Chapter 49

Mode X 256-Color Animation

Chapter 49

How to Make the VGA Really Get up and Dance

Okay—no amusing stories or informative anecdotes to kick off this chapter; lotta ground to cover, gotta hurry—you're impatient, I can smell it. I won't talk about the time a friend made the mistake of loudly saying "$100 bill" during an animated discussion while walking among the bums on Market Street in San Francisco one night, thereby graphically illustrating that context is everything. I can't spare a word about how my daughter thinks my 11-year-old floppy-disk-based CP/M machine is more powerful than my 386 with its 100-MB hard disk because the CP/M machine's word processor loads and runs twice as fast as the 386's Windows-based word processor, demonstrating that progress is not the neat exponential curve we'd like to think it is, and that features and performance are often conflicting notions. And, lord knows, I can't take the time to discuss the habits of small white dogs, notwithstanding that such dogs seem to be relevant to just about every aspect of computing, as Jeff Duntemann's writings make manifest. No lighthearted fluff for us; we have real work to do, for today we animate with 256 colors in Mode X.

Masked Copying

Over the past two chapters, we've put together most of the tools needed to implement animation in the VGA's undocumented 320x240 256-color Mode X. We now have mode set code, solid and 4x4 pattern fills, system memory-to-display memory block copies, and display memory-to-display memory block copies. The final piece of the puzzle is the ability to copy a nonrectangular image to display memory. I call this *masked copying.*

Masked copying is sort of like drawing through a stencil, in that only certain pixels within the destination rectangle are drawn. The objective is to fit the image seamlessly into the background, without the rectangular fringe that results when nonrectangular images are drawn by block copying their bounding rectangle. This is accomplished by using a second rectangular bitmap, separate from the image but corresponding to it on a pixel-by-pixel basis, to control which destination pixels are set from the source and which are left unchanged. With a masked copy, only those pixels properly belonging to an image are drawn, and the image fits perfectly into the background, with no rectangular border. In fact, masked copying even makes it possible to have transparent areas within images.

Note that another way to achieve this effect is to implement copying code that supports a transparent color; that is, a color that doesn't get copied but rather leaves the destination unchanged. Transparent copying makes for more compact images, because no separate mask is needed, and is generally faster in a software-only implementation. However, Mode X supports masked copying but not transparent copying in hardware, so we'll use masked copying in this chapter.

The system memory to display memory masked copy routine in Listing 49.1 implements masked copying in a straightforward fashion. In the main drawing loop, the corresponding mask byte is consulted as each image pixel is encountered, and the image pixel is copied only if the mask byte is nonzero. As with most of the system-to-display code I've presented, Listing 49.1 is not heavily optimized, because it's inherently slow; there's a better way to go when performance matters, and that's to use the VGA's hardware.

LISTING 49.1 L49-1.ASM

```
; Mode X (320x240, 256 colors) system memory-to-display memory masked copy
; routine. Not particularly fast; images for which performance is critical
; should be stored in off-screen memory and copied to screen via latches. Works
; on all VGAs. Copies up to but not including column at SourceEndX and row at
; SourceEndY. No clipping is performed. Mask and source image are both byte-
; per-pixel, and must be of same widths and reside at same coordinates in their
; respective bitmaps. Assembly code tested with TASM C near-callable as:
;
;    void CopySystemToScreenMaskedX(int SourceStartX,
;       int SourceStartY, int SourceEndX, int SourceEndY,
;       int DestStartX, int DestStartY, char * SourcePtr,
;       unsigned int DestPageBase, int SourceBitmapWidth,
;       int DestBitmapWidth, char * MaskPtr);

SC_INDEX        equ   03c4h           ;Sequence Controller Index register port
MAP_MASK        equ   02h             ;index in SC of Map Mask register
SCREEN_SEG      equ   0a000h          ;segment of display memory in mode X

parms   struc
                dw    2 dup (?)       ;pushed BP and return address
SourceStartX    dw    ?               ;X coordinate of upper left corner of source
                                      ; (source is in system memory)
```

```
SourceStartY     dw    ?                ;Y coordinate of upper left corner of source
SourceEndX       dw    ?                ;X coordinate of lower right corner of source
                                        ; (the column at EndX is not copied)
SourceEndY       dw    ?                ;Y coordinate of lower right corner of source
                                        ; (the row at EndY is not copied)
DestStartX       dw    ?                ;X coordinate of upper left corner of dest
                                        ; (destination is in display memory)
DestStartY       dw    ?                ;Y coordinate of upper left corner of dest
SourcePtr        dw    ?                ;pointer in DS to start of bitmap which source resides
DestPageBase     dw    ?                ;base offset in display memory of page in
                                        ; which dest resides
SourceBitmapWidth  dw  ?                ;# of pixels across source bitmap (also must
                                        ; be width across the mask)
DestBitmapWidth    dw  ?                ;# of pixels across dest bitmap (must be multiple of 4)
MaskPtr            dw  ?                ;pointer in DS to start of bitmap in which mask
                                        ; resides (byte-per-pixel format, just like the source
                                        ; image; 0-bytes mean don't copy corresponding source
                                        ; pixel, 1-bytes mean do copy)
parms   ends

RectWidth        equu  -2               ;local storage for width of rectangle
RectHeight       equ   -4               ;local storage for height of rectangle
LeftMask         equ   -6               ;local storage for left rect edge plane mask
STACK_FRAME_SIZE equ 6
        .model  small
        .code
        public  _CopySystemToScreenMaskedX
_CopySystemToScreenMaskedX proc    near
        push    bp                      ;preserve caller's stack frame
        mov     bp,sp                   ;point to local stack frame
        sub     sp,STACK_FRAME_SIZE     ;allocate space for local vars
        push    si                      ;preserve caller's register variables
        push    di

        mov     ax,SCREEN_SEG           ;point ES to display memory
        mov     es,ax
        mov     ax,[bp+SourceBitmapWidth]
        mul     [bp+SourceStartY]       ;top source rect scan line
        add     ax,[bp+SourceStartX]
        mov     bx,ax
        add     ax,[bp+SourcePtr]       ;offset of first source rect pixel
        mov     si,ax                   ; in DS
        add     bx,[bp+MaskPtr]         ;offset of first mask pixel in DS

        mov     ax,[bp+DestBitmapWidth]
        shr     ax,1                    ;convert to width in addresses
        shr     ax,1
        mov     [bp+DestBitmapWidth],ax ;remember address width
        mul     [bp+DestStartY]         ;top dest rect scan line
        mov     di,[bp+DestStartX]
        mov     cx,di
        shr     di,1                    ;X/4 = offset of first dest rect pixel in
        shr     di,1                    ; scan line
        add     di,ax                   ;offset of first dest rect pixel in page
        add     di,[bp+DestPageBase]    ;offset of first dest rect pixel
                                        ; in display memory
        and     cl,011b                 ;CL = first dest pixel's plane
        mov     al,11h                  ;upper nibble comes into play when plane wraps
                                        ; from 3 back to 0
        shl     al,cl                   ;set the bit for the first dest pixel's plane
        mov     [bp+LeftMask],al        ; in each nibble to 1
```

```
        mov     ax,[bp+SourceEndX]          ;calculate # of pixels across
        sub     ax,[bp+SourceStartX]        ; rect
        jle     CopyDone                    ;skip if 0 or negative width
        mov     [bp+RectWidth],ax
        sub     word ptr [bp+SourceBitmapWidth],ax
                ;distance from end of one source scan line to start of next
        mov     ax,[bp+SourceEndY]
        sub     ax,[bp+SourceStartY]        ;height of rectangle
        jle     CopyDone                    ;skip if 0 or negative height
        mov     [bp+RectHeight],ax
        mov     dx,SC_INDEX                 ;point to SC Index register
        mov     al,MAP_MASK
        out     dx,al                       ;point SC Index reg to the Map Mask
        inc     dx                          ;point DX to SC Data reg
CopyRowsLoop:
        mov     al,[bp+LeftMask]
        mov     cx,[bp+RectWidth]
        push    di                          ;remember the start offset in the dest
CopyScanLineLoop:
        cmp     byte ptr [bx],0             ;is this pixel mask-enabled?
        jz      MaskOff                     ;no, so don't draw it
                                            ;yes, draw the pixel
        out     dx,al                       ;set the plane for this pixel
        mov     ah,[si]                     ;get the pixel from the source
        mov     es:[di],ah                  ;copy the pixel to the screen
MaskOff:
        inc     bx                          ;advance the mask pointer
        inc     si                          ;advance the source pointer
        rol     al,1                        ;set mask for next pixel's plane
        adc     di,0                        ;advance destination address only when
                                            ; wrapping from plane 3 to plane 0
        loop    CopyScanLineLoop
        pop     di                          ;retrieve the dest start offset
        add     di,[bp+DestBitmapWidth]     ;point to the start of the
                                            ; next scan line of the dest
        add     si,[bp+SourceBitmapWidth]   ;point to the start of the
                                            ; next scan line of the source
        add     bx,[bp+SourceBitmapWidth]   ;point to the start of the
                                            ; next scan line of the mask
        dec     word ptr [bp+RectHeight]    ;count down scan lines
        jnz     CopyRowsLoop
CopyDone:
        pop     di                          ;restore caller's register variables
        pop     si
        mov     sp,bp                       ;discard storage for local variables
        pop     bp                          ;restore caller's stack frame
        ret
_CopySystemToScreenMaskedX endp
        end
```

Faster Masked Copying

In the previous chapter we saw how the VGA's latches can be used to copy four pixels at a time from one area of display memory to another in Mode X. We've further seen that in Mode X the Map Mask register can be used to select which planes are copied. That's all we need to know to be able to perform fast masked copies; we can store an image in off-screen display memory, and set the Map Mask to the appropriate mask value as up to four pixels at a time are copied.

There's a slight hitch, though. The latches can only be used when the source and destination left edge coordinates, modulo four, are the same, as explained in the previous chapter. The solution is to copy all four possible alignments of each image to display memory, each properly positioned for one of the four possible destination-left-edge-modulo-four cases. These aligned images must be accompanied by the four possible alignments of the image mask, stored in system memory. Given all four image and mask alignments, masked copying is a simple matter of selecting the alignment that's appropriate for the destination's left edge, then setting the Map Mask with the 4-bit mask corresponding to each four-pixel set as we copy four pixels at a time via the latches.

Listing 49.2 performs fast masked copying. This code expects to receive a pointer to a **MaskedImage** structure, which in turn points to four **AlignedMaskedImage** structures that describe the four possible image and mask alignments. The aligned images are already stored in display memory, and the aligned masks are already stored in system memory; further, the masks are predigested into Map Mask register-compatible form. Given all that ready-to-use data, Listing 49.2 selects and works with the appropriate image-mask pair for the destination's left edge alignment.

LISTING 49.2 L49-2.ASM

```
; Mode X (320x240, 256 colors) display memory to display memory masked copy
; routine. Works on all VGAs. Uses approach of reading 4 pixels at a time from
; source into latches, then writing latches to destination, using Map Mask
; register to perform masking. Copies up to but not including column at
; SourceEndX and row at SourceEndY. No clipping is performed. Results are not
; guaranteed if source and destination overlap. C near-callable as:
;
;    void CopyScreenToScreenMaskedX(int SourceStartX,
;       int SourceStartY, int SourceEndX, int SourceEndY,
;       int DestStartX, int DestStartY, MaskedImage * Source,
;       unsigned int DestPageBase, int DestBitmapWidth);

SC_INDEX        equ     03c4h       ;Sequence Controller Index register port
MAP_MASK        equ     02h         ;index in SC of Map Mask register
GC_INDEX        equ     03ceh       ;Graphics Controller Index register port
BIT_MASK        equ     08h         ;index in GC of Bit Mask register
SCREEN_SEG      equ     0a000h      ;segment of display memory in mode X

parms   struc
                dw      2 dup (?)   ;pushed BP and return address
SourceStartX    dw      ?           ;X coordinate of upper left corner of source
SourceStartY    dw      ?           ;Y coordinate of upper left corner of source
SourceEndX      dw      ?           ;X coordinate of lower right corner of source
                                    ; (the column at SourceEndX is not copied)
SourceEndY      dw      ?           ;Y coordinate of lower right corner of source
                                    ; (the row at SourceEndY is not copied)
DestStartX      dw      ?           ;X coordinate of upper left corner of dest
DestStartY      dw      ?           ;Y coordinate of upper left corner of dest
Source          dw      ?           ;pointer to MaskedImage struct for source
                                    ; which source resides
DestPageBase    dw      ?           ;base offset in display memory of page in
                                    ; which dest resides
DestBitmapWidth dw      ?           ;# of pixels across dest bitmap (must be multiple of 4)
parms   ends
```

```
SourceNextScanOffset   equu  -2         ;local storage for distance from end of
                                        ; one source scan line to start of next
DestNextScanOffset     equ   -4         ;local storage for distance from end of
                                        ; one dest scan line to start of next
RectAddrWidth          equ   -6         ;local storage for address width of rectangle
RectHeight             equ   -8         ;local storage for height of rectangle
SourceBitmapWidthequ   -10              ;local storage for width of source bitmap
                                        ; (in addresses)
STACK_FRAME_SIZE       equ   10
MaskedImage            struc
 Alignments            dw  4 dup(?)     ;pointers to AlignedMaskedImages for the
                                        ; 4 possible destination image alignments
MaskedImage      ends
AlignedMaskedImage      struc
 ImageWidth      dw     ?               ;image width in addresses (also mask width in bytes)
 ImagePtr        dw     ?               ;offset of image bitmap in display memory
 MaskPtr         dw     ?               ;pointer to mask bitmap in DS
AlignedMaskedImage      ends
        .model  small
        .code
        public  _CopyScreenToScreenMaskedX
_CopyScreenToScreenMaskedX proc    near
        push    bp                      ;preserve caller's stack frame
        mov     bp,sp                   ;point to local stack frame
        sub     sp,STACK_FRAME_SIZE     ;allocate space for local vars
        push    si                      ;preserve caller's register variables
        push    di

        cld
        mov     dx,GC_INDEX             ;set the bit mask to select all bits
        mov     ax,00000h+BIT_MASK      ; from the latches and none from
        out     dx,ax                   ; the CPU, so that we can write the
                                        ; latch contents directly to memory
        mov     ax,SCREEN_SEG           ;point ES to display memory
        mov     es,ax
        mov     ax,[bp+DestBitmapWidth]
        shr     ax,1                    ;convert to width in addresses
        shr     ax,1
        mul     [bp+DestStartY]         ;top dest rect scan line
        mov     di,[bp+DestStartX]
        mov     si,di
        shr     di,1                    ;X/4 = offset of first dest rect pixel in
        shr     di,1                    ; scan line
        add     di,ax                   ;offset of first dest rect pixel in page
        add     di,[bp+DestPageBase]    ;offset of first dest rect pixel in display
                                        ; memory. now look up the image that's
                                        ; aligned to match left-edge alignment
                                        ; of destination
        and     si,3                    ;DestStartX modulo 4
        mov     cx,si                   ;set aside alignment for later
        shl     si,1                    ;prepare for word look-up
        mov     bx,[bp+Source]          ;point to source MaskedImage structure
        mov     bx,[bx+Alignments+si]   ;point to AlignedMaskedImage
                                        ; struc for current left edge alignment
        mov     ax,[bx+ImageWidth]      ;image width in addresses
        mov     [bp+SourceBitmapWidth],ax     ;remember image width in addresses
        mul     [bp+SourceStartY]       ;top source rect scan line
        mov     si,[bp+SourceStartX]
        shr     si,1                    ;X/4 = address of first source rect pixel in
        shr     si,1                    ; scan line
        add     si,ax                   ;offset of first source rect pixel in image
```

```
        mov     ax,si
        add     si,[bx+MaskPtr]         ;point to mask offset of first mask pixel in DS
        mov     bx,[bx+ImagePtr]        ;offset of first source rect pixel
        add     bx,ax                   ; in display memory

        mov     ax,[bp+SourceStartX]    ;calculate # of addresses across
        add     ax,cx                   ; rect, shifting if necessary to
        add     cx,[bp+SourceEndX]      ; account for alignment
        cmp     cx,ax
        jle     CopyDone                ;skip if 0 or negative width
        add     cx,3
        and     ax,not 011b
        sub     cx,ax
        shr     cx,1
        shr     cx,1                    ;# of addresses across rectangle to copy
        mov     ax,[bp+SourceEndY]
        sub     ax,[bp+SourceStartY]    ;AX = height of rectangle
        jle     CopyDone                ;skip if 0 or negative height
        mov     [bp+RectHeight],ax
        mov     ax,[bp+DestBitmapWidth]
        shr     ax,1                    ;convert to width in addresses
        shr     ax,1
        sub     ax,cx           ;distance from end of one dest scan line to start of next
        mov     [bp+DestNextScanOffset],ax
        mov     ax,[bp+SourceBitmapWidth]     ;width in addresses
        sub     ax,cx           ;distance from end of source scan line to start of next
        mov     [bp+SourceNextScanOffset],ax
        mov     [bp+RectAddrWidth],cx   ;remember width in addresses

        mov     dx,SC_INDEX
        mov     al,MAP_MASK
        out     dx,al                   ;point SC Index register to Map Mask
        inc     dx                      ;point to SC Data register
CopyRowsLoop:
        mov     cx,[bp+RectAddrWidth]   ;width across
CopyScanLineLoop:
        lodsb                           ;get the mask for this four-pixel set
                                        ; and advance the mask pointer
        out     dx,al                   ;set the mask
        mov     al,es:[bx]              ;load the latches with four-pixel set from source
        mov     es:[di],al              ;copy the four-pixel set to the dest
        inc     bx                      ;advance the source pointer
        inc     di                      ;advance the destination pointer
        dec     cx                      ;count off four-pixel sets
        jnz     CopyScanLineLoop

        mov     ax,[bp+SourceNextScanOffset]
        add     si,ax                           ;point to the start of
        add     bx,ax                           ; the next source, mask,
        add     di,[bp+DestNextScanOffset]      ; and dest lines
        dec     word ptr [bp+RectHeight]        ;count down scan lines
        jnz     CopyRowsLoop
CopyDone:
        mov     dx,GC_INDEX+1           ;restore the bit mask to its default,
        mov     al,0ffh                 ; which selects all bits from the CPU
        out     dx,al                   ; and none from the latches (the GC
                                        ; Index still points to Bit Mask)
        pop     di                      ;restore caller's register variables
        pop     si
        mov     sp,bp                   ;discard storage for local variables
```

```
        pop     bp                      ;restore caller's stack frame
        ret
_CopyScreenToScreenMaskedX endp
        end
```

It would be handy to have a function that, given a base image and mask, generates the four image and mask alignments and fills in the **MaskedImage** structure. Listing 49.3, together with the include file in Listing 49.4 and the system memory-to-display memory block-copy routine in Listing 48.4 (in the previous chapter) does just that. It would be faster if Listing 49.3 were in assembly language, but there's no reason to think that generating aligned images needs to be particularly fast; in such cases, I prefer to use C, for reasons of coding speed, fewer bugs, and maintainability.

LISTING 49.3 L49-3.C

```
/* Generates all four possible mode X image/mask alignments, stores image
alignments in display memory, allocates memory for and generates mask
alignments, and fills out an AlignedMaskedImage structure. Image and mask must
both be in byte-per-pixel form, and must both be of width ImageWidth. Mask
maps isomorphically (one to one) onto image, with each 0-byte in mask masking
off corresponding image pixel (causing it not to be drawn), and each non-0-byte
allowing corresponding image pixel to be drawn. Returns 0 if failure, or # of
display memory addresses (4-pixel sets) used if success. For simplicity,
allocated memory is not deallocated in case of failure. Compiled with
Borland C++ in C compilation mode. */

#include <stdio.h>
#include <stdlib.h>
#include "maskim.h"

extern void CopySystemToScreenX(int, int, int, int, int, int, char *,
   unsigned int, int, int);
unsigned int CreateAlignedMaskedImage(MaskedImage * ImageToSet,
   unsigned int DispMemStart, char * Image, int ImageWidth,
   int ImageHeight, char * Mask)
{
   int Align, ScanLine, BitNum, Size, TempImageWidth;
   unsigned char MaskTemp;
   unsigned int DispMemOffset = DispMemStart;
   AlignedMaskedImage *WorkingAMImage;
   char *NewMaskPtr, *OldMaskPtr;
   /* Generate each of the four alignments in turn. */
   for (Align = 0; Align < 4; Align++) {
      /* Allocate space for the AlignedMaskedImage struct for this alignment. */
      if ((WorkingAMImage = ImageToSet->Alignments[Align] =
            malloc(sizeof(AlignedMaskedImage))) == NULL)
         return 0;
      WorkingAMImage->ImageWidth =
            (ImageWidth + Align + 3) / 4; /* width in 4-pixel sets */
      WorkingAMImage->ImagePtr = DispMemOffset; /* image dest */
      /* Download this alignment of the image. */
      CopySystemToScreenX(0, 0, ImageWidth, ImageHeight, Align, 0,
            Image, DispMemOffset, ImageWidth, WorkingAMImage->ImageWidth * 4);
      /* Calculate the number of bytes needed to store the mask in
         nibble (Map Mask-ready) form, then allocate that space. */
      Size = WorkingAMImage->ImageWidth * ImageHeight;
      if ((WorkingAMImage->MaskPtr = malloc(Size)) == NULL)
         return 0;
```

```
      /* Generate this nibble oriented (Map Mask-ready) alignment of
         the mask, one scan line at a time. */
      OldMaskPtr = Mask;
      NewMaskPtr = WorkingAMImage->MaskPtr;
      for (ScanLine = 0; ScanLine < ImageHeight; ScanLine++) {
         BitNum = Align;
         MaskTemp = 0;
         TempImageWidth = ImageWidth;
         do {
            /* Set the mask bit for next pixel according to its alignment. */
            MaskTemp |= (*OldMaskPtr++ != 0) << BitNum;
            if (++BitNum > 3) {
               *NewMaskPtr++ = MaskTemp;
               MaskTemp = BitNum = 0;
            }
         } while (--TempImageWidth);
         /* Set any partial final mask on this scan line. */
         if (BitNum != 0) *NewMaskPtr++ = MaskTemp;
      }
      DispMemOffset += Size; /* mark off the space we just used */
   }
   return DispMemOffset - DispMemStart;
}
```

LISTING 49.4 MASKIM.H

```
/* MASKIM.H: structures used for storing and manipulating masked
   images */

/* Describes one alignment of a mask-image pair. */
typedef struct {
   int ImageWidth; /* image width in addresses in display memory (also
                      mask width in bytes) */
   unsigned int ImagePtr; /* offset of image bitmap in display mem */
   char *MaskPtr;  /* pointer to mask bitmap */
} AlignedMaskedImage;

/* Describes all four alignments of a mask-image pair. */
typedef struct {
   AlignedMaskedImage *Alignments[4]; /* ptrs to AlignedMaskedImage
                                         structs for four possible destination
                                         image alignments */
} MaskedImage;
```

Notes on Masked Copying

Listings 49.1 and 49.2, like all Mode X code I've presented, perform no clipping, because clipping code would complicate the listings too much. While clipping can be implemented directly in the low-level Mode X routines (at the beginning of Listing 49.1, for instance), another, potentially simpler approach would be to perform clipping at a higher level, modifying the coordinates and dimensions passed to low-level routines such as Listings 49.1 and 49.2 as necessary to accomplish the desired clipping. It is for precisely this reason that the low-level Mode X routines support programmable start coordinates in the source images, rather than assuming (0,0); likewise for the distinction between the width of the image and the width of the area of the image to draw.

Also, it would be more efficient to make up structures that describe the source and destination bitmaps, with dimensions and coordinates built in, and simply pass pointers to these structures to the low level, rather than passing many separate parameters, as is now the case. I've used separate parameters for simplicity and flexibility.

*Be aware that as nifty as Mode X hardware-assisted masked copying is, whether or not it's actually faster than software-only masked or transparent copying depends upon the processor and the video adapter. The advantage of Mode X masked copying is the 32-bit parallelism; the disadvantages are the need to read display memory and the need to perform an **OUT** for every four pixels. (**OUT** is a slow 486/Pentium instruction, and most VGAs respond to **OUT**s much more slowly than to display memory writes.)*

Animation

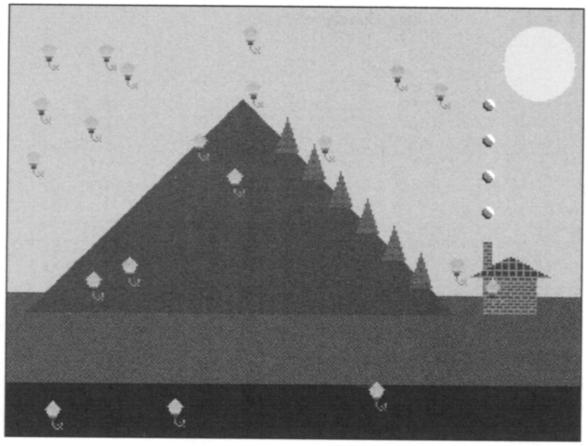
Gosh. There's just no way I can discuss high-level animation fundamentals in any detail here; I could spend an entire (and entirely separate) book on animation techniques alone. You might want to have a look at Chapters 43 through 46 before attacking the code in this chapter; that will have to do us for the present volume. (I will return to *3-D* animation in the next chapter.)

Basically, I'm going to perform page flipped animation, in which one page (that is, a bitmap large enough to hold a full screen) of display memory is displayed while another page is drawn to. When the drawing is finished, the newly modified page is displayed, and the other—now invisible—page is drawn to. The process repeats ad infinitum. For further information, some good places to start are *Computer Graphics*, by Foley and van Dam (Addison-Wesley); *Principles of Interactive Computer Graphics*, by Newman and Sproull (McGraw Hill); and "Real-Time Animation" by Rahner James (January 1990, *Dr. Dobb's Journal*).

Some of the code in this chapter was adapted for Mode X from the code in Chapter 44—yet another reason to read that chapter before finishing this one.

Mode X Animation in Action

Listing 49.5 ties together everything I've discussed about Mode X so far in a compact but surprisingly powerful animation package. Listing 49.5 first uses solid and patterned fills and system-memory-to-screen-memory masked copying to draw a static background containing a mountain, a sun, a plain, water, and a house with puffs of smoke coming out of the chimney, and sets up the four alignments of a masked kite image. The background is transferred to both display pages, and drawing of 20 kite images in the nondisplayed page using fast masked copying begins. After all images have been drawn, the page is flipped to show the newly updated screen, and the kites are moved and drawn in the other page, which is no longer displayed. Kites are erased at their old positions in the nondisplayed page by block copying from the



An animated Mode X screen.
Figure 49.1

background page. (See the discussion in the previous chapter for the display memory organization used by Listing 49.5.) So far as the displayed image is concerned, there is never any hint of flicker or disturbance of the background. This continues at a rate of up to 60 times a second until Esc is pressed to exit the program. See Figure 49.1 for a screen shot of the resulting image—add the animation in your imagination.

LISTING 49.5 L49-5.C

```
/* Sample mode X VGA animation program. Portions of this code first appeared
   in PC Techniques. Compiled with Borland C++ 2.0 in C compilation mode. */

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

#define SCREEN_SEG         0xA000
#define SCREEN_WIDTH       320
#define SCREEN_HEIGHT      240
#define PAGE0_START_OFFSET 0
#define PAGE1_START_OFFSET (((long)SCREEN_HEIGHT*SCREEN_WIDTH)/4)
#define BG_START_OFFSET    (((long)SCREEN_HEIGHT*SCREEN_WIDTH*2)/4)
#define DOWNLOAD_START_OFFSET (((long)SCREEN_HEIGHT*SCREEN_WIDTH*3)/4)

static unsigned int PageStartOffsets[2] = {PAGE0_START_OFFSET,PAGE1_START_OFFSET};
static char GreenAndBrownPattern[] = {2,6,2,6, 6,2,6,2, 2,6,2,6, 6,2,6,2};
static char PineTreePattern[] = {2,2,2,2, 2,6,2,6, 2,2,6,2, 2,2,2,2};
static char BrickPattern[] = {6,6,7,6, 7,7,7,7, 7,6,6,6, 7,7,7,7,};
static char RoofPattern[] = {8,8,8,7, 7,7,7,7, 8,8,8,7, 8,8,8,7};

#define SMOKE_WIDTH  7
#define SMOKE_HEIGHT 7
```

```
static char SmokePixels[] = {
   0, 0,15,15,15, 0, 0,
   0, 7, 7,15,15,15, 0,
   8, 7, 7, 7,15,15,15,
   8, 7, 7, 7, 7,15,15,
   0, 8, 7, 7, 7, 7,15,
   0, 0, 8, 7, 7, 7, 0,
   0, 0, 0, 8, 8, 0, 0};
static char SmokeMask[] = {
   0, 0, 1, 1, 1, 0, 0,
   0, 1, 1, 1, 1, 1, 0,
   1, 1, 1, 1, 1, 1, 1,
   1, 1, 1, 1, 1, 1, 1,
   1, 1, 1, 1, 1, 1, 1,
   0, 1, 1, 1, 1, 1, 0,
   0, 0, 1, 1, 1, 0, 0};
#define KITE_WIDTH  10
#define KITE_HEIGHT 16
static char KitePixels[] = {
   0, 0, 0, 0,45, 0, 0, 0, 0, 0,
   0, 0, 0,46,46,46, 0, 0, 0, 0,
   0, 0,47,47,47,47,47, 0, 0, 0,
   0,48,48,48,48,48,48,48, 0, 0,
  49,49,49,49,49,49,49,49,49, 0,
   0,50,50,50,50,50,50,50, 0, 0,
   0,51,51,51,51,51,51,51, 0, 0,
   0, 0,52,52,52,52,52, 0, 0, 0,
   0, 0,53,53,53,53,53, 0, 0, 0,
   0, 0, 0,54,54,54, 0, 0, 0, 0,
   0, 0, 0,55,55,55, 0, 0, 0, 0,
   0, 0, 0, 0,58, 0, 0, 0, 0, 0,
   0, 0, 0, 0,59, 0, 0, 0, 0,66,
   0, 0, 0, 0,60, 0, 0,64, 0,65,
   0, 0, 0, 0, 0,61, 0, 0,64, 0,
   0, 0, 0, 0, 0, 0,62,63, 0,64};
static char KiteMask[] = {
   0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
   0, 0, 0, 1, 1, 1, 0, 0, 0, 0,
   0, 0, 1, 1, 1, 1, 1, 0, 0, 0,
   0, 1, 1, 1, 1, 1, 1, 1, 0, 0,
   1, 1, 1, 1, 1, 1, 1, 1, 1, 0,
   0, 1, 1, 1, 1, 1, 1, 1, 0, 0,
   0, 1, 1, 1, 1, 1, 1, 1, 0, 0,
   0, 0, 1, 1, 1, 1, 1, 0, 0, 0,
   0, 0, 1, 1, 1, 1, 1, 0, 0, 0,
   0, 0, 0, 1, 1, 1, 0, 0, 0, 0,
   0, 0, 0, 1, 1, 1, 0, 0, 0, 0,
   0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 1, 0, 0, 0, 0, 1,
   0, 0, 0, 0, 1, 0, 0, 1, 0, 1,
   0, 0, 0, 0, 0, 1, 0, 0, 1, 0,
   0, 0, 0, 0, 0, 0, 1, 1, 0, 1};
static MaskedImage KiteImage;

#define NUM_OBJECTS  20
typedef struct {
   int X,Y,Width,Height,XDir,YDir,XOtherPage,YOtherPage;
   MaskedImage *Image;
} AnimatedObject;
```

```
AnimatedObject AnimatedObjects[] = {
   {  0,  0,KITE_WIDTH,KITE_HEIGHT, 1, 1,  0,  0,&KiteImage},
   { 10, 10,KITE_WIDTH,KITE_HEIGHT, 0, 1, 10, 10,&KiteImage},
   { 20, 20,KITE_WIDTH,KITE_HEIGHT,-1, 1, 20, 20,&KiteImage},
   { 30, 30,KITE_WIDTH,KITE_HEIGHT,-1,-1, 30, 30,&KiteImage},
   { 40, 40,KITE_WIDTH,KITE_HEIGHT, 1,-1, 40, 40,&KiteImage},
   { 50, 50,KITE_WIDTH,KITE_HEIGHT, 0,-1, 50, 50,&KiteImage},
   { 60, 60,KITE_WIDTH,KITE_HEIGHT, 1, 0, 60, 60,&KiteImage},
   { 70, 70,KITE_WIDTH,KITE_HEIGHT,-1, 0, 70, 70,&KiteImage},
   { 80, 80,KITE_WIDTH,KITE_HEIGHT, 1, 2, 80, 80,&KiteImage},
   { 90, 90,KITE_WIDTH,KITE_HEIGHT, 0, 2, 90, 90,&KiteImage},
   {100,100,KITE_WIDTH,KITE_HEIGHT,-1, 2,100,100,&KiteImage},
   {110,110,KITE_WIDTH,KITE_HEIGHT,-1,-2,110,110,&KiteImage},
   {120,120,KITE_WIDTH,KITE_HEIGHT, 1,-2,120,120,&KiteImage},
   {130,130,KITE_WIDTH,KITE_HEIGHT, 0,-2,130,130,&KiteImage},
   {140,140,KITE_WIDTH,KITE_HEIGHT, 2, 0,140,140,&KiteImage},
   {150,150,KITE_WIDTH,KITE_HEIGHT,-2, 0,150,150,&KiteImage},
   {160,160,KITE_WIDTH,KITE_HEIGHT, 2, 2,160,160,&KiteImage},
   {170,170,KITE_WIDTH,KITE_HEIGHT,-2, 2,170,170,&KiteImage},
   {180,180,KITE_WIDTH,KITE_HEIGHT,-2,-2,180,180,&KiteImage},
   {190,190,KITE_WIDTH,KITE_HEIGHT, 2,-2,190,190,&KiteImage},
};
void main(void);
void DrawBackground(unsigned int);
void MoveObject(AnimatedObject *);
extern void Set320x240Mode(void);
extern void FillRectangleX(int, int, int, int, unsigned int, int);
extern void FillPatternX(int, int, int, int, unsigned int, char*);
extern void CopySystemToScreenMaskedX(int, int, int, int, int, int,
   char *, unsigned int, int, int, char *);
extern void CopyScreenToScreenX(int, int, int, int, int, int,
   unsigned int, unsigned int, int, int);
extern unsigned int CreateAlignedMaskedImage(MaskedImage *,
   unsigned int, char *, int, int, char *);
extern void CopyScreenToScreenMaskedX(int, int, int, int, int, int,
   MaskedImage *, unsigned int, int);
extern void ShowPage(unsigned int);

void main()
{
   int DisplayedPage, NonDisplayedPage, Done, i;
   union REGS regset;
   Set320x240Mode();
   /* Download the kite image for fast copying later. */
   if (CreateAlignedMaskedImage(&KiteImage, DOWNLOAD_START_OFFSET,
         KitePixels, KITE_WIDTH, KITE_HEIGHT, KiteMask) == 0) {
      regset.x.ax = 0x0003; int86(0x10, &regset, &regset);
      printf("Couldn't get memory\n"); exit();
   }
   /* Draw the background to the background page. */
   DrawBackground(BG_START_OFFSET);
   /* Copy the background to both displayable pages. */
   CopyScreenToScreenX(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT, 0, 0,
         BG_START_OFFSET, PAGE0_START_OFFSET, SCREEN_WIDTH, SCREEN_WIDTH);
   CopyScreenToScreenX(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT, 0, 0,
         BG_START_OFFSET, PAGE1_START_OFFSET, SCREEN_WIDTH, SCREEN_WIDTH);
   /* Move the objects and update their images in the nondisplayed
      page, then flip the page, until Esc is pressed. */
   Done = DisplayedPage = 0;
   do {
      NonDisplayedPage = DisplayedPage ^ 1;
```

```
      /* Erase each object in nondisplayed page by copying block from
           background page at last location in that page. */
      for (i=0; i<NUM_OBJECTS; i++) {
         CopyScreenToScreenX(AnimatedObjects[i].XOtherPage,
               AnimatedObjects[i].YOtherPage,
               AnimatedObjects[i].XOtherPage +
               AnimatedObjects[i].Width,
               AnimatedObjects[i].YOtherPage +
               AnimatedObjects[i].Height,
               AnimatedObjects[i].XOtherPage,
               AnimatedObjects[i].YOtherPage, BG_START_OFFSET,
               PageStartOffsets[NonDisplayedPage], SCREEN_WIDTH, SCREEN_WIDTH);
      }
      /* Move and draw each object in the nondisplayed page. */
      for (i=0; i<NUM_OBJECTS; i++) {
         MoveObject(&AnimatedObjects[i]);
         /* Draw object into nondisplayed page at new location */
         CopyScreenToScreenMaskedX(0, 0, AnimatedObjects[i].Width,
               AnimatedObjects[i].Height, AnimatedObjects[i].X,
               AnimatedObjects[i].Y, AnimatedObjects[i].Image,
               PageStartOffsets[NonDisplayedPage], SCREEN_WIDTH);
      }
      /* Flip to the page into which we just drew. */
      ShowPage(PageStartOffsets[DisplayedPage = NonDisplayedPage]);
      /* See if it's time to end. */
      if (kbhit()) {
         if (getch() == 0x1B) Done = 1;   /* Esc to end */
      }
   } while (!Done);
   /* Restore text mode and done. */
   regset.x.ax = 0x0003; int86(0x10, &regset, &regset);
}
void DrawBackground(unsigned int PageStart)
{
   int i,j,Temp;
   /* Fill the screen with cyan. */
   FillRectangleX(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT, PageStart, 11);
   /* Draw a green and brown rectangle to create a flat plain. */
   FillPatternX(0, 160, SCREEN_WIDTH, SCREEN_HEIGHT, PageStart,
            GreenAndBrownPattern);
   /* Draw blue water at the bottom of the screen. */
   FillRectangleX(0, SCREEN_HEIGHT-30, SCREEN_WIDTH, SCREEN_HEIGHT,
               PageStart, 1);
   /* Draw a brown mountain rising out of the plain. */
   for (i=0; i<120; i++)
      FillRectangleX(SCREEN_WIDTH/2-30-i, 51+i, SCREEN_WIDTH/2-30+i+1,
               51+i+1, PageStart, 6);
   /* Draw a yellow sun by overlapping rects of various shapes. */
   for (i=0; i<=20; i++) {
      Temp = (int)(sqrt(20.0*20.0 - (float)i*(float)i) + 0.5);
      FillRectangleX(SCREEN_WIDTH-25-i, 30-Temp, SCREEN_WIDTH-25+i+1,
               30+Temp+1, PageStart, 14);
   }
   /* Draw green trees down the side of the mountain. */
   for (i=10; i<90; i += 15)
      for (j=0; j<20; j++)
       FillPatternX(SCREEN_WIDTH/2+i-j/3-15, i+j+51,SCREEN_WIDTH/2+i+j/3-15+1,
                              i+j+51+1, PageStart, PineTreePattern);
   /* Draw a house on the plain. */
   FillPatternX(265, 150, 295, 170, PageStart, BrickPattern);
```

```
    FillPatternX(265, 130, 270, 150, PageStart, BrickPattern);
    for (i=0; i<12; i++)
       FillPatternX(280-i*2, 138+i, 280+i*2+1, 138+i+1, PageStart, RoofPattern);
    /* Finally, draw puffs of smoke rising from the chimney. */
    for (i=0; i<4; i++)
       CopySystemToScreenMaskedX(0, 0, SMOKE_WIDTH, SMOKE_HEIGHT, 264,
         110-i*20, SmokePixels, PageStart, SMOKE_WIDTH,SCREEN_WIDTH, SmokeMask);
}
/* Move the specified object, bouncing at the edges of the screen and
   remembering where the object was before the move for erasing next time. */
void MoveObject(AnimatedObject * ObjectToMove) {
   int X, Y;
   X = ObjectToMove->X + ObjectToMove->XDir;
   Y = ObjectToMove->Y + ObjectToMove->YDir;
   if ((X < 0) || (X > (SCREEN_WIDTH - ObjectToMove->Width))) {
      ObjectToMove->XDir = -ObjectToMove->XDir;
      X = ObjectToMove->X + ObjectToMove->XDir;
   }
   if ((Y < 0) || (Y > (SCREEN_HEIGHT - ObjectToMove->Height))) {
      ObjectToMove->YDir = -ObjectToMove->YDir;
      Y = ObjectToMove->Y + ObjectToMove->YDir;
   }
   /* Remember previous location for erasing purposes. */
   ObjectToMove->XOtherPage = ObjectToMove->X;
   ObjectToMove->YOtherPage = ObjectToMove->Y;
   ObjectToMove->X = X; /* set new location */
   ObjectToMove->Y = Y;
}
```

Here's something worth noting: The animation is extremely smooth on a 20 MHz 386. It is somewhat more jerky on an 8 MHz 286, because only 30 frames a second can be processed. If animation looks jerky on your PC, try reducing the number of kites.

The kites draw perfectly into the background, with no interference or fringe, thanks to masked copying. In fact, the kites also cross with no interference (the last-drawn kite is always in front), although that's not readily apparent because they all look the same anyway and are moving fast. Listing 49.5 isn't inherently limited to kites; create your own images and initialize the object list to display a mix of those images and see the full power of Mode X animation.

The external functions called by Listing 49.5 can be found in Listings 49.1, 49.2, 49.3, and 49.6, and in the listings for the previous two chapters.

LISTING 49.6 L49-6.ASM

```
; 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);
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_LOWequ  0dh               ;bitmap start address low byte

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

```
        .model  small
        .code
        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
        end
```

Works Fast, Looks Great

We now end our exploration of Mode X, although we'll use it again shortly for 3-D animation. Mode X admittedly has its complexities; that's why I've provided a broad and flexible primitive set. Still, so what if it *is* complex? Take a look at Listing 49.5 in action. That sort of colorful, high-performance animation is worth jumping through a few hoops for; drawing 20, or even 10, fair-sized objects at a rate of 60 Hz, with no flicker, interference, or fringe, is no mean accomplishment, even on a 386.

There's much more we could do with animation in general and with Mode X in particular, but it's time to move on to new challenges. In closing, I'd like to point out that all of the VGA's hardware features, including the built-in AND, OR, and XOR functions, are available in Mode X, just as they are in the standard VGA modes. If you understand the VGA's hardware in mode 12H, try applying that knowledge to Mode X; you might be surprised at what you find you can do.