Chapter 44

Split Screens Save the Page Flipped Day

Chapter 44

640x480 Page Flipped Animation in 64K...Almost

Almost doesn't count, they say—at least in horseshoes and maybe a few other things. This is especially true in digital circles, where if you need 12 MB of hard disk to install something and you only have 10 MB left (a situation that seems to be some sort of eternal law) you're stuck.

And that's only infuriating until you dredge up the gumption to go in there and free up some space. How would you feel if you were up against an "almost-but-not-quite" kind of a wall that couldn't be breached by freeing up something elsewhere? Suppose you were within a few KB of implementing a wonderful VGA animation scheme that provided lots of screen space, square pixels, smooth motion and more than adequate speed—but all the memory you have is all there is? What would you do?

Scream a little. Or throw something that won't break easily. Then you sit down and let your right brain do what it was designed to do. Sure enough, there's a way, and in this chapter I'll explain how a little VGA secret called *page splitting* can save the day for page flipped animation in 640×480 mode. But to do that, I have to lay a little groundwork first. Or maybe a lot of groundwork.

No horseshoes here.

A Plethora of Challenges

In its simplest terms, computer animation consists of rapidly redrawing similar images at slightly differing locations, so that the eye interprets the successive images as a single object in motion over time. The fact that the world is an analog realm and the images displayed on a computer screen consist of discrete pixels updated at a maximum rate of about 70 Hz is irrelevant; your eye can interpret both real-world images and pixel patterns on the screen as objects in motion, and that's that.

One of the key problems of computer animation is that it takes time to redraw a screen, time during which the bitmap controlling the screen is in an intermediate state, with, quite possibly, many objects erased and others half-drawn. Even when only briefly displayed, a partially-updated screen can cause flicker at best, and at worst can destroy the illusion of motion entirely.

Another problem of animation is that the screen must update often enough so that motion appears continuous. A moving object that moves just once every second, shifting by hundreds of pixels each time it does move, will appear to jump, not to move smoothly. Therefore, there are two overriding requirements for smooth animation: 1) the bitmap must be updated quickly (once per frame—60 to 70 Hz—is ideal, although 30 Hz will do fine), and, 2) the process of redrawing the screen must be invisible to the user; only the end result should ever be seen. Both of these requirements are met by the program presented in Listings 44.1 and 44.2.

A Page Flipping Animation Demonstration

The listings taken together form a sample animation program, in which a single object bounces endlessly off other objects, with instructions and a count of bounces displayed at the bottom of the screen. I'll discuss various aspects of Listings 44.1 and 44.2 during the balance of this article. The listings are too complex and involve too much VGA and animation knowledge for for me to discuss it all in exhaustive detail (and I've covered a lot of this stuff earlier in the book); instead, I'll cover the major elements, leaving it to you to explore the finer points—and, I hope, to experiment with and expand on the code I'll provide.

LISTING 44.1 L44-1.C

```
/* Split screen VGA animation program. Performs page flipping in the
top portion of the screen while displaying non-page flipped
information in the split screen at the bottom of the screen.
Compiled with Borland C++ in C compilation mode. */

#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include <math.h>

#define SCREEN_SEG         0xA000
#define SCREEN_PIXWIDTH    640   /* in pixels */
#define SCREEN_WIDTH       80    /* in bytes */
#define SPLIT_START_LINE   339
#define SPLIT_LINES        141
#define NONSPLIT_LINES     339
#define SPLIT_START_OFFSET 0
#define PAGE0_START_OFFSET (SPLIT_LINES*SCREEN_WIDTH)
```

```
#define PAGE1_START_OFFSET ((SPLIT_LINES+NONSPLIT_LINES)*SCREEN_WIDTH)
#define CRTC_INDEX   0x3D4 /* CRT Controller Index register */
#define CRTC_DATA    0x3D5 /* CRT Controller Data register */
#define OVERFLOW     0x07  /* index of CRTC reg holding bit 8 of the
                              line the split screen starts after */
#define MAX_SCAN     0x09  /* index of CRTC reg holding bit 9 of the
                              line the split screen starts after */
#define LINE_COMPARE 0x18  /* index of CRTC reg holding lower 8 bits
                              of line split screen starts after */
#define NUM_BUMPERS  (sizeof(Bumpers)/sizeof(bumper))
#define BOUNCER_COLOR 15
#define BACK_COLOR   1     /* playfield background color */

typedef struct {  /* one solid bumper to be bounced off of */
   int LeftX,TopY,RightX,BottomY;
   int Color;
} bumper;

typedef struct {     /* one bit pattern to be used for drawing */
   int WidthInBytes;
   int Height;
   unsigned char *BitPattern;
} image;

typedef struct {  /* one bouncing object to move around the screen */
   int LeftX,TopY;          /* location */
   int Width,Height;        /* size in pixels */
   int DirX,DirY;           /* motion vectors */
   int CurrentX[2],CurrentY[2]; /* current location in each page */
   int Color;               /* color in which to be drawn */
   image *Rotation0;        /* rotations for handling the 8 possible */
   image *Rotation1;        /* intrabyte start address at which the */
   image *Rotation2;        /* left edge can be */
   image *Rotation3;
   image *Rotation4;
   image *Rotation5;
   image *Rotation6;
   image *Rotation7;
} bouncer;

void main(void);
void DrawBumperList(bumper *, int, unsigned int);
void DrawSplitScreen(void);
void EnableSplitScreen(void);
void MoveBouncer(bouncer *, bumper *, int);
extern void DrawRect(int,int,int,int,int,unsigned int,unsigned int);
extern void ShowPage(unsigned int);
extern void DrawImage(int,int,image **,int,unsigned int,unsigned int);
extern void ShowBounceCount(void);
extern void TextUp(char *,int,int,unsigned int,unsigned int);
extern void SetBIOS8x8Font(void);

/* All bumpers in the playfield */
bumper Bumpers[] = {
   {0,0,19,339,2}, {0,0,639,19,2}, {620,0,639,339,2},
   {0,320,639,339,2}, {60,48,79,67,12}, {60,108,79,127,12},
   {60,168,79,187,12}, {60,228,79,247,12}, {120,68,131,131,13},
   {120,188,131,271,13}, {240,128,259,147,14}, {240,192,259,211,14},
   {208,160,227,179,14}, {272,160,291,179,14}, {228,272,231,319,11},
   {192,52,211,55,11}, {302,80,351,99,12}, {320,260,379,267,13},
   {380,120,387,267,13}, {420,60,579,63,11}, {428,110,571,113,11},
   {420,160,579,163,11}, {428,210,571,213,11}, {420,260,579,263,11} };

/* Image for bouncing object when left edge is aligned with bit 7 */
unsigned char _BouncerRotation0[] = {
   0xFF,0x0F,0xF0, 0xFE,0x07,0xF0, 0xFC,0x03,0xF0, 0xFC,0x03,0xF0,
   0xFE,0x07,0xF0, 0xFF,0xFF,0xF0, 0xCF,0xFF,0x30, 0x87,0xFE,0x10,
   0x07,0x0E,0x00, 0x07,0x0E,0x00, 0x07,0x0E,0x00, 0x07,0x0E,0x00,
   0x87,0xFE,0x10, 0xCF,0xFF,0x30, 0xFF,0xFF,0xF0, 0xFE,0x07,0xF0,
   0xFC,0x03,0xF0, 0xFC,0x03,0xF0, 0xFE,0x07,0xF0, 0xFF,0x0F,0xF0};
image BouncerRotation0 = {3, 20, _BouncerRotation0};

/* Image for bouncing object when left edge is aligned with bit 3 */
unsigned char _BouncerRotation4[] = {
   0x0F,0xF0,0xFF, 0x0F,0xE0,0x7F, 0x0F,0xC0,0x3F, 0x0F,0xC0,0x3F,
   0x0F,0xE0,0x7F, 0x0F,0xFF,0xFF, 0x0C,0xFF,0xF3, 0x08,0x7F,0xE1,
   0x00,0x70,0xE0, 0x00,0x70,0xE0, 0x00,0x70,0xE0, 0x00,0x70,0xE0,
   0x08,0x7F,0xE1, 0x0C,0xFF,0xF3, 0x0F,0xFF,0xFF, 0x0F,0xE0,0x7F,
   0x0F,0xC0,0x3F, 0x0F,0xC0,0x3F, 0x0F,0xE0,0x7F, 0x0F,0xF0,0xFF};
image BouncerRotation4 = {3, 20, _BouncerRotation4};

/* Initial settings for bouncing object. Only 2 rotations are needed
   because the object moves 4 pixels horizontally at a time */
bouncer Bouncer = {156,60,20,20,4,4,156,156,60,60,BOUNCER_COLOR,
   &BouncerRotation0,NULL,NULL,NULL,&BouncerRotation4,NULL,NULL,NULL};
unsigned int PageStartOffsets[2] =
   {PAGE0_START_OFFSET,PAGE1_START_OFFSET};
unsigned int BounceCount;

void main() {
   int DisplayedPage, NonDisplayedPage, Done, i;
   union REGS regset;

   regset.x.ax = 0x0012; /* set display to 640x480 16-color mode */
   int86(0x10, &regset, &regset);
   SetBIOS8x8Font();    /* set the pointer to the BIOS 8x8 font */
   EnableSplitScreen(); /* turn on the split screen */

   /* Display page 0 above the split screen */
   ShowPage(PageStartOffsets[DisplayedPage = 0]);

   /* Clear both pages to background and draw bumpers in each page */
   for (i=0; i<2; i++) {
      DrawRect(0,0,SCREEN_PIXWIDTH-1,NONSPLIT_LINES-1,BACK_COLOR,
            PageStartOffsets[i],SCREEN_SEG);
      DrawBumperList(Bumpers,NUM_BUMPERS,PageStartOffsets[i]);
   }

   DrawSplitScreen();   /* draw the static split screen info */
   BounceCount = 0;
   ShowBounceCount();   /* put up the initial zero count */

   /* Draw the bouncing object at its initial location */
   DrawImage(Bouncer.LeftX,Bouncer.TopY,&Bouncer.Rotation0,
         Bouncer.Color,PageStartOffsets[DisplayedPage],SCREEN_SEG);

   /* Move the object, draw it in the nondisplayed page, and flip the
      page until Esc is pressed */
   Done = 0;
   do {
      NonDisplayedPage = DisplayedPage ^ 1;
      /* Erase at current location in the nondisplayed page */
      DrawRect(Bouncer.CurrentX[NonDisplayedPage],
            Bouncer.CurrentY[NonDisplayedPage],
            Bouncer.CurrentX[NonDisplayedPage]+Bouncer.Width-1,
            Bouncer.CurrentY[NonDisplayedPage]+Bouncer.Height-1,
            BACK_COLOR,PageStartOffsets[NonDisplayedPage],SCREEN_SEG);
      /* Move the bouncer */
      MoveBouncer(&Bouncer, Bumpers, NUM_BUMPERS);
      /* Draw at the new location in the nondisplayed page */
      DrawImage(Bouncer.LeftX,Bouncer.TopY,&Bouncer.Rotation0,
            Bouncer.Color,PageStartOffsets[NonDisplayedPage],
            SCREEN_SEG);
      /* Remember where the bouncer is in the nondisplayed page */
      Bouncer.CurrentX[NonDisplayedPage] = Bouncer.LeftX;
      Bouncer.CurrentY[NonDisplayedPage] = Bouncer.TopY;
      /* Flip to the page we just drew into */
      ShowPage(PageStartOffsets[DisplayedPage = NonDisplayedPage]);
      /* Respond to any keystroke */
      if (kbhit()) {
         switch (getch()) {
            case 0x1B:           /* Esc to end */
               Done = 1; break;
            case 0:              /* branch on the extended code */
               switch (getch()) {
                  case 0x48:  /* nudge up */
                     Bouncer.DirY = -abs(Bouncer.DirY); break;
                  case 0x4B:  /* nudge left */
                     Bouncer.DirX = -abs(Bouncer.DirX); break;
                  case 0x4D:  /* nudge right */
                     Bouncer.DirX = abs(Bouncer.DirX); break;
                  case 0x50:  /* nudge down */
                     Bouncer.DirY = abs(Bouncer.DirY); break;
               }
               break;
            default:
               break;
         }
      }
   } while (!Done);

   /* Restore text mode and done */
   regset.x.ax = 0x0003;
   int86(0x10, &regset, &regset);
}

/* Draws the specified list of bumpers into the specified page */
void DrawBumperList(bumper * Bumpers, int NumBumpers,
      unsigned int PageStartOffset)
{
   int i;

   for (i=0; i<NumBumpers; i++,Bumpers++) {
      DrawRect(Bumpers->LeftX,Bumpers->TopY,Bumpers->RightX,
            Bumpers->BottomY,Bumpers->Color,PageStartOffset,
            SCREEN_SEG);
   }
}

/* Displays the current bounce count */
void ShowBounceCount() {
   char CountASCII[7];
   itoa(BounceCount,CountASCII,10); /* convert the count to ASCII */
   TextUp(CountASCII,344,64,SPLIT_START_OFFSET,SCREEN_SEG);
}

/* Frames the split screen and fills it with various text */
void DrawSplitScreen() {
   DrawRect(0,0,SCREEN_PIXWIDTH-1,SPLIT_LINES-1,0,SPLIT_START_OFFSET,
         SCREEN_SEG);
   DrawRect(0,1,SCREEN_PIXWIDTH-1,4,15,SPLIT_START_OFFSET,
         SCREEN_SEG);
   DrawRect(0,SPLIT_LINES-4,SCREEN_PIXWIDTH-1,SPLIT_LINES-1,15,
         SPLIT_START_OFFSET,SCREEN_SEG);
   DrawRect(0,1,3,SPLIT_LINES-1,15,SPLIT_START_OFFSET,SCREEN_SEG);
   DrawRect(SCREEN_PIXWIDTH-4,1,SCREEN_PIXWIDTH-1,SPLIT_LINES-1,15,
         SPLIT_START_OFFSET,SCREEN_SEG);
   TextUp("This is the split screen area...",8,8,SPLIT_START_OFFSET,
         SCREEN_SEG);
   TextUp("Bounces: ",272,64,SPLIT_START_OFFSET,SCREEN_SEG);
   TextUp("\033: nudge left",520,78,SPLIT_START_OFFSET,SCREEN_SEG);
   TextUp("\032: nudge right",520,90,SPLIT_START_OFFSET,SCREEN_SEG);
   TextUp("\031: nudge down",520,102,SPLIT_START_OFFSET,SCREEN_SEG);
   TextUp("\030: nudge up",520,114,SPLIT_START_OFFSET,SCREEN_SEG);
   TextUp("Esc to end",520,126,SPLIT_START_OFFSET,SCREEN_SEG);
}

/* Turn on the split screen at the desired line (minus 1 because the
   split screen starts *after* the line specified by the LINE_COMPARE
   register) (bit 8 of the split screen start line is stored in the
   Overflow register, and bit 9 is in the Maximum Scan Line reg) */
void EnableSplitScreen() {
   outp(CRTC_INDEX, LINE_COMPARE);
   outp(CRTC_DATA, (SPLIT_START_LINE - 1) & 0xFF);
   outp(CRTC_INDEX, OVERFLOW);
   outp(CRTC_DATA, (((((SPLIT_START_LINE - 1) & 0x100) >> 8) << 4) |
         (inp(CRTC_DATA) & ~0x10)));
   outp(CRTC_INDEX, MAX_SCAN);
   outp(CRTC_DATA, (((((SPLIT_START_LINE - 1) & 0x200) >> 9) << 6) |
         (inp(CRTC_DATA) & ~0x40)));
}

/* Moves the bouncer, bouncing if bumpers are hit */
void MoveBouncer(bouncer *Bouncer, bumper *BumperPtr, int NumBumpers) {
   int NewLeftX, NewTopY, NewRightX, NewBottomY, i;

   /* Move to new location, bouncing if necessary */
   NewLeftX = Bouncer->LeftX + Bouncer->DirX;   /* new coords */
   NewTopY = Bouncer->TopY + Bouncer->DirY;
   NewRightX = NewLeftX + Bouncer->Width - 1;
   NewBottomY = NewTopY + Bouncer->Height - 1;
   /* Compare the new location to all bumpers, checking for bounce */
   for (i=0; i<NumBumpers; i++,BumperPtr++) {
      /* If moving puts the bouncer inside this bumper, bounce */
      if (  (NewLeftX <= BumperPtr->RightX) &&
            (NewRightX >= BumperPtr->LeftX) &&
            (NewTopY <= BumperPtr->BottomY) &&
            (NewBottomY >= BumperPtr->TopY) ) {
         /* The bouncer has tried to move into this bumper; figure
            out which edge(s) it crossed, and bounce accordingly */
         if (((Bouncer->LeftX > BumperPtr->RightX) &&
              (NewLeftX <= BumperPtr->RightX)) ||
              (((Bouncer->LeftX + Bouncer->Width - 1) <
               BumperPtr->LeftX) &&
               (NewRightX >= BumperPtr->LeftX))) {
            Bouncer->DirX = -Bouncer->DirX;  /* bounce horizontally */
            NewLeftX = Bouncer->LeftX + Bouncer->DirX;
         }
         if (((Bouncer->TopY > BumperPtr->BottomY) &&
               (NewTopY <= BumperPtr->BottomY)) ||
               (((Bouncer->TopY + Bouncer->Height - 1) <
               BumperPtr->TopY) &&
               (NewBottomY >= BumperPtr->TopY))) {
            Bouncer->DirY = -Bouncer->DirY; /* bounce vertically */
            NewTopY = Bouncer->TopY + Bouncer->DirY;
         }
         /* Update the bounce count display; turn over at 10000 */
         if (++BounceCount >= 10000) {
            TextUp("0    ",344,64,SPLIT_START_OFFSET,SCREEN_SEG);
            BounceCount = 0;
         } else {
            ShowBounceCount();
         }
      }
   }
   Bouncer->LeftX = NewLeftX; /* set the final new coordinates */
   Bouncer->TopY = NewTopY;
}
```

LISTING 44.2 L44-2.ASM

```
; Low-level animation routines.
; Tested with TASM

SCREEN_WIDTH            equ     80      ;screen width in bytes
INPUT_STATUS_1          equ     03dah   ;Input Status 1 register
CRTC_INDEX              equ     03d4h   ;CRT Controller Index reg
START_ADDRESS_HIGH      equ     0ch     ;bitmap start address high byte
START_ADDRESS_LOW       equ     0dh     ;bitmap start address low byte
GC_INDEX                equ     03ceh   ;Graphics Controller Index reg
SET_RESET               equ     0       ;GC index of Set/Reset reg
G_MODE                  equ     5       ;GC index of Mode register

        .model  small
        .data
BIOS8x8Ptr dd   ?       ;points to BIOS 8x8 font
; Tables used to look up left and right clip masks.
LeftMask db     0ffh, 07fh, 03fh, 01fh, 00fh, 007h, 003h, 001h
RightMask db    080h, 0c0h, 0e0h, 0f0h, 0f8h, 0fch, 0feh, 0ffh

        .code
; Draws the specified filled rectangle in the specified color.
; Assumes the display is in mode 12h. Does not clip and assumes
; rectangle coordinates are valid.
;
; C near-callable as: void DrawRect(int LeftX, int TopY, int RightX,
;       int BottomY, int Color, unsigned int ScrnOffset,
;       unsigned int ScrnSegment);

DrawRectParms   struc
           dw      2 dup (?);pushed BP and return address
LeftX      dw      ?              ;X coordinate of left side of rectangle
TopY       dw      ?              ;Y coordinate of top side of rectangle
RightX     dw      ?              ;X coordinate of right side of rectangle
```

```
BottomY     dw      ?               ;Y coordinate of bottom side of rectangle
Color       dw      ?               ;color in which to draw rectangle (only the
                                    ; lower 4 bits matter)
ScrnOffset       dw    ?            ;offset of base of bitmap in which to draw
ScrnSegment      dw    ?            ;segment of base of bitmap in which to draw
DrawRectParms   ends

        public  _DrawRect
_DrawRect       proc    near
        push    bp                  ;preserve caller's stack frame
        mov     bp,sp               ;point to local stack frame
        push    si                  ;preserve caller's register variables
        push    di

        cld
        mov     dx,GC_INDEX
        mov     al,SET_RESET
        mov     ah,byte ptr Color[bp]
        out     dx,ax               ;set the color in which to draw
        mov     ax,G_MODE + (0300h)
        out     dx,ax               ;set to write mode 3
        les     di,dword ptr ScrnOffset[bp] ;point to bitmap start
        mov     ax,SCREEN_WIDTH
        mul     TopY[bp]            ;point to the start of the top scan
        add     di,ax               ; line to fill
        mov     ax,LeftX[bp]
        mov     bx,ax
        shr     ax,1                ;/8 = byte offset from left of screen
        shr     ax,1
        shr     ax,1
        add     di,ax               ;point to the upper-left corner of fill area
        and     bx,7                ;isolate intrapixel address
        mov     dl,LeftMask[bx]     ;set the left-edge clip mask
        mov     bx,RightX[bp]
        mov     si,bx
        and     bx,7                ;isolate intrapixel address of right edge
        mov     dh,RightMask[bx]    ;set the right-edge clip mask
        mov     bx,LeftX[bp]
        and     bx,NOT 7            ;intrapixel address of left edge
        sub     si,bx
        shr     si,1
        shr     si,1
        shr     si,1                ;# of bytes across spanned by rectangle - 1
        jnz     MasksSet            ;if there's only one byte across,
        and     dl,dh               ; combine the masks
MasksSet:
        mov     bx,BottomY[bp]
        sub     bx,TopY[bp]         ;# of scan lines to fill - 1
FillLoop:
        push    di                  ;remember line start offset
        mov     al,dl               ;left edge clip mask
        xchg    es:[di],al          ;draw the left edge
        inc     di                  ;point to the next byte
        mov     cx,si               ;# of bytes left to do
        dec     cx                  ;# of bytes left to do - 1
        js      LineDone            ;that's it if there's only 1 byte across
        jz      DrawRightEdge       ;no middle bytes if only 2 bytes across
        mov     al,0ffh             ;non-edge bytes are solid
        rep     stosb               ;draw the solid bytes across the middle
```

```
DrawRightEdge:
        mov     al,dh               ;right-edge clip mask
        xchg    es:[di],al          ;draw the right edge
LineDone:
        pop     di                  ;retrieve line start offset
        add     di,SCREEN_WIDTH     ;point to the next line
        dec     bx                  ;count off scan lines
        jns     FillLoop

        pop     di                  ;restore caller's register variables
        pop     si
        pop     bp                  ;restore caller's stack frame
        ret
_DrawRect       endp

; Shows the page at the specified offset in the bitmap. Page is
; displayed when this routine returns.
;
; C near-callable as: void ShowPage(unsigned int StartOffset);

ShowPageParms   struc
        dw      2 dup (?)           ;pushed BP and return address
StartOffset dw  ?           ;offset in bitmap of page to display
ShowPageParms   ends

        public  _ShowPage
_ShowPage       proc    near
        push    bp                  ;preserve caller's stack frame
        mov     bp,sp               ;point to local stack frame
; Wait for display enable to be active (status is active low), to be
; sure both halves of the start address will take in the same frame.
        mov     bl,START_ADDRESS_LOW          ;preload for fastest
        mov     bh,byte ptr StartOffset[bp]   ; flipping once display
        mov     cl,START_ADDRESS_HIGH         ; enable is detected
        mov     ch,byte ptr StartOffset+1[bp]
        mov     dx,INPUT_STATUS_1
WaitDE:
        in      al,dx
        test    al,01h
        jnz     WaitDE              ;display enable is active low (0 = active)
; Set the start offset in display memory of the page to display.
        mov     dx,CRTC_INDEX
        mov     ax,bx
        out     dx,ax               ;start address low
        mov      ax,cx
        out     dx,ax               ;start address high
; Now wait for vertical sync, so the other page will be invisible when
; we start drawing to it.
        mov     dx,INPUT_STATUS_1
WaitVS:
        in      al,dx
        test    al,08h
        jz      WaitVS              ;vertical sync is active high (1 = active)
        pop     bp                  ;restore caller's stack frame
        ret
_ShowPage       endp

; Displays the specified image at the specified location in the
; specified bitmap, in the desired color.
;
```

```
; C near-callable as: void DrawImage(int LeftX, int TopY,
;       image **RotationTable, int Color, unsigned int ScrnOffset,
;       unsigned int ScrnSegment);

DrawImageParms  struc
           dw      2 dup (?);pushed BP and return address
DILeftX          dw    ?           ;X coordinate of left side of image
DITopY           dw    ?           ;Y coordinate of top side of image
RotationTable    dw    ?           ;pointer to table of pointers to image
                                   ; rotations
DIColor          dw    ?           ;color in which to draw image (only the
                                   ; lower 4 bits matter)
DIScrnOffset     dw    ?           ;offset of base of bitmap in which to draw
DIScrnSegment    dw    ?           ;segment of base of bitmap in which to draw
DrawImageParms  ends

image struc
WidthInBytes    dw      ?
Height          dw      ?
BitPattern      dw      ?
image ends

        public  _DrawImage
_DrawImage      proc    near
        push    bp                 ;preserve caller's stack frame
        mov     bp,sp              ;point to local stack frame
        push    si                 ;preserve caller's register variables
        push    di

        cld
        mov     dx,GC_INDEX
        mov     al,SET_RESET
        mov     ah,byte ptr DIColor[bp]
        out     dx,ax              ;set the color in which to draw
        mov     ax,G_MODE + (0300h)
        out     dx,ax              ;set to write mode 3
        les     di,dword ptr DIScrnOffset[bp] ;point to bitmap start
        mov     ax,SCREEN_WIDTH
        mul     DITopY[bp]         ;point to the start of the top scan
        add     di,ax              ; line on which to draw
        mov     ax,DILeftX[bp]
        mov     bx,ax
        shr     ax,1               ;/8 - byte offset from left of screen
        shr     ax,1
        shr     ax,1
        add     di,ax              ;point to the upper-left corner of draw area
        and     bx,7               ;isolate intrapixel address
        shl     bx,1               ;*2 for word look-up
        add     bx,RotationTable[bp] ;point to the image structure for
        mov     bx,[bx]              ; the intrabyte rotation
        mov     dx,[bx].WidthInBytes ;image width
        mov     si,[bx].BitPattern   ;pointer to image pattern bytes
        mov     bx,[bx].Height       ;image height
DrawImageLoop:
        push    di                 ;remember line start offset
        mov     cx,dx              ;# of bytes across
DrawImageLineLoop:
        lodsb                      ;get the next image byte
        xchg    es:[di],al         ;draw the next image byte
        inc     di                 ;point to the following screen byte
        loop    DrawImageLineLoop
        pop     di                      ;retrieve line start offset
        add     di,SCREEN_WIDTH         ;point to the next line
        dec     bx                      ;count off scan lines
        jnz     DrawImageLoop

        pop     di                      ;restore caller's register variables
        pop     si
        pop     bp                      ;restore caller's stack frame
        ret
_DrawImage      endp

; Draws a 0-terminated text string at the specified location in the
; specified bitmap in white, using the 8x8 BIOS font. Must be at an X
; coordinate that's a multiple of 8.
;
; C near-callable as: void TextUp(char *Text, int LeftX, int TopY,
;       unsigned int ScrnOffset, unsigned int ScrnSegment);

TextUpParms     struc
                 dw     2 dup (?);pushed BP and return address
Text             dw     ?               ;pointer to text to draw
TULeftX          dw     ?               ;X coordinate of left side of rectangle
                                        ; (must be a multiple of 8)
TUTopY           dw     ?               ;Y coordinate of top side of rectangle
TUScrnOffset     dw     ?               ;offset of base of bitmap in which to draw
TUScrnSegment    dw     ?               ;segment of base of bitmap in which to draw
TextUpParms     ends

        public  _TextUp
_TextUp proc    near
        push    bp                      ;preserve caller's stack frame
        mov     bp,sp                   ;point to local stack frame
        push    si                      ;preserve caller's register variables
        push    di

        cld
        mov     dx,GC_INDEX
        mov     ax,G_MODE + (0000h)
        out     dx,ax                   ;set to write mode 0
        les     di,dword ptr TUScrnOffset[bp] ;point to bitmap start
        mov     ax,SCREEN_WIDTH
        mul     TUTopY[bp]              ;point to the start of the top scan
        add     di,ax                   ; line the text starts on
        mov     ax,TULeftX[bp]
        mov     bx,ax
        shr     ax,1                    ;/8 - byte offset from left of screen
        shr     ax,1
        shr     ax,1
        add     di,ax                   ;point to the upper-left corner of first char
        mov     si,Text[bp]             ;point to text to draw
TextUpLoop:
        lodsb                           ;get the next character to draw
        and     al,al
        jz      TextUpDone              ;done if null byte
        push    si                      ;preserve text string pointer
        push    di                      ;preserve character's screen offset
        push    ds                      ;preserve default data segment
        call    CharUp                  ;draw this character
        pop     ds                      ;restore default data segment
        pop     di                      ;retrieve character's screen offset
        pop     si                      ;retrieve text string pointer
        inc     di                      ;point to next character's start location
        jmp     TextUpLoop

TextUpDone:
        pop     di                      ;restore caller's register variables
        pop     si
        pop     bp                      ;restore caller's stack frame
        ret

CharUp:                                 ;draws the character in AL at ES:DI
        lds     si,[BIOS8x8Ptr]         ;point to the 8x8 font start
        mov     bl,al
        sub     bh,bh
        shl     bx,1
        shl     bx,1
        shl     bx,1                    ;*8 to look up character offset in font
        add     si,bx                   ;point DS:SI to character data in font
        mov     cx,8                    ;characters are 8 high
CharUpLoop:
        movsb                           ;copy the next character pattern byte
        add     di,SCREEN_WIDTH-1       ;point to the next dest byte
        loop    CharUpLoop
        ret
_TextUp endp

; Sets the pointer to the BIOS 8x8 font.
;
; C near-callable as: extern void SetBIOS8x8Font(void);

        public  _SetBIOS8x8Font
_SetBIOS8x8Font proc    near
        push    bp                      ;preserve caller's stack frame
        push    si                      ;preserve caller's register variables
        push    di                      ; and data segment (don't assume BIOS
        push    ds                      ; preserves anything)
        mov     ah,11h                  ;BIOS character generator function
        mov     al,30h                  ;BIOS information subfunction
        mov     bh,3                    ;request 8x8 font pointer
        int     10h                     ;invoke BIOS video services
        mov     word ptr [BIOS8x8Ptr],bp  ;store the pointer
        mov     word ptr [BIOS8x8Ptr+2],es
        pop     ds
        pop     di                      ;restore caller's register variables
        pop     si
        pop     bp                      ;restore caller's stack frame
        ret
_SetBIOS8x8Font endp
        end
```

Listing 44.1 is written in C. It could equally well have been written in assembly language, and would then have been somewhat faster. However, I wanted to make the point (as I've made again and again) that assembly language, and, indeed, optimization in general, is needed only in the most critical portions of any program, and then only when the program would otherwise be too slow. Only in a highly performance-sensitive situation would the performance boost resulting from converting Listing 44.1 to assembly justify the time spent in coding and the bugs that would likely creep in—and the sample program already updates the screen at the maximum possible rate of once per frame even on a 1985-vintage 8-MHz AT. In this case, faster performance would result only in a longer wait for the page to flip.

Write Mode 3

It's possible to update the bitmap very efficiently on the VGA, because the VGA can draw up to 8 pixels at once, and because the VGA provides a number of hardware features to speed up drawing. This article makes considerable use of one particularly unusual hardware feature, write mode 3. We discussed write mode 3 back in Chapter 26, but we've covered a lot of ground since then—so I'm going to run through a quick refresher on write mode 3.

Some background: In the standard VGA's high-resolution mode, mode 12H (640×480 with 16 colors, the mode in which this chapter's sample program runs), each byte of display memory controls 8 adjacent pixels on the screen. (The color of each pixel is, in turn, controlled by 4 bits spread across the four VGA memory planes, but we need not concern ourselves with that here.) Now, there will often be times when we want to change some but not all of the pixels controlled by a particular byte of display memory. This is not easily done, for there is no way to write half a byte, or two bits, or such to memory; it's the whole byte or none of it at all.

You might think that using AND and OR to manipulate individual bits could solve the problem. Alas, not so. ANDing and ORing would work if the VGA had only one plane of memory (like the original monochrome Hercules Graphics Adapter) but the VGA has four planes, and ANDing and ORing would work only if we selected and manipulated each plane separately, a process that would be hideously slow. No, with the VGA you must use the hardware assist features, or you might as well forget about real-time screen updates altogether. Write mode 3 will do the trick for our present needs.

Write mode 3 is useful when you want to set some but not all of the pixels in a single byte of display memory *to the same color*. That is, if you want to draw a number of pixels within a byte in a single color, write mode 3 is a good way to do it.

Write mode 3 works like this. First, set the Graphics Controller Mode register to write mode 3. (Look at Listing 44.2 for code that does everything described here.) Next, set the Set/Reset register to the color with which you wish to draw, in the range 0-15. (It is not necessary to explicitly enable set/reset via the Enable Set/Reset register; write mode 3 does that automatically.) Then, to draw individual pixels within a single byte, simply read display memory, and then write a byte to display memory with 1-bits where you want the color to be drawn and 0-bits where you want the current bitmap contents to be preserved. (Note well that *the data actually read by the CPU doesn't matter*; the read operation latches all four planes' data, as described way back in Chapter 24.) So, for example, if write mode 3 is enabled and the Set/Reset register is set to 1 (blue), then the following sequence of operations:

```
mov   dx,0a000h
mov   es,dx
mov   al,es:[0]
mov   byte ptr es:[0],0f0h
```

will change the first 4 pixels on the screen (the left nibble of the byte at offset 0 in display memory) to blue, and will leave the next 4 pixels (the right nibble of the byte at offset 0) unchanged.

Using one **MOV** to read from display memory and another to write to display memory is not particularly efficient on some processors. In Listing 44.2, I instead use **XCHG**, which reads and then writes a memory location in a single operation, as in:

```
mov   dx,0a000h
mov   es,dx
mov   al,0f0h
xchg  es:[0],al
```

Again, the actual value that's read is irrelevant. In general, the **XCHG** approach is more compact than two **MOV**s, and is faster on 386 and earlier processors, but slower on 486s and Pentiums.

If all pixels in a byte of display memory are to be drawn in a single color, it's not necessary to read before writing, because none of the information in display memory at that byte needs to be preserved; a simple write of 0FFH (to draw all bits) will set all 8 pixels to the set/reset color:

```
mov   dx,0a000h
mov   es,dx
mov   byte ptr es:[di],0ffh
```

*If you're familiar with VGA programming, you're no doubt aware that everything that can be done with write mode 3 can also be accomplished in write mode 0 or write mode 2 by using the Bit Mask register. However, setting the Bit Mask register requires at least one **OUT** per byte written, in addition to the read and write of display memory, and **OUT**s are often slower than display memory accesses, especially on 386s and 486s. One of the great virtues of write mode 3 is that it requires virtually no **OUT**s and is therefore substantially faster for masking than the other write modes.*

In short, write mode 3 is a good choice for single-color drawing that modifies individual pixels within display memory bytes. Not coincidentally, the sample application draws only single-color objects within the animation area; this allows write mode 3 to be used for all drawing, in keeping with our desire for speedy screen updates.

Drawing Text

We'll need text in the sample application; is that also a good use for write mode 3? Sometimes it is, but not in this particular case.

Each character in a font is represented by a pattern of bits, with 1-bits representing character pixels and 0-bits representing background pixels. Since we'll be using the 8x8 font stored in the BIOS ROM (a pointer to which can be obtained by calling a BIOS service, as illustrated by Listing 44.2), each character is exactly 8 bits, or 1 byte wide. We'll further insist that characters be placed on byte boundaries (that is, with their left edges only at pixels with X coordinates that are multiples of 8); this means that the character bytes in the font are automatically aligned with display memory, and no rotation or clipping of characters is needed. Finally, we'll draw all text in white.

Given the above assumptions, drawing text is easy; we simply copy each byte of each character to the appropriate location in display memory, and *voila*, we're done. Text copying is done in write mode 0, in which the byte written to display memory is copied to all four planes at once; hence, 1-bits turn into white (color value 0FH, with 1-bits in all four planes), and 0-bits turn into black (color value 0). This is faster than using write mode 3 because write mode 3 requires a read/write of display memory (or at least preloading the latches with the background color), while the write mode 0 approach requires only a write to display memory.

Is write mode 0 always the best way to do text? Not at all. The write mode 0 approach described above draws both foreground and background pixels within the character box, forcing the background pixels to black at the same time that it forces the foreground pixels to white. If you want to draw transparent text (that is, draw only the character pixels, not the surrounding background box), write mode 3 is ideal. Also, matters get far more complicated if characters that aren't 8 pixels wide are drawn, or if characters are drawn starting at arbitrary pixel locations, without the multiple-of-8 column restriction, so that rotation and masking are required. Lastly, the Map Mask register can be used to draw text in colors other than white—but only if the background is black. Otherwise, the data remaining in the planes protected by the Map Mask will remain and can interfere with the colors of the text being drawn.

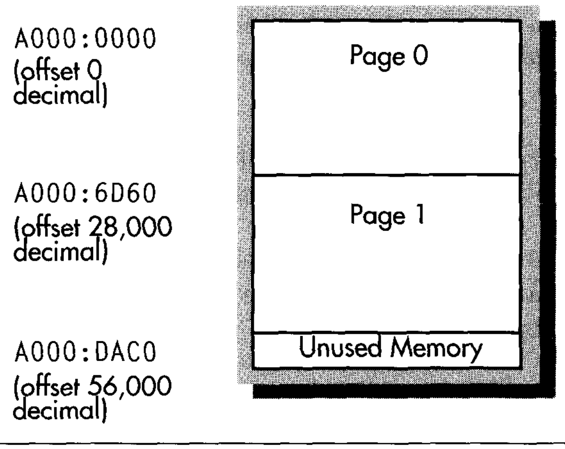
I'm not going to delve any deeper into the considerable issues of drawing VGA text; I just want to sensitize you to the existence of approaches other than the ones used in Listings 44.1 and 44.2. On the VGA, the rule is: If there's something you want to do, there probably are 10 ways to do it, each with unique strengths and weaknesses. Your mission, should you decide to accept it, is to figure out which one is best for your particular application.

Page Flipping

Now that we know how to update the screen reasonably quickly, it's time to get on to the fun stuff. Page flipping answers the second requirement for animation, by keeping bitmap changes off the screen until they're complete. In other words, page flipping guarantees that partially updated bitmaps are never seen.

How is it possible to update a bitmap without seeing the changes as they're made? Easy—with page flipping, there are *two* bitmaps; the program shows you one bitmap while it updates the other. Conceptually, it's that simple. In practice, unfortunately, it's not so simple, because of the design of the VGA. To understand why that is, we must look at how the VGA turns bytes in display memory into pixels on the screen.

The VGA bitmap is a linear 64 K block of memory. (True, most adapters nowadays are SuperVGAs with more than 256 K of display memory, but every make of SuperVGA has its own way of letting you access that extra memory, so going beyond standard VGA is a daunting and difficult task. Also, it's hard to manipulate the large frame buffers of SuperVGA modes fast enough for real-time animation.) Normally, the VGA picks up the first byte of memory (the byte at offset 0) and displays the corresponding 8 pixels on the screen, then picks up the byte at offset 1 and displays the next 8 pixels, and so on to the end of the screen. However, the offset of the first byte of display memory picked up during each frame is not fixed at 0, but is rather programmable by way of the Start Address High and Low registers, which together store the 16-bit offset in display memory at which the bitmap to be displayed during the next frame starts. So, for example, in mode 10H (640×350, 16 colors), a large enough bitmap to store a complete screen of information can be stored at display memory offsets 0 through 27,999, and *another* full bitmap could be stored at offsets 28,000 through 55,999, as shown in Figure 44.1. (I'm discussing 640×350 mode at the moment for good reason; we'll get to 640×480 shortly.) When the Start Address registers are set to 0, the first bitmap (or page) is displayed; when they are set to 28,000, the second bitmap is displayed. Page flipped animation can be performed by displaying



Memory allocation for mode 10h page flipping.
Figure 44.1

page 0 and drawing to page 1, then setting the start address to page 1 to display that page and drawing to page 0, and so on *ad infinitum.*

Knowing When to Flip

There's a hitch, though, and that hitch is knowing exactly when it is that the page has flipped. The page doesn't flip the instant that you set the Start Address registers. The VGA loads the starting offset from the Start Address registers once before starting each frame, then pays those registers no nevermind until the next frame comes around. This means that you can set the Start Address registers whenever you want—but the page actually being displayed doesn't change until after the VGA loads that new offset in preparation for the next frame.

The potential problem should be obvious. Suppose that page 1 is being displayed, and you're updating page 0. You finish drawing to page 0, set the Start Address registers to 0 to switch to displaying page 0, and start updating page 1, which is no longer displayed. Or is it? If the VGA was in the middle of the current frame, displaying page 1, when you set the Start Address registers, then page 1 is going to be displayed for the rest of the frame, no matter what you do with the Start Address registers. If you start updating page 1 right away, any changes you make may well show up on the screen, because page 0 hasn't yet flipped to being displayed in place of page 1—and that defeats the whole purpose of page flipping.

To avoid this problem, it is mandatory that you wait until you're sure the page has flipped. The Start Address registers are, according to my tests, loaded at the start of the Vertical Sync signal, although that may not be the case with all VGA clones. The Vertical Sync status is provided as bit 3 of the Input Status 1 register, so it would seem that all you need to do to flip a page is set the new Start Address registers, wait for the start of the Vertical Sync pulse that indicates that the page has flipped, and be on your merry way.

Almost—but not quite. (Do I hear teeth gnashing in the background?) The problem is this: Suppose that, by coincidence, you set one of the Start Address registers just before the start of Vertical Sync, and the other right after the start of Vertical Sync. Why, then, for one frame the Start Address High value for one page would be mixed with the Start Address Low value for the other page, and, depending on the start address values, the whole screen could appear to shift any number of pixels for a single, horrible frame. *This must never happen!* The solution is to set the Start Address registers when you're certain Vertical Sync is not about to start. The easiest way to know that is to check for the Display Enable status (bit 0 of the Input Status 1 register) being active; that means that bitmap-controlled pixels are being scanned onto the screen, and, since Vertical Sync happens in the middle of the vertical non-display portion of the frame, Vertical Sync can never be anywhere nearby if Display Enable is active. (Note that one good alternative is to set up both pages with a start address that's a multiple of 256, and just change the Start Address High register and wait for Vertical Sync, with no Display Enable wait required.)

So, to flip pages, you must complete all drawing to the non-displayed page, wait for Display Enable to be active, set the new start address, and wait for Vertical Sync to be active. At that point, you can be fully confident that the page that you just flipped off the screen is not displayed and can safely (invisibly) be updated. A side benefit of page flipping is that your program will automatically have a constant time base, with the rate at which new screens are drawn synchronized to the frame rate of the display (typically 60 or 70 Hz). However, complex updates may take more than one frame to complete, especially on slower processors; this can be compensated for by maintaining a count of new screens drawn and cross-referencing that to the BIOS timer count periodically, accelerating the overall pace of the animation (moving farther each time and the like) if updates are happening too slowly.

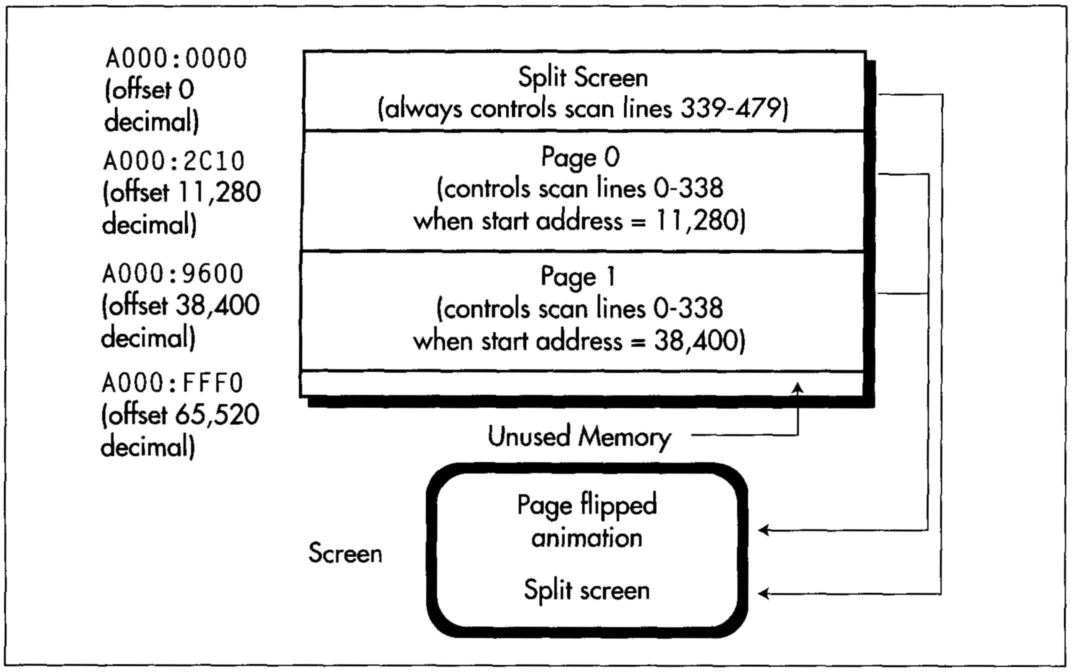
Enter the Split Screen

So far, I've discussed page flipping in 640×350 mode. There's a reason for that: 640×350 is the highest-resolution standard mode in which there's enough display memory for two full pages on a standard VGA. It's possible to program the VGA to a non-standard 640×400 mode and still have two full pages, but that's pretty much the limit. One 640×480 page takes 38,400 bytes of display memory, and clearly there isn't enough room in 64 K of display memory for two of *those* monster pages.

And yet, 640×480 is a wonderful mode in many ways. It offers a 1:1 aspect ratio (square pixels), and it provides by far the best resolution of any 16-color mode. Surely there's *some* way to bring the visual appeal of page flipping to this mode?

Surely there is—but it's an odd solution indeed. The VGA has a feature, known as the *split screen*, that allows you to force the offset from which the VGA fetches video data back to 0 after any desired scan line. For example, you can program the VGA to scan through display memory as usual until it finishes scan line number 338, and then get the first byte of information for scan line number 339 from offset 0 in display memory.

That, in turn, allows us to divvy up display memory into three areas, as shown in Figure 44.2. The area from 0 to 11,279 is reserved for the split screen, the area from 11,280 to 38,399 is used for page 0, and the area from 38,400 to 65,519 is used for page 1. This allows page flipping to be performed in the top 339 scan lines (about 70 percent) of the screen, and leaves the bottom 141 scan lines for non-animation purposes, such as showing scores, instructions, statuses, and suchlike. (Note that the allocation of display memory and number of scan lines are dictated by the desire to have as many page-flipped scan lines as possible; you may, if you wish, have fewer page-flipped lines and reserve part of the bitmap for other uses, such as off-screen storage for images.)


A000:0000 (offset 0 decimal)
A000:2C10 (offset 11,280 decimal)
A000:9600 (offset 38,400 decimal)
A000:FFF0 (offset 65,520 decimal)
Split Screen (always controls scan lines 339-479)
Page 0 (controls scan lines 0-338 when start address = 11,280)
Page 1 (controls scan lines 0-338 when start address = 38,400)
Unused Memory
Screen
Page flipped animation
Split screen

Memory allocation for mode 12h page flipping.
Figure 44.2

The sample program for this chapter uses the split screen and page flipping exactly as described above. The playfield through which the object bounces is the page-flipped portion of the screen, and the rectangle at the bottom containing the bounce count and the instructions is the split (that is, not animatable) portion of the screen. Of course, to the user it all looks like one screen. There are no visible boundaries between the two unless you choose to create them.

Very few animation applications use the entire screen for animation. If you can get by with 339 scan lines of animation, split-screen page flipping gives you the best combination of square pixels and high resolution possible on a standard VGA.

So. Is VGA animation worth all the fuss? *Mais oui.* Run the sample program; if you've never seen aggressive VGA animation before, you'll be amazed at how smooth it can be. Not every square millimeter of every animated screen must be in constant motion. Most graphics screens need a little quiet space to display scores, coordinates, file names, or (if all else fails) company logos. If you don't tell the user he's/she's only getting 339 scan lines of animation, he'll/she'll probably never know.