go in display
                                     ; memory
      dec   dx                       ;count down the lines of the image
      jnz   DrawLoop
      ret
```

```
DrawObject endp
;
Code ends
     end  Start
```

For those of you who haven't experienced the frustrations of animation programming on a PC, there's a *whole* lot of animation going on in Listing 43.1. What's more, the animation is virtually flicker-free, partly thanks to bit-plane animation and partly because images are never really erased but rather are simply overwritten. (The principle behind the animation is that of redrawing each image with a blank fringe around it when it moves, so that the blank fringe erases the part of the old image that the new image doesn't overwrite. For details on this sort of animation, see the above-mentioned *PC Tech Journal* July 1986 article.) Better yet, the red images take precedence over the green images, which take precedence over the blue images, which take precedence over the white backdrop, and all obscured images show through holes in and around the edges of images in front of them.

In short, Listing 43.1 accomplishes everything we wished for earlier in an animation technique.

If you possibly can, run Listing 43.1. The animation may be a revelation to those of you who are used to weak, slow animation on PCs with EGA or VGA adapters. Bit-plane animation makes the PC look an awful lot like—dare I say it?—a games machine.

Listing 43.1 was designed to run at the absolute fastest speed, and as I mentioned it puts in a pretty amazing performance on the slowest PCs of all. Assuming you'll be running Listing 43.1 on an faster computer, you'll have to crank up the **DELAY** equate at the start of Listing 43.1 to slow things down to a reasonable pace. (It's not a very good game where all the pieces are a continual blur!) Even on something as modest as a 286-based AT, Listing 43.1 runs much too fast without a substantial delay (although it does look rather interesting at warp speed). We should all have such problems, eh? In fact, we could easily increase the number of animated images past 20 on that old AT, and well into the hundreds on a cutting-edge local-bus 486 or Pentium.

I'm not going to discuss Listing 43.1 in detail; the code is very thoroughly commented and should speak for itself, and most of the individual components of Listing 43.1—the Map Mask register, mode sets, word versus byte **OUT** instructions to the VGA—have been covered in earlier chapters. Do notice, however, that Listing 43.1 sets the palette exactly as I described earlier. This is accomplished by passing a pointer to a 17-byte array (1 byte for each of the 16 palette registers, and 1 byte for the border color) to the BIOS video interrupt (**INT** 10H), function 10H, subfunction 2.

Bit-plane animation does have inherent limitations, which we'll get to in a second. One limitation that is *not* inherent to bit-plane animation but simply a shortcoming of Listing 43.1 is somewhat choppy horizontal motion. In the interests of both clarity and keeping Listing 43.1 to a reasonable length, I decided to byte-align all images horizontally. This saved the many tables needed to define the 7 non-byte-aligned

rotations of the images, as well as the code needed to support rotation. Unfortunately, it also meant that the smallest possible horizontal movement was 8 pixels (1 byte of display memory), which is far enough to be noticeable at certain speeds. The situation is, however, easily correctable with the additional rotations and code. We'll see an implementation of fully rotated images (in this case for Mode X, but the principles generalize nicely) in Chapter 49. Vertically, where there is no byte-alignment issue, the images move 4 or 6 pixels at a times, resulting in considerably smoother animation.

The addition of code to support rotated images would also open the door to support for internal animation, where the appearance of a given image changes over time to suggest that the image is an active entity. For example, propellers could whirl, jaws could snap, and jets could flare. Bit-plane animation with bit-aligned images and internal animation can look truly spectacular. It's a sight worth seeing, particularly for those who doubt the PC's worth when it comes to animation.

Limitations of Bit-Plane Animation

As I've said, bit-plane animation is not perfect. For starters, bit-plane animation can only be used in the VGA's planar modes, modes 0DH, 0EH, 10H, and 12H. Also, the reprogramming of the palette registers that provides image precedence also reduces the available color set from the normal 16 colors to just 5 (one color per plane plus the background color). Worse still, each image must consist entirely of only one of the four colors. Mixing colors within an image is not allowed, since the bits for each image are limited to a single plane and can therefore select only one color. Finally, all images of the same precedence must be the same color.

It is possible to work around the color limitations to some extent by using only one or two planes for bit-plane animation, while reserving the other planes for multicolor drawing. For example, you could use plane 3 for bit-plane animation while using planes 0-2 for normal 8-color drawing. The images in plane 3 would then appear to be in front of the 8-color images. If we wanted the plane 3 images to be yellow, we could set up the palette registers as shown in Table 43.2.

As you can see, the color yellow is displayed whenever a pixel's bit from plane 3 is 1. This gives the images from plane 3 precedence, while leaving us with the 8 normal low-intensity colors for images drawn across the other 3 planes, as shown in Figure 43.5. Of course, this approach provides only 1 rather than 3 high-precedence planes, but that might be a good tradeoff for being able to draw multi-colored images as a backdrop to the high-precedence images. For the right application, high-speed flicker-free plane 3 images moving in front of an 8-color backdrop could be a potent combination indeed.

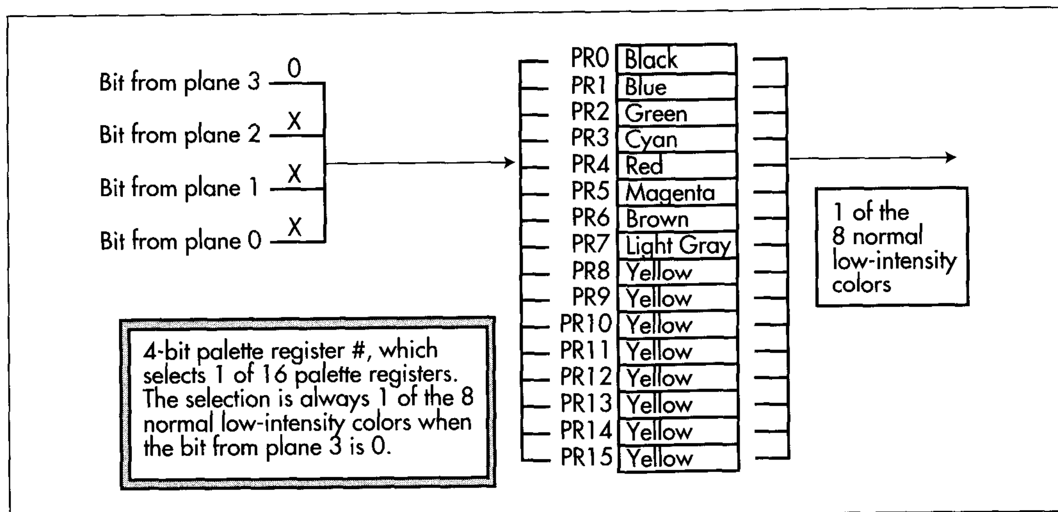
Another limitation of bit-plane animation is that it's best if images stored in the same plane never cross each other. Why? Because when images do cross, the blank fringe

Palette Register	Register Setting
0	00H (black)
1	01H (blue)
2	02H (green)
3	03H (cyan)
4	04H (red)
5	05H (magenta)
6	14H (brown)
7	07H (light gray)
8	3EH (yellow)
9	3EH (yellow)
10	3EH (yellow)
11	3EH (yellow)
12	3EH (yellow)
13	3EH (yellow)
14	3EH (yellow)
15	3EH (yellow)

Table 43.2 Palette RAM settings for two-plane animation.

around each image can temporarily erase the overlapped parts of the other image or images, resulting in momentary flicker. While that's not fatal, it certainly detracts from the rock-solid animation effect of bit-plane animation.

Not allowing images in the same plane to overlap is actually less of a limitation than it seems. Run Listing 43.1 again. Unless you were looking for it, you'd never notice that images of the same color almost never overlap—there's plenty of action to distract the eye, and the trajectories of images of the same color are arranged so that they have a full range of motion without running into each other. The only exception is the chain of green images, which occasionally doubles back on itself when it bounces directly into a corner and reverses direction. Here, however, the images are moving so quickly that the brief moment during which one image's fringe blanks a



Pixel precedence for plane 3 only.
Figure 43.5

portion of another image is noticeable only upon close inspection, and not particularly unaesthetic even then.

When a technique has such tremendous visual and performance advantages as does bit-plane animation, it behooves you to design your animation software so that the limitations of the animation technique don't get in the way. For example, you might design a shooting gallery game with all the images in a given plane marching along in step in a continuous band. The images could never overlap, so bit-plane animation would produce very high image quality.

Shearing and Page Flipping

As Listing 43.1 runs, you may occasionally see an image shear, with the top and bottom parts of the image briefly offset. This is a consequence of drawing an image directly into memory as that memory is being scanned for video data. Occasionally the CRT controller scans a given area of display memory for pixel data just as the program is changing that same memory. If the CRT controller scans memory faster than the CPU can modify that memory, then the CRT controller can scan out the bytes of display memory that have been already been changed, pass the point in the image that the CPU is currently drawing, and start scanning out bytes that haven't yet been changed. The result: Mismatched upper and lower portions of the image.

If the CRT controller scans more slowly than the CPU can modify memory (likely with a 386, a fast VGA, and narrow images), then the CPU can rip right past the CRT

controller, with the same net result of mismatched top and bottom parts of the image, as the CRT controller scans out first unchanged bytes and then changed bytes. Basically, shear will occasionally occur unless the CPU and CRT proceed at exactly the same rate, which is most unlikely. Shear is more noticeable when there are fewer but larger images, since it's more apparent when a larger screen area is sheared, and because it's easier to spot one out of three large images momentarily shearing than one out of twenty small images.

Image shear isn't terrible—I've written and sold several games in which images occasionally shear, and I've never heard anyone complain—but neither is it ideal. One solution is page flipping, in which drawing is done to a non-displayed page of display memory while another page of display memory is shown on the screen. (We saw page flipping back in Chapter 23, we'll see it again in the next chapter, and we'll use it heavily starting in Chapter 47.) When the drawing is finished, the newly-drawn part of display memory is made the displayed page, so that the new screen becomes visible all at once, with no shearing or flicker. The other page is then drawn to, and when the drawing is complete the display is switched back to that page.

Page flipping can be used in conjunction with bit-plane animation, although page flipping does diminish some of the unique advantages of bit-plane animation. Page flipping produces animation of the highest visual quality whether bit-plane animation is used or not. There are a few drawbacks to page flipping, however.

Page flipping requires two display memory buffers, one to draw in and one to display at any given time. Unfortunately, in mode 12H there just isn't enough memory for two buffers, so page flipping is not an option in that mode.

Also, page flipping requires that you keep the contents of both buffers up to date, which can require a good deal of extra drawing.

Finally, page flipping requires that you wait until you're sure the page has flipped before you start drawing to the other page. Otherwise, you could end up modifying a page while it's still being displayed, defeating the whole purpose of page flipping. Waiting for pages to flip takes time and can slow overall performance significantly. What's more, it's sometimes difficult to be sure when the page has flipped, since not all VGA clones implement the display adapter status bits and page flip timing identically.

To sum up, bit-plane animation by itself is very fast and looks good. In conjunction with page flipping, bit-plane animation looks a little better but is slower, and the overall animation scheme is more difficult to implement and perhaps a bit less reliable on some computers.

Beating the Odds in the Jaw-Dropping Contest

Bit-plane animation is neat stuff. Heck, good animation of *any* sort is fun, and the PC is as good a place as any (well, almost any) to make people's jaws drop. (Certainly it's

the place to go if you want to make a *lot* of jaws drop.) Don't let anyone tell you that you can't do good animation on the PC. You can—*if* you stretch your mind to find ways to bring the full power of the VGA to bear on your applications. Bit-plane animation isn't for every task; neither are page flipping, exclusive-ORing, pixel panning, or any of the many other animation techniques you have available. One or more tricks from that grab-bag should give you what you need, though, and the bigger your grab-bag, the better your programs.