Chapter 23

Bones and Sinew

Chapter 23

At the Very Heart of Standard PC Graphics

The VGA is unparalleled in the history of computer graphics, for it is by far the most widely-used graphics standard ever, the closest we may ever come to a *lingua franca* of computer graphics. No other graphics standard has even come close to the 50,000,000 or so VGAs in use today, and virtually every PC compatible sold today has full VGA compatibility built in. There are, of course, a variety of graphics accelerators that outperform the standard VGA, and indeed, it is becoming hard to find a plain vanilla VGA anymore—but there is no standard for accelerators, and every accelerator contains a true-blue VGA at its core.

What that means is that if you write your programs for the VGA, you'll have the largest possible market for your software. In order for graphics-based software to succeed, however, it must perform well. Wringing the best performance from the VGA is no simple task, and it's *impossible* unless you really understand how the VGA works—unless you have the internals down cold. This book is about PC graphics at many levels, but high performance is the foundation for all that is to come, so it is with the inner workings of the VGA that we will begin our exploration of PC graphics.

The first eight chapters of Part II is a guided tour of the heart of the VGA; after you've absorbed what we'll cover in this and the next seven chapters, you'll have the foundation for understanding just about everything the VGA can do, including the fabled Mode X and more. As you read through these first chapters, please keep in mind that the *really* exciting stuff—animation, 3-D, blurry-fast lines and circles and

polygons—has to wait until we have the fundamentals out of the way. So hold on and follow along, and before you know it the fireworks will be well underway.

We'll start our exploration with a quick overview of the VGA, and then we'll dive right in and get a taste of what the VGA can do.

The VGA

The VGA is the baseline adapter for modern IBM PC compatibles, present in virtually every PC sold today or in the last several years. (Note that the VGA is often nothing more than a chip on a motherboard, with some memory, a DAC, and maybe a couple of glue chips; nonetheless, I'll refer to it as an adapter from now on for simplicity.) It guarantees that every PC is capable of documented resolutions up to 640×480 (with 16 possible colors per pixel) and 320×200 (with 256 colors per pixel), as well as undocumented—but nonetheless thoroughly standard—resolutions up to 360×480 in 256-color mode, as we'll see in Chapters 31–34 and 47–49. In order for a video adapter to claim VGA compatibility, it must support all the features and code discussed in this book (with a very few minor exceptions that I'll note)—and my experience is that just about 100 percent of the video hardware currently shipping or shipped since 1990 is in fact VGA compatible. Therefore, VGA code will run on nearly all of the 50,000,000 or so PC compatibles out there, with the exceptions being almost entirely obsolete machines from the 1980s. This makes good VGA code and VGA programming expertise valuable commodities indeed.

Right off the bat, I'd like to make one thing perfectly clear: The VGA is hard—sometimes *very* hard—to program for good performance. Hard, but not impossible—and that's why I like this odd board. It's a throwback to an earlier generation of micros, when inventive coding and a solid understanding of the hardware were the best tools for improving performance. Increasingly, faster processors and powerful coprocessors are seen as the solution to the sluggish software produced by high-level languages and layers of interface and driver code, and that's surely a valid approach. However, there are tens of millions of VGAs installed right now, in machines ranging from 6-MHz 286s to 90-MHz Pentiums. What's more, because the VGAs are generally 8- or at best 16-bit devices, and because of display memory wait states, a faster processor isn't as much of a help as you'd expect. The upshot is that only a seasoned performance programmer who understands the VGA through and through can drive the board to its fullest potential.

Throughout this book, I'll explore the VGA by selecting a specific algorithm or feature and implementing code to support it on the VGA, examining aspects of the VGA architecture as they become relevant. You'll get to see VGA features in context, where they are more comprehensible than in IBM's somewhat arcane documentation, and you'll get working code to use or to modify to meet your needs.

The prime directive of VGA programming is that there's rarely just one way to program the VGA for a given purpose. Once you understand the tools the VGA provides,

you'll be able to combine them to generate the particular synergy your application needs. My VGA routines are not intended to be taken as gospel, or to show "best" implementations, but rather to start you down the road to understanding the VGA. Let's begin.

An Introduction to VGA Programming

Most discussions of the VGA start out with a traditional "Here's a block diagram of the VGA" approach, with lists of registers and statistics. I'll get to that eventually, but you can find it in IBM's VGA documentation and several other books. Besides, it's numbing to read specifications and explanations, and the VGA is an exciting adapter, the kind that makes you want to get your hands dirty probing under the hood, to write some nifty code just to see what the board can do. What's more, the best way to understand the VGA is to see it work, so let's jump right into a sample of the VGA in action, getting a feel for the VGA's architecture in the process.

Listing 23.1 is a sample VGA program that pans around an animated 16-color medium-resolution (640×350) playfield. There's a lot packed into this code; I'm going to focus on the VGA-specific aspects so we don't get sidetracked. I'm not going to explain how the ball is animated, for example; we'll get to animation starting in Chapter 42. What I will do is cover each of the VGA features used in this program—the virtual screen, vertical and horizontal panning, color plane manipulation, multi-plane block copying, and page flipping—at a conceptual level, letting the code itself demonstrate the implementation details. We'll return to many of these concepts in more depth later in this book.

At the Core

A little background is necessary before we're ready to examine Listing 23.1. The VGA is built around four functional blocks, named the CRT Controller (CRTC), the Sequence Controller (SC), the Attribute Controller (AC), and the Graphics Controller (GC). The single-chip VGA could have been designed to treat the registers for all the blocks as one large set, addressed at one pair of I/O ports, but in the EGA, each of these blocks was a separate chip, and the legacy of EGA compatibility is why each of these blocks has a separate set of registers and is addressed at different I/O ports in the VGA.

Each of these blocks has a sizable complement of registers. It is not particularly important that you understand why a given block has a given register; all the registers together make up the programming interface, and it is the entire interface that is of interest to the VGA programmer. However, the means by which most VGA registers are addressed makes it necessary for you to remember which registers are in which blocks.

Most VGA registers are addressed as *internally indexed* registers. The internal address of the register is written to a given block's Index register, and then the data for that register is written to the block's Data register. For example, GC register 8, the Bit

Mask register, is set to 0FFH by writing 8 to port 3CEH, the GC Index register, and then writing 0FFH to port 3CFH, the GC Data register. Internal indexing makes it possible to address the 9 GC registers through only two ports, and allows the entire VGA programming interface to be squeezed into fewer than a dozen ports. The downside is that two I/O operations are required to access most VGA registers.

The ports used to control the VGA are shown in Table 23.1. The CRTC, SC, and GC Data registers are located at the addresses of their respective Index registers plus one. However, the AC Index and Data registers are located at the same address, 3C0H. The function of this port toggles on every **OUT** to 3C0H, and resets to Index mode (in which the Index register is programmed by the next **OUT** to 3C0H) on every read from the Input Status 1 register (3DAH when the VGA is in a color mode,

Register	Address
AC Index/Data register	3C0H (write with toggle)
AC Index register	3C0H (read)
AC Data register	3C1H (read)
Miscellaneous Output register	3C2H (write)
	3CCH (read)
Input Status 0 register	3C2H (read)
SC Index register	3C4H (read/write)
SC Data register	3C5H (read/write)
GC Index register	3CEH (read/write)
GC Data register	3CFH (read/write)
CRTC Index register	3B4H/3D4H (read/write)
CRTC Data register	3B5H/3D5H (read/write)
Input Status 1 register/	
AC Index/Data reset	3 BAH/3DAH (read)
Feature Control	3BAH/3DAH (write)
	3CAH (read)

Table 1.1 The Ports through which the VGA is controlled.

3BAH in monochrome modes). Note that all CRTC registers are addressed at either 3DXH or 3BXH, the former in color modes and the latter in monochrome modes. This provides compatibility with the register addressing of the now-vanished Color/Graphics Adapter and Monochrome Display Adapter.

The method used in the VGA BIOS to set registers is to point DX to the desired Index register, load AL with the index, perform a byte **OUT**, increment DX to point to the Data register (except in the case of the AC, where DX remains the same), load AL with the desired data, and perform a byte **OUT**. A handy shortcut is to point DX to the desired Index register, load AL with the index, load AH with the data, and perform a word **OUT**. Since the high byte of the **OUT** value goes to port DX+1, this is equivalent to the first method but is faster. However, this technique does not work for programming the AC Index and Data registers; both AC registers are addressed at 3C0H, so two separate byte **OUT**s must be used to program the AC. (Actually, word **OUT**s to the AC do work in the EGA, but not in the VGA, so they shouldn't be used.) As mentioned above, you must be sure which mode—Index or Data—the AC is in before you do an **OUT** to 3C0H; you can read the Input Status 1 register at any time to force the AC to Index mode.

How safe is the word-**OUT** method of addressing VGA registers? I have, in the past, run into adapter/computer combinations that had trouble with word **OUT**s; however, all such problems I am aware of have been fixed. Moreover, a great deal of graphics software now uses word **OUT**s, so any computer or VGA that doesn't properly support word **OUT**s could scarcely be considered a clone at all.

A speed tip: The setting of each chip's Index register remains the same until it is reprogrammed. This means that in cases where you are setting the same internal register repeatedly, you can set the Index register to point to that internal register once, then write to the Data register multiple times. For example, the Bit Mask register (GC register 8) is often set repeatedly inside a loop when drawing lines. The standard code for this is:

```
MOV     DX,03CEH    ;point to GC Index register
MOV     AL,8        ;internal index of Bit Mask register
OUT     DX,AX       ;AH contains Bit Mask register setting
```

Alternatively, the GC Index register could initially be set to point to the Bit Mask register with

```
MOV         DX,03CEH  ;point to GC Index register
MOV         AL,8      ;internal index of Bit Mask register
OUT         DX,AL     ;set GC Index register
INC         DX        ;point to GC Data register
```

*and then the Bit Mask register could be set repeatedly with the byte-size **OUT** instruction*

```
OUT         DX,AL     ;AL contains Bit Mask register setting
```

which is generally faster (and never slower) than a word-sized ***OUT****, and which does not require AH to be set, freeing up a register. Of course, this method only works if the GC Index register remains unchanged throughout the loop.*

Linear Planes and True VGA Modes

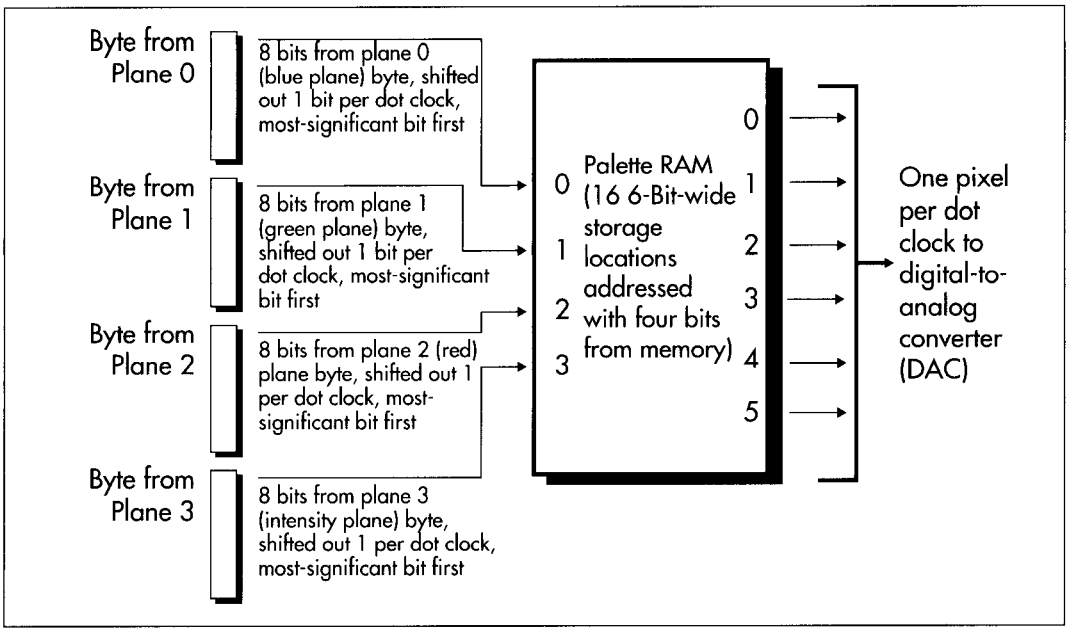
The VGA's memory is organized as four 64K planes. Each of these planes is a linear bitmap; that is, each byte from a given plane controls eight adjacent pixels on the screen, the next byte controls the next eight pixels, and so on to the end of the scan line. The next byte then controls the first eight pixels of the next scan line, and so on to the end of the screen.

The VGA adds a powerful twist to linear addressing; the logical width of the screen in VGA memory need not be the same as the physical width of the display. The programmer is free to define all or part of the VGA's large memory map as a logical screen of up to 4,080 pixels in width, and then use the physical screen as a window onto any part of the logical screen. What's more, a virtual screen can have any logical height up to the capacity of VGA memory. Such a virtual screen could be used to store a spreadsheet or a CAD/CAM drawing, for instance. As we will see shortly, the VGA provides excellent hardware for moving around the virtual screen; taken together, the virtual screen and the VGA's smooth panning capabilities can generate very impressive effects.

All four linear planes are addressed in the same 64K memory space starting at A000:0000. Consequently, there are four bytes at any given address in VGA memory. The VGA provides special hardware to assist the CPU in manipulating all four planes, in parallel, with a single memory access, so that the programmer doesn't have to spend a great deal of time switching between planes. Astute use of this VGA hardware allows VGA software to as much as quadruple performance by processing the data for all the planes in parallel.

Each memory plane provides one bit of data for each pixel. The bits for a given pixel from each of the four planes are combined into a nibble that serves as an address into the VGA's palette RAM, which maps the one of 16 colors selected by display memory into any one of 64 colors, as shown in Figure 23.1. All sixty-four mappings for all 16 colors are independently programmable. (We'll discuss the VGA's color capabilities in detail starting in Chapter 33.)

The VGA BIOS supports several graphics modes (modes 4, 5, and 6) in which VGA memory appears not to be organized as four linear planes. These modes exist for CGA compatibility only, and are not true VGA graphics modes; use them when you need CGA-type operation and ignore them the rest of the time. The VGA's special features are most powerful in true VGA modes, and it is on the 16-color true-VGA modes (modes 0DH (320×200), 0EH (640×200), 10H (640×350), and 12H (640×480)) that I will concentrate in this part of the book. There is also a 256-color mode, mode 13H, that appears to be a single linear plane, but, as we will see in Chapters 31–34



Video data from memory to pixel.
Figure 23.1

and 47–49 of this book, that's a polite fiction—and discarding that fiction gives us an opportunity to unleash the power of the VGA's hardware for vastly better performance. VGA text modes, which feature soft fonts, are another matter entirely, upon which we'll touch from time to time.

With that background out of the way, we can get on to the sample VGA program shown in Listing 23.1. I suggest you run the program before continuing, since the explanations will mean far more to you if you've seen the features in action.

LISTING 23.1 L23-1.ASM

```
; Sample VGA program.
; Animates four balls bouncing around a playfield by using
; page flipping. Playfield is panned smoothly both horizontally
; and vertically.
; By Michael Abrash.
;
stack   segment para stack 'STACK'
        db      512 dup(?)
stack   ends
;
MEDRES_VIDEO_MODE       equ     0       ;define for 640x350 video mode
                                        ; comment out for 640x200 mode
VIDEO_SEGMENT   equ     0a000h          ;display memory segment for
                                        ; true VGA graphics modes
LOGICAL_SCREEN_WIDTH    equ     672/8   ;width in bytes and height in scan
```

```
LOGICAL_SCREEN_HEIGHT   equ     384     ; lines of the virtual screen
                                        ; we'll work with
PAGE0           equ     0       ;flag for page 0 when page flipping
PAGE1           equ     1       ;flag for page 1 when page flipping
PAGE0_OFFSET    equ     0       ;start offset of page 0 in VGA memory
PAGE1_OFFSET    equ     LOGICAL_SCREEN_WIDTH * LOGICAL_SCREEN_HEIGHT
                        ;start offset of page 1 (both pages
                        ; are 672x384 virtual screens)
BALL_WIDTH      equ     24/8    ;width of ball in display memory bytes
BALL_HEIGHT     equ     24      ;height of ball in scan lines
BLANK_OFFSET    equ     PAGE1_OFFSET * 2        ;start of blank image
                                                ; in VGA memory
BALL_OFFSET     equ     BLANK_OFFSET + (BALL_WIDTH * BALL_HEIGHT)
                        ;start offset of ball image in VGA memory
NUM_BALLS       equ     4       ;number of balls to animate
;
; VGA register equates.
;
SC_INDEX        equ     3c4h    ;SC index register
MAP_MASK        equ     2       ;SC map mask register
GC_INDEX        equ     3ceh    ;GC index register
GC_MODE         equ     5       ;GC mode register
CRTC_INDEX      equ     03d4h   ;CRTC index register
START_ADDRESS_HIGH equ  0ch     ;CRTC start address high byte
START_ADDRESS_LOW equ   0dh     ;CRTC start address low byte
CRTC_OFFSET     equ     13h     ;CRTC offset register
INPUT_STATUS_1  equ     03dah   ;VGA status register
VSYNC_MASK      equ     08h     ;vertical sync bit in status register 1
DE_MASK         equ     01h     ;display enable bit in status register 1
AC_INDEX        equ     03c0h   ;AC index register
HPELPAN         equ     20h OR 13h   ;AC horizontal pel panning register
                                     ; (bit 7 is high to keep palette RAM
                                     ; addressing on)
dseg    segment para common 'DATA'
CurrentPage             db      PAGE1           ;page to draw to
CurrentPageOffset       dw      PAGE1_OFFSET
;
; Four plane's worth of multicolored ball image.
;
BallPlane0Image label   byte            ;blue plane image
        db      000h, 03ch, 000h, 001h, 0ffh, 080h
        db      007h, 0ffh, 0e0h, 00fh, 0ffh, 0f0h
        db      4 * 3 dup(000h)
        db      07fh, 0ffh, 0feh, 0ffh, 0ffh, 0ffh
        db      0ffh, 0ffh, 0ffh, 0ffh, 0ffh, 0ffh
        db      4 * 3 dup(000h)
        db      07fh, 0ffh, 0feh, 03fh, 0ffh, 0fch
        db      03fh, 0ffh, 0fch, 01fh, 0ffh, 0f8h
        db      4 * 3 dup(000h)
BallPlane1Image label   byte            ;green plane image
        db      4 * 3 dup(000h)
        db      01fh, 0ffh, 0f8h, 03fh, 0ffh, 0fch
        db      03fh, 0ffh, 0fch, 07fh, 0ffh, 0feh
        db      07fh, 0ffh, 0feh, 0ffh, 0ffh, 0ffh
        db      0ffh, 0ffh, 0ffh, 0ffh, 0ffh, 0ffh
        db      8 * 3 dup(000h)
        db      00fh, 0ffh, 0f0h, 007h, 0ffh, 0e0h
        db      001h, 0ffh, 080h, 000h, 03ch, 000h
BallPlane2Image label   byte            ;red plane image
        db      12 * 3 dup(000h)
```

```
        db      0ffh, 0ffh, 0ffh, 0ffh, 0ffh, 0ffh
        db      0ffh, 0ffh, 0ffh, 07fh, 0ffh, 0feh
        db      07fh, 0ffh, 0feh, 03fh, 0ffh, 0fch
        db      03fh, 0ffh, 0fch, 01fh, 0ffh, 0f8h
        db      00fh, 0ffh, 0f0h, 007h, 0ffh, 0e0h
        db      001h, 0ffh, 080h, 000h, 03ch, 000h
BallPlane3Image label   byte            ;intensity on for all planes,
                                        ; to produce high-intensity colors
        db      000h, 03ch, 000h, 001h, 0ffh, 080h
        db      007h, 0ffh, 0e0h, 00fh, 0ffh, 0f0h
        db      01fh, 0ffh, 0f8h, 03fh, 0ffh, 0fch
        db      03fh, 0ffh, 0fch, 07fh, 0ffh, 0feh
        db      07fh, 0ffh, 0feh, 0ffh, 0ffh, 0ffh
        db      0ffh, 0ffh, 0ffh, 0ffh, 0ffh, 0ffh
        db      0ffh, 0ffh, 0ffh, 0ffh, 0ffh, 0ffh
        db      0ffh, 0ffh, 0ffh, 07fh, 0ffh, 0feh
        db      07fh, 0ffh, 0feh, 03fh, 0ffh, 0fch
        db      03fh, 0ffh, 0fch, 01fh, 0ffh, 0f8h
        db      00fh, 0ffh, 0f0h, 007h, 0ffh, 0e0h
        db      001h, 0ffh, 080h, 000h, 03ch, 000h
;
BallX           dw      15, 50, 40, 70          ;array of ball x coords
BallY           dw      40, 200, 110, 300       ;array of ball y coords
LastBallX       dw      15, 50, 40, 70          ;previous ball x coords
LastBallY       dw      40, 100, 160, 30        ;previous ball y coords
BallXInc        dw      1, 1, 1, 1              ;x move factors for ball
BallYInc        dw      8, 8, 8, 8              ;y move factors for ball
BallRep         dw      1, 1, 1, 1              ;# times to keep moving
                                                ; ball according to current
                                                ; increments
BallControl     dw      Ball0Control, Ball1Control     ;pointers to current
                dw      Ball2Control, Ball3Control     ; locations in ball
                                                       ; control strings
BallControlString     dw    Ball0Control, Ball1Control ;pointers to
                      dw    Ball2Control, Ball3Control ; start of ball
                                                       ; control strings
;
; Ball control strings.
;
Ball0Control    label   word
        dw      10, 1, 4, 10, -1, 4, 10, -1, -4, 10, 1, -4, 0
Ball1Control    label   word
        dw      12, -1, 1, 28, -1, -1, 12, 1, -1, 28, 1, 1, 0
Ball2Control    label   word
        dw      20, 0, -1, 40, 0, 1, 20, 0, -1, 0
Ball3Control    label   word
        dw      8, 1, 0, 52, -1, 0, 44, 1, 0, 0
;
; Panning control string.
;
ifdef MEDRES_VIDEO_MODE
PanningControlString    dw      32, 1, 0, 34, 0, 1, 32, -1, 0, 34, 0, -1, 0
else
PanningControlString    dw      32, 1, 0, 184, 0, 1, 32, -1, 0, 184, 0, -1, 0
endif
PanningControl  dw      PanningControlString   ;pointer to current location
                                               ; in panning control string
PanningRep      dw      1       ;# times to pan according to current
                                ; panning increments
PanningXInc     dw      1       ;x panning factor
PanningYInc     dw      0       ;y panning factor
```

```
HPan            db      0       ;horizontal pel panning setting
PanningStartOffset dw   0       ;start offset adjustment to produce vertical
                                ; panning & coarse horizontal panning
dseg    ends
;
; Macro to set indexed register P2 of chip with index register
; at P1 to AL.
;
SETREG  macro   P1, P2
        mov     dx,P1
        mov     ah,al
        mov     al,P2
        out     dx,ax
        endm
;
cseg    segment para public 'CODE'
        assume  cs:cseg, ds:dseg
start   proc    near
        mov     ax,dseg
        mov     ds,ax
;
; Select graphics mode.
;
ifdef MEDRES_VIDEO_MODE
        mov     ax,010h
else
        mov     ax,0eh
endif
        int     10h
;
; ES always points to VGA memory.
;
        mov     ax,VIDEO_SEGMENT
        mov     es,ax
;
; Draw border around playfield in both pages.
;
        mov     di,PAGE0_OFFSET
        call    DrawBorder      ;page 0 border
        mov     di,PAGE1_OFFSET
        call    DrawBorder      ;page 1 border
;
; Draw all four plane's worth of the ball to undisplayed VGA memory.
;
        mov     al,01h          ;enable plane 0
        SETREG  SC_INDEX, MAP_MASK
        mov     si,offset BallPlane0Image
        mov     di,BALL_OFFSET
        mov     cx,BALL_WIDTH * BALL_HEIGHT
        rep movsb
        mov     al,02h          ;enable plane 1
        SETREG  SC_INDEX, MAP_MASK
        mov     si,offset BallPlane1Image
        mov     di,BALL_OFFSET
        mov     cx,BALL_WIDTH * BALL_HEIGHT
        rep movsb
        mov     al,04h          ;enable plane 2
        SETREG  SC_INDEX, MAP_MASK
        mov     si,offset BallPlane2Image
        mov     di,BALL_OFFSET
```

```
        mov     cx,BALL_WIDTH * BALL_HEIGHT
        rep movsb
        mov     al,08h          ;enable plane 3
        SETREG  SC_INDEX, MAP_MASK
        mov     si,offset BallPlane3Image
        mov     di,BALL_OFFSET
        mov     cx,BALL_WIDTH * BALL_HEIGHT
        rep movsb
;
; Draw a blank image the size of the ball to undisplayed VGA memory.
;
        mov     al,0fh                  ;enable all memory planes, since the
        SETREG  SC_INDEX, MAP_MASK      ; blank has to erase all planes
        mov     di,BLANK_OFFSET
        mov     cx,BALL_WIDTH * BALL_HEIGHT
        sub     al,al
        rep stosb
;
; Set VGA to write mode 1, for block copying ball and blank images.
;
        mov     dx,GC_INDEX
        mov     al,GC_MODE
        out     dx,al           ;point GC Index to GC Mode register
        inc     dx              ;point to GC Data register
        jmp     $+2             ;delay to let bus settle
        in      al,dx           ;get current state of GC Mode
        and     al,not 3        ;clear the write mode bits
        or      al,1            ;set the write mode field to 1
        jmp     $+2             ;delay to let bus settle
        out     dx,al
;
; Set VGA offset register in words to define logical screen width.
;
        mov     al,LOGICAL_SCREEN_WIDTH / 2
        SETREG  CRTC_INDEX, CRTC_OFFSET
;
; Move the balls by erasing each ball, moving it, and
; redrawing it, then switching pages when they're all moved.
;
BallAnimationLoop:
        mov     bx,( NUM_BALLS * 2 ) - 2
EachBallLoop:
;
; Erase old image of ball in this page (at location from one more earlier).
;
        mov     si,BLANK_OFFSET ;point to blank image
        mov     cx,[LastBallX+bx]
        mov     dx,[LastBallY+bx]
        call    DrawBall
;
; Set new last ball location.
;
        mov     ax,[BallX+bx]
        mov     [LastballX+bx],ax
        mov     ax,[BallY+bx]
        mov     [LastballY+bx],ax
;
; Change the ball movement values if it's time to do so.
;
        dec     [BallRep+bx]            ;has current repeat factor run out?
        jnz     MoveBall
        mov     si,[BallControl+bx]     ;it's time to change movement values
```

```
        lodsw                           ;get new repeat factor from
                                        ; control string
        and     ax,ax                   ;at end of control string?
        jnz     SetNewMove
        mov     si,[BallControlString+bx]       ;reset control string
        lodsw                           ;get new repeat factor
SetNewMove:
        mov     [BallRep+bx],ax         ;set new movement repeat factor
        lodsw                           ;set new x movement increment
        mov     [BallXInc+bx],ax
        lodsw                           ;set new y movement increment
        mov     [BallYInc+bx],ax
        mov     [BallControl+bx],si     ;save new control string pointer
;
; Move the ball.
;
MoveBall:
        mov     ax,[BallXInc+bx]
        add     [BallX+bx],ax           ;move in x direction
        mov     ax,[BallYInc+bx]
        add     [BallY+bx],ax           ;move in y direction
;
; Draw ball at new location.
;
        mov     si,BALL_OFFSET  ;point to ball's image
        mov     cx,[BallX+bx]
        mov     dx,[BallY+bx]
        call    DrawBall
;
        dec     bx
        dec     bx
        jns     EachBallLoop

;
; Set up the next panning state (but don't program it into the
; VGA yet).
;
        call    AdjustPanning

;
; Wait for display enable (pixel data being displayed) so we know
; we're nowhere near vertical sync, where the start address gets
; latched and used.
;
        call    WaitDisplayEnable
;
; Flip to the new page by changing the start address.
;
        mov     ax,[CurrentPageOffset]
        add     ax,[PanningStartOffset]
        push    ax
        SETREG  CRTC_INDEX, START_ADDRESS_LOW
        mov     al,byte ptr [CurrentPageOffset+1]
        pop     ax
        mov     al,ah
        SETREG  CRTC_INDEX, START_ADDRESS_HIGH
;
; Wait for vertical sync so the new start address has a chance
; to take effect.
;
```

```
        call    WaitVSync
;
; Set horizontal panning now, just as new start address takes effect.
;
        mov     al,[HPan]
        mov     dx,INPUT_STATUS_1
        in      al,dx                   ;reset AC addressing to index reg
        mov     dx,AC_INDEX
        mov     al,HPELPAN
        out     dx,al                   ;set AC index to pel pan reg
        mov     al,[HPan]
        out     dx,al                   ;set new pel panning
;
; Flip the page to draw to to the undisplayed page.
;
        xor     [CurrentPage],1
        jnz     IsPage1
        mov     [CurrentPageOffset],PAGE0_OFFSET
        jmp     short EndFlipPage
IsPage1:
        mov     [CurrentPageOffset],PAGE1_OFFSET
EndFlipPage:
;
; Exit if a key's been hit.
;
        mov     ah,1
        int     16h
        jnz     Done
        jmp     BallAnimationLoop
;
; Finished, clear key, reset screen mode and exit.
;
Done:
        mov     ah,0    ;clear key
        int     16h
;
        mov     ax,3    ;reset to text mode
        int     10h
;
        mov     ah,4ch  ;exit to DOS
        int     21h
;
start   endp
;
; Routine to draw a ball-sized image to all planes, copying from
; offset SI in VGA memory to offset CX,DX (x,y) in VGA memory in
; the current page.
;
DrawBall        proc    near
        mov     ax,LOGICAL_SCREEN_WIDTH
        mul     dx      ;offset of start of top image scan line
        add     ax,cx   ;offset of upper left of image
        add     ax,[CurrentPageOffset]  ;offset of start of page
        mov     di,ax
        mov     bp,BALL_HEIGHT
        push    ds
        push    es
        pop     ds      ;move from VGA memory to VGA memory
DrawBallLoop:
```

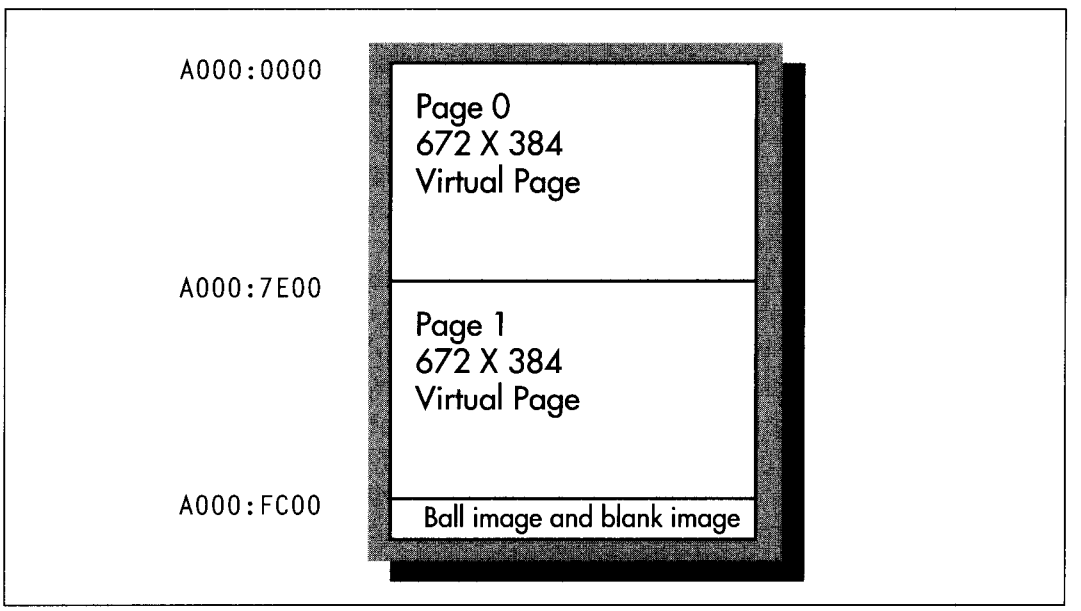
```
        push    di
        mov     cx,BALL_WIDTH
        rep movsb       ;draw a scan line of image
        pop     di
        add     di,LOGICAL_SCREEN_WIDTH ;point to next destination scan line
        dec     bp
        jnz     DrawBallLoop
        pop     ds
        ret
DrawBall        endp
;
; Wait for the leading edge of vertical sync pulse.
;
WaitVSync       proc    near
        mov     dx,INPUT_STATUS_1
WaitNotVSyncLoop:
        in      al,dx
        and     al,VSYNC_MASK
        jnz     WaitNotVSyncLoop
WaitVSyncLoop:
        in      al,dx
        and     al,VSYNC_MASK
        jz      WaitVSyncLoop
        ret
WaitVSync       endp

;
; Wait for display enable to happen (pixels to be scanned to
; the screen, indicating we're in the middle of displaying a frame).
;
WaitDisplayEnable       proc    near
        mov     dx,INPUT_STATUS_1
WaitDELoop:
        in      al,dx
        and     al,DE_MASK
        jnz     WaitDELoop
        ret
WaitDisplayEnable       endp

;
; Perform horizontal/vertical panning.
;
AdjustPanning   proc    near
        dec     [PanningRep]    ;time to get new panning values?
        jnz     DoPan
        mov     si,[PanningControl]     ;point to current location in
                                        ; panning control string
        lodsw                           ;get panning repeat factor
        and     ax,ax                   ;at end of panning control string?
        jnz     SetnewPanValues
        mov     si,offset PanningControlString  ;reset to start of string
        lodsw                           ;get panning repeat factor
SetNewPanValues:
        mov     [PanningRep],ax         ;set new panning repeat value
        lodsw
        mov     [PanningXInc],ax        ;horizontal panning value
        lodsw
        mov     [PanningYInc],ax        ;vertical panning value
        mov     [PanningControl],si     ;save current location in panning
```

```
                                        ; control string
;
; Pan according to panning values.
;
DoPan:
        mov     ax,[PanningXInc]        ;horizontal panning
        and     ax,ax
        js      PanLeft                 ;negative means pan left
        jz      CheckVerticalPan
        mov     al,[HPan]
        inc     al                      ;pan right; if pel pan reaches
        cmp     al,8                    ; 8, it's time to move to the
        jb      SetHPan                 ; next byte with a pel pan of 0
        sub     al,al                   ; and a start offset that's one
        inc     [PanningStartOffset]    ; higher
        jmp     short SetHPan
PanLeft:
        mov     al,[HPan]
        dec     al                      ;pan left; if pel pan reaches -1,
        jns     SetHPan                 ; it's time to move to the next
        mov     al,7                    ; byte with a pel pan of 7 and a
        dec     [PanningStartOffset]    ; start offset that's one lower
SetHPan:
        mov     [HPan],al               ;save new pel pan value
CheckVerticalPan:
        mov     ax,[PanningYInc]        ;vertical panning
        and     ax,ax
        js      PanUp                   ;negative means pan up
        jz      EndPan
        add     [PanningStartOffset],LOGICAL_SCREEN_WIDTH
                                        ;pan down by advancing the start
                                        ; address by a scan line
        jmp     short EndPan
PanUp:
        sub     [PanningStartOffset],LOGICAL_SCREEN_WIDTH
                                        ;pan up by retarding the start
                                        ; address by a scan line
EndPan:
        ret
;
; Draw textured border around playfield that starts at DI.
;
DrawBorder      proc    near
;
; Draw the left border.
;
        push    di
        mov     cx,LOGICAL_SCREEN_HEIGHT / 16
DrawLeftBorderLoop:
        mov     al,0ch          ;select red color for block
        call    DrawBorderBlock
        add     di,LOGICAL_SCREEN_WIDTH * 8
        mov     al,0eh          ;select yellow color for block
        call    DrawBorderBlock
        add     di,LOGICAL_SCREEN_WIDTH * 8
        loop    DrawLeftBorderLoop
        pop     di
;
; Draw the right border.
;
        push    di
```

```
        add     di,LOGICAL_SCREEN_WIDTH - 1
        mov     cx,LOGICAL_SCREEN_HEIGHT / 16
DrawRightBorderLoop:
        mov     al,0eh          ;select yellow color for block
        call    DrawBorderBlock
        add     di,LOGICAL_SCREEN_WIDTH * 8
        mov     al,0ch          ;select red color for block
        call    DrawBorderBlock
        add     di,LOGICAL_SCREEN_WIDTH * 8
        loop    DrawRightBorderLoop
        pop     di
;
; Draw the top border.
;
        push    di
        mov     cx,(LOGICAL_SCREEN_WIDTH - 2) / 2
DrawTopBorderLoop:
        inc     di
        mov     al,0eh          ;select yellow color for block
        call    DrawBorderBlock
        inc     di
        mov     al,0ch          ;select red color for block
        call    DrawBorderBlock
        loop    DrawTopBorderLoop
        pop     di
;
; Draw the bottom border.
;
        add     di,(LOGICAL_SCREEN_HEIGHT - 8) * LOGICAL_SCREEN_WIDTH
        mov     cx,(LOGICAL_SCREEN_WIDTH - 2) / 2
DrawBottomBorderLoop:
        inc     di
        mov     al,0ch          ;select red color for block
        call    DrawBorderBlock
        inc     di
        mov     al,0eh          ;select yellow color for block
        call    DrawBorderBlock
        loop    DrawBottomBorderLoop
        ret
DrawBorder      endp
;
; Draws an 8x8 border block in color in AL at location DI.
; DI preserved.
;
DrawBorderBlock proc    near
        push    di
        SETREG  SC_INDEX, MAP_MASK
        mov     al,0ffh
        rept 8
        stosb
        add     di,LOGICAL_SCREEN_WIDTH - 1
        endm
        pop     di
        ret
DrawBorderBlock endp
AdjustPanning   endp
cseg    ends
        end     start
```

Smooth Panning

The first thing you'll notice upon running the sample program is the remarkable smoothness with which the display pans from side-to-side and up-and-down. That the display can pan at all is made possible by two VGA features: 256K of display memory and the virtual screen capability. Even the most memory-hungry of the VGA modes, mode 12H (640×480), uses only 37.5K per plane, for a total of 150K out of the total 256K of VGA memory. The medium-resolution mode, mode 10H (640×350), requires only 28K per plane, for a total of 112K. Consequently, there is room in VGA memory to store more than two full screens of video data in mode 10H (which the sample program uses), and there is room in all modes to store a larger virtual screen than is actually displayed. In the sample program, memory is organized as two virtual screens, each with a resolution of 672×384, as shown in Figure 23.2. The area of the virtual screen actually displayed at any given time is selected by setting the display memory address at which to begin fetching video data; this is set by way of the start address registers (Start Address High, CRTC register 0CH, and Start Address Low, CRTC register 0DH). Together these registers make up a 16-bit display memory address at which the CRTC begins fetching data at the beginning of each video frame. Increasing the start address causes higher-memory areas of the virtual screen to be



Video memory organization for Listing 23.1.
Figure 23.2

displayed. For example, the Start Address High register could be set to 80H and the Start Address Low register could be set to 00H in order to cause the display screen to reflect memory starting at offset 8000H in each plane, rather than at the default offset of 0.

The logical height of the virtual screen is defined by the amount of VGA memory available. As the VGA scans display memory for video data, it progresses from the start address toward higher memory one scan line at a time, until the frame is completed. Consequently, if the start address is increased, lines farther toward the bottom of the virtual screen are displayed; in effect, the virtual screen appears to scroll up on the physical screen.

The logical width of the virtual screen is defined by the Offset register (CRTC register 13H), which allows redefinition of the number of words of display memory considered to make up one scan line. Normally, 40 words of display memory constitute a scan line; after the CRTC scans these 40 words for 640 pixels worth of data, it advances 40 words from the start of that scan line to find the start of the next scan line in memory. This means that displayed scan lines are contiguous in memory. However, the Offset register can be set so that scan lines are logically wider (or narrower, for that matter) than their displayed width. The sample program sets the Offset register to 2AH, making the logical width of the virtual screen 42 words, or 42 * 2 * 8 = 672 pixels, as contrasted with the actual width of the mode 10h screen, 40 words or 640 pixels. The logical height of the virtual screen in the sample program is 384; this is accomplished simply by reserving 84 * 384 contiguous bytes of VGA memory for the virtual screen, where 84 is the virtual screen width in bytes and 384 is the virtual screen height in scan lines.

The start address is the key to panning around the virtual screen. The start address registers select the row of the virtual screen that maps to the top of the display; panning down a scan line requires only that the start address be increased by the logical scan line width in bytes, which is equal to the Offset register times two. The start address registers select the column that maps to the left edge of the display as well, allowing horizontal panning, although in this case only relatively coarse byte-sized adjustments—panning by eight pixels at a time—are supported.

Smooth horizontal panning is provided by the Horizontal Pel Panning register, AC register 13H, working in conjunction with the start address. Up to 7 pixels worth of single pixel panning of the displayed image to the left is performed by increasing the Horizontal Pel Panning register from 0 to 7. This exhausts the range of motion possible via the Horizontal Pel Panning register; the next pixel's worth of smooth panning is accomplished by incrementing the start address by one and resetting the Horizontal Pel Panning register to 0. Smooth horizontal panning should be viewed as a series of fine adjustments in the 8-pixel range between coarse byte-sized adjustments.

A horizontal panning oddity: Alone among VGA modes, text mode (in most cases) has 9 dots per character clock. Smooth panning in this mode requires cycling the

Horizontal Pel Panning register through the values 8, 0, 1, 2, 3, 4, 5, 6, and 7. 8 is the "no panning" setting.

There is one annoying quirk about programming the AC. When the AC Index register is set, only the lower five bits are used as the internal index. The next most significant bit, bit 5, controls the source of the video data sent to the monitor by the VGA. When bit 5 is set to 1, the output of the palette RAM, derived from display memory, controls the displayed pixels; this is normal operation. When bit 5 is 0, video data does not come from the palette RAM, and the screen becomes a solid color. The only time bit 5 of the AC Index register should be 0 is during the setting of a palette RAM register, since the CPU is only able to write to palette RAM when bit 5 is 0. (Some VGAs do not enforce this, but you should always set bit 5 to 0 before writing to the palette RAM just to be safe.) Immediately after setting palette RAM, however, 20h (or any other value with bit 5 set to 1) should be written to the AC Index register to restore normal video, and at all other times bit 5 should be set to 1.

By the way, palette RAM can be set via the BIOS video interrupt (interrupt 10H), function 10H. Whenever an VGA function can be performed reasonably well through a BIOS function, as it can in the case of setting palette RAM, it should be, both because there is no point in reinventing the wheel and because the BIOS may well mask incompatibilities between the IBM VGA and VGA clones.

Color Plane Manipulation

The VGA provides a considerable amount of hardware assistance for manipulating the four display memory planes. Two features illustrated by the sample program are the ability to control which planes are written to by a CPU write and the ability to copy four bytes—one from each plane—with a single CPU read and a single CPU write.

The Map Mask register (SC register 2) selects which planes are written to by CPU writes. If bit 0 of the Map Mask register is 1, then each byte written by the CPU will be written to VGA memory plane 0, the plane that provides the video data for the least significant bit of the palette RAM address. If bit 0 of the Map Mask register is 0, then CPU writes will not affect plane 0. Bits 1, 2, and 3 of the Map Mask register similarly control CPU access to planes 1, 2, and 3, respectively. Any of the 16 possible combinations of enabled and disabled planes can be selected. Beware, however, of writing to an area of memory that is not zeroed. Planes that are disabled by the Map Mask register are not altered by CPU writes, so old and new images can mix on the screen, producing unwanted color effects as, say, three planes from the old image mix with one plane from the new image. The sample program solves this by ensuring that the memory written to is zeroed. A better way to set all planes at once is provided by the set/reset capabilities of the VGA, which I'll cover in Chapter 25.

The sample program writes the image of the colored ball to VGA memory by enabling one plane at a time and writing the image of the ball for that plane. Each

image is written to the same VGA addresses; only the destination plane, selected by the Map Mask register, is different. You might think of the ball's image as consisting of four colored overlays, which together make up a multicolored image. The sample program writes a blank image to VGA memory by enabling all planes and writing a block of zero bytes; the zero bytes are written to all four VGA planes simultaneously.

The images are written to a nondisplayed portion of VGA memory in order to take advantage of a useful VGA hardware feature, the ability to copy all four planes at once. As shown by the image-loading code discussed above, four different sets of reads and writes—and several **OUT**s as well—are required to copy a multicolored image into VGA memory as would be needed to draw the same image into a non-planar pixel buffer. This causes unacceptably slow performance, all the more so because the wait states that occur on accesses to VGA memory make it very desirable to minimize display memory accesses, and because **OUT**s tend to be very slow.

The solution is to take advantage of the VGA's write mode 1, which is selected via bits 0 and 1 of the GC Mode register (GC register 5). (Be careful to preserve bits 2-7 when setting bits 0 and 1, as is done in Listing 23.1.) In write mode 1, a single CPU read loads the addressed byte from all four planes into the VGA's four internal latches, and a single CPU write writes the contents of the latches to the four planes. During the write, the byte written by the CPU is irrelevant.

The sample program uses write mode 1 to copy the images that were previously drawn to the high end of VGA memory into a desired area of display memory, all in a single block copy operation. This is an excellent way to keep the number of reads, writes, and OUTs required to manipulate the VGA's display memory low enough to allow real-time drawing.

The Map Mask register can still mask out planes in write mode 1. All four planes are copied in the sample program because the Map Mask register is still 0Fh from when the blank image was created.

The animated images appear to move a bit jerkily because they are byte-aligned and so must move a minimum of 8 pixels horizontally. This is easily solved by storing rotated versions of all images in VGA memory, and then in each instance drawing the correct rotation for the pixel alignment at which the image is to be drawn; we'll see this technique in action in Chapter 49.

Don't worry if you're not catching everything in this chapter on the first pass; the VGA is a complicated beast, and learning about it is an iterative process. We'll be going over these features again, in different contexts, over the course of the rest of this book.

Page Flipping

When animated graphics are drawn directly on the screen, with no intermediate frame-composition stage, the image typically flickers and/or ripples, an unavoidable

result of modifying display memory at the same time that it is being scanned for video data. The display memory of the VGA makes it possible to perform page flipping, which eliminates such problems. The basic premise of page flipping is that one area of display memory is displayed while another is being modified. The modifications never affect an area of memory as it is providing video data, so no undesirable side effects occur. Once the modification is complete, the modified buffer is selected for display, causing the screen to change to the new image in a single frame's time, typically 1/60th or 1/70th of a second. The other buffer is then available for modification.

As described above, the VGA has 64K per plane, enough to hold two pages and more in 640×350 mode 10H, but not enough for two pages in 640×480 mode 12H. For page flipping, two non-overlapping areas of display memory are needed. The sample program uses two 672×384 virtual pages, each 32,256 bytes long, one starting at A000:0000 and the other starting at A000:7E00. Flipping between the pages is as simple as setting the start address registers to point to one display area or the other—but, as it turns out, that's not as simple as it sounds.

The timing of the switch between pages is critical to achieving flicker-free animation. It is essential that the program never be modifying an area of display memory as that memory is providing video data. Achieving this is surprisingly complicated on the VGA, however.

The problem is as follows. The start address is latched by the VGA's internal circuitry exactly once per frame, typically (but not always on all clones) at the start of the vertical sync pulse. The vertical sync status is, in fact, available as bit 3 of the Input Status 0 register, addressable at 3BAH (in monochrome modes) or 3DAH (color). Unfortunately, by the time the vertical sync status is observed by a program, the start address for the next frame has already been latched, having happened the instant the vertical sync pulse began. That means that it's no good to wait for vertical sync to begin, then set the new start address; if we did that, we'd have to wait until the *next* vertical sync pulse to start drawing, because the page wouldn't flip until then.

Clearly, what we want is to set the new start address, then wait for the start of the vertical sync pulse, at which point we can be sure the page has flipped. However, we can't just set the start address and wait, because we might have the extreme misfortune to set one of the start address registers before the start of vertical sync and the other after, resulting in mismatched halves of the start address and a nasty jump of the displayed image for one frame.

One possible solution to this problem is to pick a second page start address that has a 0 value for the lower byte, so only the Start Address High register ever needs to be set, but in the sample program in Listing 23.1 I've gone for generality and always set both bytes. To avoid mismatched start address bytes, the sample program waits for pixel data to be displayed, as indicated by the Display Enable status; this tells us we're somewhere in the displayed portion of the frame, far enough away from vertical sync so we can be sure the new start address will get used at the next vertical sync. Once

the Display Enable status is observed, the program sets the new start address, waits for vertical sync to happen, sets the new pel panning state, and then continues drawing. Don't worry about the details right now; page flipping will come up again, at considerably greater length, in later chapters.

As an interesting side note, be aware that if you run DOS software under a multitasking environment such as Windows NT, timeslicing delays can make mismatched start address bytes or mismatched start address and pel panning settings much more likely, for the graphics code can be interrupted at any time. This is also possible, although much less likely, under non-multitasking environments such as DOS, because strategically placed interrupts can cause the same sorts of problems there. For maximum safety, you should disable interrupts around the key portions of your page-flipping code, although here we run into the problem that if interrupts are disabled from the time we start looking for Display Enable until we set the Pel Panning register, they will be off for far too long, and keyboard, mouse, and network events will potentially be lost. Also, disabling interrupts won't help in true multitasking environments, which never let a program hog the entire CPU. This is one reason that pel panning, although indubitably flashy, isn't widely used and should be reserved for only those cases where it's absolutely necessary.

Waiting for the sync pulse has the side effect of causing program execution to synchronize to the VGA's frame rate of 60 or 70 frames per second, depending on the display mode. This synchronization has the useful consequence of causing the program to execute at the same speed on any CPU that can draw fast enough to complete the drawing in a single frame; the program just idles for the rest of each frame that it finishes before the VGA is finished displaying the previous frame.

An important point illustrated by the sample program is that while the VGA's display memory is far larger and more versatile than is the case with earlier adapters, it is nonetheless a limited resource and must be used judiciously. The sample program uses VGA memory to store two 672×384 virtual pages, leaving only 1024 bytes free to store images. In this case, the only images needed are a colored ball and a blank block with which to erase it, so there is no problem, but many applications require dozens or hundreds of images. The tradeoffs between virtual page size, page flipping, and image storage must always be kept in mind when designing programs for the VGA.

To see the program run in 640×200 16-color mode, comment out the **EQU** line for **MEDRES_VIDEO_MODE**.

The Hazards of VGA Clones

Earlier, I said that any VGA that doesn't support the features and functionality covered in this book can't properly be called VGA compatible. I also noted that there are some exceptions, however, and we've just come to the most prominent one. You see, all VGAs really *are* compatible with the IBM VGA's functionality when it comes to

drawing pixels into display memory; all the write modes and read modes and set/reset capabilities and everything else involved with manipulating display memory really does work in the same way on all VGAs and VGA clones. That compatibility isn't as airtight when it comes to scanning pixels out of display memory and onto the screen in certain infrequently-used ways, however.

The areas of incompatibility of which I'm aware are illustrated by the sample program, and may in fact have caused you to see some glitches when you ran Listing 23.1. The problem, which arises only on certain VGAs, is that some settings of the Row Offset register cause some pixels to be dropped or displaced to the wrong place on the screen; often, this happens only in conjunction with certain start address settings. (In my experience, only VRAM (Video RAM)-based VGAs exhibit this problem, no doubt due to the way that pixel data is fetched from VRAM in large blocks.) Panning and large virtual bitmaps can be made to work reliably, by careful selection of virtual bitmap sizes and start addresses, but it's difficult; that's one of the reasons that most commercial software does not use these features, although a number of games do. The upshot is that if you're going to use oversized virtual bitmaps and pan around them, you should take great care to test your software on a wide variety of VRAM- and DRAM-based VGAs.

Just the Beginning

That pretty well covers the important points of the sample VGA program in Listing 23.1. There are many VGA features we didn't even touch on, but the object was to give you a feel for the variety of features available on the VGA, to convey the flexibility and complexity of the VGA's resources, and in general to give you an initial sense of what VGA programming is like. Starting with the next chapter, we'll begin to explore the VGA systematically, on a more detailed basis.

The Macro Assembler

The code in this book is written in both C and assembly. I think C is a good development environment, but I believe that often the best code (although not necessarily the easiest to write or the most reliable) is written in assembly. This is especially true of graphics code for the x86 family, given segments, the string instructions, and the asymmetric and limited register set, and for real-time programming of a complex board like the VGA, there's really no other choice for the lowest-level code.

Before I'm deluged with protests from C devotees, let me add that the majority of my productive work is done in C; no programmer is immune to the laws of time, and C is simply a more time-efficient environment in which to develop, particularly when working in a programming team. In this book, however, we're after the *sine qua non* of PC graphics—performance—and we can't get there from here without a fair amount of assembly language.

Now that we know what the VGA looks like in broad strokes and have a sense of what VGA programming is like, we can start looking at specific areas in depth. In the next chapter, we'll take a look at the hardware assistance the VGA provides the CPU during display memory access. There are four latches and four ALUs in those chips, along with some useful masks and comparators, and it's that hardware that's the difference between sluggish performance and making the VGA get up and dance.