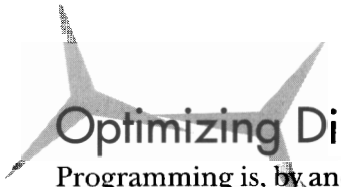
Chapter 46

Who Was that Masked Image?

Chapter 46



Optimizing Dirty-Rectangle Animation

Programming is, by and large, a linear process. One statement or instruction follows another, in predictable sequences, with tiny building blocks strung together to make a custom state machine. As programmers, we grow adept at this sort of idealized linear thinking, which is, of course, A Good Thing. Still, it's important to keep in mind that there's a large chunk of the human mind that doesn't work in a linear fashion.

I've written elsewhere about the virtues of nonlinear/right-brain/lateral/what-have-you thinking in solving tough programming problems, such as debugging or optimization, but it bears repeating. The mind can be an awesome pattern-matching and extrapolation tool, if you let it. For example, the other day I was grinding my way through a particularly difficult bit of debugging. The code had been written by someone else, and, to my mind, there's nothing worse than debugging someone else's code; there's always the nasty feeling that you don't quite know what's going on. The overall operation of this code wouldn't come clear in my head, no matter how long I stared at it, leaving me with a rising sense of frustration and a determination not to quit until I got this bug.

In the midst of this, a coworker poked his head through the door and told me he had something I had to listen to. Reluctantly, I went to his office, whereupon he played a tape of what is surely one of the most bizarre 911 calls in history. No doubt some of you have heard this tape, which I will briefly describe as involving a deer destroying the interior of a car and biting a man in the neck. Perhaps you found it

funny, perhaps not—but as for me, it hit me exactly right. I started laughing helplessly, tears rolling down my face. When I went back to work—presto!—the pieces of the debugging puzzle had come together in my head, and the work went quickly and easily.

Obviously, my mind needed a break from linear, left-brain, push-it-out thinking, so it could do the sort of integrating work it does so well—but that it's rarely willing to do under conscious control. It was exactly this sort of thinking I had in mind when I titled my 1989 optimization book *Zen of Assembly Language.* (Although I must admit that few people seem to have gotten the connection, and I've had to field a lot of questions about whether I'm a Zen disciple. I'm not—actually, I'm more of a Dave Barry disciple. If you don't know who Dave Barry is, you should; he's good for your right brain.) Give your mind a break once in a while, and I'll bet you'll find you're more productive.

We're strange thinking machines, but we're the best ones yet invented, and it's worth learning how to tap our full potential. And with that, it's back to dirty-rectangle animation.

Dirty-Rectangle Animation, Continued

In the last chapter, I introduced the idea of dirty-rectangle animation. This technique is an alternative to page flipping that's capable of producing animation of very high visual quality, without any help at all from video hardware, and without the need for any extra, nondisplayed video memory. This makes dirty-rectangle animation more widely usable than page flipping, because many adapters don't support page flipping. Dirty-rectangle animation also tends to be simpler to implement than page flipping, because there's only one bitmap to keep track of. A final advantage of dirty-rectangle animation is that it's potentially somewhat faster than page flipping, because display-memory accesses can theoretically be reduced to exactly one access for each pixel that changes from one frame to the next.

The speed advantage of dirty-rectangle animation was entirely theoretical in the previous chapter, because the implementation was completely in C, and because no attempt was made to minimize display memory accesses. The visual quality of Chapter 45's animation was also less than ideal, for reasons we'll explore shortly. The code in Listings 46.1 and 46.2 addresses the shortcomings of Chapter 45's code.

Listing 46.2 implements the low-level drawing routines in assembly language, which boosts performance a good deal. For maximum performance, it would be worthwhile to convert more of Listing 46.1 into assembly, so a call isn't required for each animated image, and overall performance could be improved by streamlining the C code, but Listing 46.2 goes a long way toward boosting animation speed. This program now supports snappy animation of 15 images (as opposed to 10 for the software presented in the last chapter), and the images are now two pixels wider. That level of performance is all the more impressive considering that for this chapter I've converted the code from using rectangular images to using masked images.

LISTING 46.1 L46-1.C

```
/* Sample simple dirty-rectangle animation program, partially optimized and
   featuring internal animation, masked images (sprites), and nonoverlapping dirty
   rectangle copying. Tested with Borland C++ in the small model. */

#include <stdlib.h>
#include <conio.h>
#include <alloc.h>
#include <memory.h>
#include <dos.h>

/* Comment out to disable overlap elimination in the dirty rectangle list. */
#define CHECK_OVERLAP 1
#define SCREEN_WIDTH  320
#define SCREEN_HEIGHT 200
#define SCREEN_SEGMENT 0xA000

/* Describes a dirty rectangle */
typedef struct {
   void *Next;    /* pointer to next node in linked dirty rect list */
   int Top;
   int Left;
   int Right;
   int Bottom;
} DirtyRectangle;
/* Describes an animated object */
typedef struct {
   int X;            /* upper left corner in virtual bitmap */
   int Y;
   int XDirection;   /* direction and distance of movement */
   int YDirection;
   int InternalAnimateCount; /* tracking internal animation state */
   int InternalAnimateMax;   /* maximum internal animation state */
} Entity;
/* storage used for dirty rectangles */
#define MAX_DIRTY_RECTANGLES  100
int NumDirtyRectangles;
DirtyRectangle DirtyRectangles[MAX_DIRTY_RECTANGLES];
/* head/tail of dirty rectangle list */
DirtyRectangle DirtyHead;
/* If set to 1, ignore dirty rectangle list and copy the whole screen. */
int DrawWholeScreen = 0;
/* pixels and masks for the two internally animated versions of the image
   we'll animate */
#define IMAGE_WIDTH  13
#define IMAGE_HEIGHT 11
char ImagePixels0[] = {
   0, 0, 0, 9, 9, 9, 9, 9, 0, 0, 0, 0, 0,
   0, 0, 9, 9, 9, 9, 9, 9, 9, 0, 0, 0, 0,
   0, 9, 9, 0, 0,14,14,14, 9, 9, 0, 0, 0,
   9, 9, 0, 0, 0, 0,14,14,14, 9, 9, 0, 0,
   9, 9, 0, 0, 0, 0,14,14,14, 9, 9, 0, 0,
   9, 9,14, 0, 0,14,14,14,14, 9, 9, 0, 0,
   9, 9,14,14,14,14,14,14,14, 9, 9, 0, 0,
   9, 9,14,14,14,14,14,14,14, 9, 9, 0, 0,
   0, 9, 9,14,14,14,14,14, 9, 9, 0, 0, 0,
   0, 0, 9, 9, 9, 9, 9, 9, 9, 0, 0, 0, 0,
   0, 0, 0, 9, 9, 9, 9, 9, 0, 0, 0, 0, 0,
};
```

```
char ImageMask0[] = {
   0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0,
   0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
   0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0,
   1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0,
   1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0,
   1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0,
   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0,
   1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0,
   0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
   0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
   0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0,
};
char ImagePixels1[] = {
   0, 0, 0, 9, 9, 9, 9, 9, 0, 0, 0, 0, 9,
   0, 0, 9, 9, 9, 9, 9, 9, 9, 0, 9, 9, 9,
   0, 9, 9, 0, 0,14,14,14, 9, 9, 9, 9, 0,
   9, 9, 0, 0, 0, 0,14,14,14, 0, 0, 0, 0,
   9, 9, 0, 0, 0, 0,14,14, 0, 0, 0, 0, 0,
   9, 9,14, 0, 0,14,14,14, 0, 0, 0, 0, 0,
   9, 9,14,14,14,14,14,14, 0, 0, 0, 0, 0,
   9, 9,14,14,14,14,14,14,14, 0, 0, 0, 0,
   0, 9, 9,14,14,14,14,14, 9, 9, 9, 9, 0,
   0, 0, 9, 9, 9, 9, 9, 9, 9, 0, 9, 9, 9,
   0, 0, 0, 9, 9, 9, 9, 9, 0, 0, 0, 9, 9,
};
char ImageMask1[] = {
   0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1,
   0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1,
   0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0,
   1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0,
   1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0,
   1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0,
   1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0,
   1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
   0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0,
   0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1,
   0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1,
};
/* Pointers to pixel and mask data for various internally animated
   versions of our animated image. */
char * ImagePixelArray[] = {ImagePixels0, ImagePixels1};
char * ImageMaskArray[] = {ImageMask0, ImageMask1};
/* Animated entities */
#define NUM_ENTITIES 15
Entity Entities[NUM_ENTITIES];
/* pointer to system buffer into which we'll draw */
char far *SystemBufferPtr;
/* pointer to screen */
char far *ScreenPtr;
void EraseEntities(void);
void CopyDirtyRectanglesToScreen(void);
void DrawEntities(void);
void AddDirtyRect(Entity *, int, int);
void DrawMasked(char far *, char *, char *, int, int, int);
void FillRect(char far *, int, int, int, int);
void CopyRect(char far *, char far *, int, int, int, int);
```

```
void main()
{
   int i, XTemp, YTemp;
   unsigned int TempCount;
   char far *TempPtr;
   union REGS regs;
   /* Allocate memory for the system buffer into which we'll draw */
   if (!(SystemBufferPtr = farmalloc((unsigned int)SCREEN_WIDTH*
         SCREEN_HEIGHT))) {
      printf("Couldn't get memory\n");
      exit(1);
   }
   /* Clear the system buffer */
   TempPtr = SystemBufferPtr;
   for (TempCount = ((unsigned)SCREEN_WIDTH*SCREEN_HEIGHT); TempCount--; ) {
      *TempPtr++ = 0;
   }
   /* Point to the screen */
   ScreenPtr = MK_FP(SCREEN_SEGMENT, 0);
   /* Set up the entities we'll animate, at random locations */
   randomize();
   for (i = 0; i < NUM_ENTITIES; i++) {
      Entities[i].X = random(SCREEN_WIDTH - IMAGE_WIDTH);
      Entities[i].Y = random(SCREEN_HEIGHT - IMAGE_HEIGHT);
      Entities[i].XDirection = 1;
      Entities[i].YDirection = -1;
      Entities[i].InternalAnimateCount = i & 1;
      Entities[i].InternalAnimateMax = 2;
   }
   /* Set the dirty rectangle list to empty, and set up the head/tail node
      as a sentinel */
   NumDirtyRectangles = 0;
   DirtyHead.Next = &DirtyHead;
   DirtyHead.Top = 0x7FFF;
   DirtyHead.Left= 0x7FFF;
   DirtyHead.Bottom = 0x7FFF;
   DirtyHead.Right = 0x7FFF;
   /* Set 320x200 256-color graphics mode */
   regs.x.ax = 0x0013;
   int86(0x10, &regs, &regs);
   /* Loop and draw until a key is pressed */
   do {
      /* Draw the entities to the system buffer at their current locations,
         updating the dirty rectangle list */
      DrawEntities();
      /* Draw the dirty rectangles, or the whole system buffer if
         appropriate */
      CopyDirtyRectanglesToScreen();
      /* Reset the dirty rectangle list to empty */
      NumDirtyRectangles = 0;
      DirtyHead.Next = &DirtyHead;
      /* Erase the entities in the system buffer at their old locations,
         updating the dirty rectangle list */
      EraseEntities();
      /* Move the entities, bouncing off the edges of the screen */
      for (i = 0; i < NUM_ENTITIES; i++) {
         XTemp = Entities[i].X + Entities[i].XDirection;
         YTemp = Entities[i].Y + Entities[i].YDirection;
```

```
            if ((XTemp < 0) || ((XTemp + IMAGE_WIDTH) > SCREEN_WIDTH)) {
               Entities[i].XDirection = -Entities[i].XDirection;
               XTemp = Entities[i].X + Entities[i].XDirection;
            }
            if ((YTemp < 0) || ((YTemp + IMAGE_HEIGHT) > SCREEN_HEIGHT)) {
               Entities[i].YDirection = -Entities[i].YDirection;
               YTemp = Entities[i].Y + Entities[i].YDirection;
            }
            Entities[i].X = XTemp;
            Entities[i].Y = YTemp;
         }
      } while (!kbhit());
      getch();    /* clear the keypress */

      /* Return back to text mode */
      regs.x.ax = 0x0003;
      int86(0x10, &regs, &regs);
}
/* Draw entities at their current locations, updating dirty rectangle list. */
void DrawEntities()
{
      int i;
      char far *RowPtrBuffer;
      char *TempPtrImage;
      char *TempPtrMask;
      Entity *EntityPtr;

      for (i = 0, EntityPtr = Entities; i < NUM_ENTITIES; i++, EntityPtr++) {
         /* Remember the dirty rectangle info for this entity */
         AddDirtyRect(EntityPtr, IMAGE_HEIGHT, IMAGE_WIDTH);
         /* Point to the destination in the system buffer */
         RowPtrBuffer = SystemBufferPtr + (EntityPtr->Y * SCREEN_WIDTH) +
               EntityPtr->X;
         /* Advance the image animation pointer */
         if (++EntityPtr->InternalAnimateCount >=
               EntityPtr->InternalAnimateMax) {
            EntityPtr->InternalAnimateCount = 0;
         }
         /* Point to the image and mask to draw */
         TempPtrImage = ImagePixelArray[EntityPtr->InternalAnimateCount];
         TempPtrMask = ImageMaskArray[EntityPtr->InternalAnimateCount];
         DrawMasked(RowPtrBuffer, TempPtrImage, TempPtrMask, IMAGE_HEIGHT,
                  IMAGE_WIDTH, SCREEN_WIDTH);
      }
}
/* Copy the dirty rectangles, or the whole system buffer if appropriate,
   to the screen. */
void CopyDirtyRectanglesToScreen()
{
      int i, RectWidth, RectHeight;
      unsigned int Offset;
      DirtyRectangle * DirtyPtr;
      if (DrawWholeScreen) {
         /* Just copy the whole buffer to the screen */
         DrawWholeScreen = 0;
         CopyRect(ScreenPtr, SystemBufferPtr, SCREEN_HEIGHT, SCREEN_WIDTH,
                  SCREEN_WIDTH, SCREEN_WIDTH);
      } else {
         /* Copy only the dirty rectangles, in the YX-sorted order in which
            they're linked */
```

```
      DirtyPtr = DirtyHead.Next;
      for (i = 0; i < NumDirtyRectangles; i++) {
         /* Offset in both system buffer and screen of image */
         Offset = (unsigned int) (DirtyPtr->Top * SCREEN_WIDTH) +
               DirtyPtr->Left;
         /* Dimensions of dirty rectangle */
         RectWidth = DirtyPtr->Right - DirtyPtr->Left;
         RectHeight = DirtyPtr->Bottom - DirtyPtr->Top;
         /* Copy a dirty rectangle */
         CopyRect(ScreenPtr + Offset, SystemBufferPtr + Offset,
               RectHeight, RectWidth, SCREEN_WIDTH, SCREEN_WIDTH);
         /* Point to the next dirty rectangle */
         DirtyPtr = DirtyPtr->Next;
      }
   }
}
/* Erase the entities in the system buffer at their current locations,
   updating the dirty rectangle list. */
void EraseEntities()
{
   int i;
   char far *RowPtr;

   for (i = 0; i < NUM_ENTITIES; i++) {
      /* Remember the dirty rectangle info for this entity */
      AddDirtyRect(&Entities[i], IMAGE_HEIGHT, IMAGE_WIDTH);
      /* Point to the destination in the system buffer */
      RowPtr = SystemBufferPtr + (Entities[i].Y * SCREEN_WIDTH) +
            Entities[i].X;
      /* Clear the rectangle */
      FillRect(RowPtr, IMAGE_HEIGHT, IMAGE_WIDTH, SCREEN_WIDTH, 0);
   }
}
/* Add a dirty rectangle to the list. The list is maintained in top-to-bottom,
   left-to-right (YX sorted) order, with no pixel ever included twice, to minimize
   the number of display memory accesses and to avoid screen artifacts resulting
   from a large time interval between erasure and redraw for a given object or for
   adjacent objects. The technique used is to check for overlap between the
   rectangle and all rectangles already in the list. If no overlap is found, the
   rectangle is added to the list. If overlap is found, the rectangle is broken
   into nonoverlapping pieces, and the pieces are added to the list by recursive
   calls to this function. */
   void AddDirtyRect(Entity * pEntity, int ImageHeight, int ImageWidth)
{
   DirtyRectangle * DirtyPtr;
   DirtyRectangle * TempPtr;
   Entity TempEntity;
   int i;
   if (NumDirtyRectangles >= MAX_DIRTY_RECTANGLES) {
      /* Too many dirty rectangles; just redraw the whole screen */
      DrawWholeScreen = 1;
      return;
   }
   /* Remember this dirty rectangle. Break up if necessary to avoid
      overlap with rectangles already in the list, then add whatever
      rectangles are left, in YX sorted order */
#ifdef CHECK_OVERLAP
   /* Check for overlap with existing rectangles */
   TempPtr = DirtyHead.Next;
   for (i = 0; i < NumDirtyRectangles; i++, TempPtr = TempPtr->Next) {
```

```
      if ((TempPtr->Left < (pEntity->X + ImageWidth)) &&
          (TempPtr->Right > pEntity->X) &&
          (TempPtr->Top < (pEntity->Y + ImageHeight)) &&
          (TempPtr->Bottom > pEntity->Y)) {

         /* We've found an overlapping rectangle. Calculate the
            rectangles, if any, remaining after subtracting out the
            overlapped areas, and add them to the dirty list */
         /* Check for a nonoverlapped left portion */
         if (TempPtr->Left > pEntity->X) {
            /* There's definitely a nonoverlapped portion at the left; add
               it, but only to at most the top and bottom of the overlapping
               rect; top and bottom strips are taken care of below */
            TempEntity.X = pEntity->X;
            TempEntity.Y = max(pEntity->Y, TempPtr->Top);
            AddDirtyRect(&TempEntity,
                  min(pEntity->Y + ImageHeight, TempPtr->Bottom) -
                  TempEntity.Y,
                        TempPtr->Left - pEntity->X);
         }
         /* Check for a nonoverlapped right portion */
         if (TempPtr->Right < (pEntity->X + ImageWidth)) {
            /* There's definitely a nonoverlapped portion at the right; add
               it, but only to at most the top and bottom of the overlapping
               rect; top and bottom strips are taken care of below */
            TempEntity.X = TempPtr->Right;
            TempEntity.Y = max(pEntity->Y, TempPtr->Top);
            AddDirtyRect(&TempEntity,
                  min(pEntity->Y + ImageHeight, TempPtr->Bottom) -
                  TempEntity.Y,
                  (pEntity->X + ImageWidth) - TempPtr->Right);
         }
         /* Check for a nonoverlapped top portion */
         if (TempPtr->Top > pEntity->Y) {
            /* There's a top portion that's not overlapped */
            TempEntity.X = pEntity->X;
            TempEntity.Y = pEntity->Y;
            AddDirtyRect(&TempEntity, TempPtr->Top - pEntity->Y, ImageWidth);
         }
         /* Check for a nonoverlapped bottom portion */
         if (TempPtr->Bottom < (pEntity->Y + ImageHeight)) {
            /* There's a bottom portion that's not overlapped */
            TempEntity.X = pEntity->X;
            TempEntity.Y = TempPtr->Bottom;
            AddDirtyRect(&TempEntity,
                  (pEntity->Y + ImageHeight) - TempPtr->Bottom, ImageWidth);
         }
         /* We've added all non-overlapped portions to the dirty list */
         return;
      }
   }
#endif /* CHECK_OVERLAP */
   /* There's no overlap with any existing rectangle, so we can just
      add this rectangle as-is */
   /* Find the YX-sorted insertion point. Searches will always terminate,
      because the head/tail rectangle is set to the maximum values */
   TempPtr = &DirtyHead;
   while (((DirtyRectangle *)TempPtr->Next)->Top < pEntity->Y) {
      TempPtr = TempPtr->Next;
   }
```

```
   while ((((DirtyRectangle *)TempPtr->Next)->Top == pEntity->Y) &&
          (((DirtyRectangle *)TempPtr->Next)->Left < pEntity->X)) {
      TempPtr = TempPtr->Next;
   }
   /* Set the rectangle and actually add it to the dirty list */
   DirtyPtr = &DirtyRectangles[NumDirtyRectangles++];
   DirtyPtr->Left = pEntity->X;
   DirtyPtr->Top = pEntity->Y;
   DirtyPtr->Right = pEntity->X + ImageWidth;
   DirtyPtr->Bottom = pEntity->Y + ImageHeight;
   DirtyPtr->Next = TempPtr->Next;
   TempPtr->Next = DirtyPtr;
}
```

LISTING 46.2 L46-2.ASM

```
; Assembly language helper routines for dirty rectangle animation. Tested with
; TASM.
; Fills a rectangle in the specified buffer.
; C-callable as:
;  void FillRect(char far * BufferPtr, int RectHeight, int RectWidth,
;                   int BufferWidth, int Color);
;
        .model  small
        .code
parms   struc
                dw      ?       ;pushed BP
                dw      ?       ;pushed return address
BufferPtr       dd      ?       ;far pointer to buffer in which to fill
RectHeight      dw      ?       ;height of rectangle to fill
RectWidth       dw      ?       ;width of rectangle to fill
BufferWidth     dw      ?       ;width of buffer in which to fill
Color           dw      ?       ;color with which to fill
parms   ends
        public  _FillRect
_FillRect        proc    near
        cld
        push    bp
        mov     bp,sp
        push    di

        les     di,[bp+BufferPtr]
        mov     dx,[bp+RectHeight]
        mov     bx,[bp+BufferWidth]
        sub     bx,[bp+RectWidth]       ;distance from end of one dest scan
                                        ; to start of next
        mov     al,byte ptr [bp+Color]
        mov     ah,al                   ;double the color for REP STOSW
RowLoop:
        mov     cx,[bp+RectWidth]
        shr     cx,1
        rep     stosw
        adc     cx,cx
        rep     stosb
        add     di,bx                   ;point to next scan to fill
        dec     dx                      ;count down rows to fill
        jnz     RowLoop

        pop     di
        pop     bp
        ret
_FillRect   endp
```

```
; Draws a masked image (a sprite) to the specified buffer. C-callable as:
;     void DrawMasked(char far * BufferPtr, char * Pixels, char * Mask,
;                     int ImageHeight, int ImageWidth, int BufferWidth);
parms2  struc
                dw      ?       ;pushed BP
                dw      ?       ;pushed return address
BufferPtr2      dd      ?       ;far pointer to buffer in which to draw
Pixels          dw      ?       ;pointer to image pixels
Mask            dw      ?       ;pointer to image mask
ImageHeight     dw      ?       ;height of image to draw
ImageWidth      dw      ?       ;width of image to draw
BufferWidth2    dw      ?       ;width of buffer in which to draw
parms2  ends
        public  _DrawMasked
_DrawMasked     proc    near
        cld
        push    bp
        mov     bp,sp
        push    si
        push    di

        les     di,[bp+BufferPtr2]
        mov     si,[bp+Mask]
        mov     bx,[bp+Pixels]
        mov     dx,[bp+ImageHeight]
        mov     ax,[bp+BufferWidth2]
        sub     ax,[bp+ImageWidth]      ;distance from end of one dest scan
        mov     [bp+BufferWidth2],ax    ; to start of next
RowLoop2:
        mov     cx,[bp+ImageWidth]
ColumnLoop:
        lodsb                           ;get the next mask byte
        and     al,al                   ;draw this pixel?
        jz      SkipPixel               ;no
        mov     al,[bx]                 ;yes, draw the pixel
        mov     es:[di],al
SkipPixel:
        inc     bx                      ;point to next source pixel
        inc     di                      ;point to next dest pixel
        dec     cx
        jnz     ColumnLoop
        add     di,[bp+BufferWidth2]    ;point to next scan to fill
        dec     dx                      ;count down rows to fill
        jnz     RowLoop2

        pop     di
        pop     si
        pop     bp
        ret
_DrawMasked     endp

; Copies a rectangle from one buffer to another. C-callable as:
;     void CopyRect(DestBufferPtr, SrcBufferPtr, CopyHeight, CopyWidth,
;                   DestBufferWidth, SrcBufferWidth);

parms3  struc
                dw      ?       ;pushed BP
                dw      ?       ;pushed return address
DestBufferPtr   dd      ?       ;far pointer to buffer to which to copy
SrcBufferPtr    dd      ?       ;far pointer to buffer from which to copy
```

```
CopyHeight      dw      ?       ;height of rect to copy
CopyWidth       dw      ?       ;width of rect to copy
DestBufferWidth dw      ?       ;width of buffer to which to copy
SrcBufferWidth  dw      ?       ;width of buffer from which to copy
parms3  ends
        public  _CopyRect
_CopyRect       proc    near
        cld
        push    bp
        mov     bp,sp
        push    si
        push    di
        push    ds

        les     di,[bp+DestBufferPtr]
        lds     si,[bp+SrcBufferPtr]
        mov     dx,[bp+CopyHeight]
        mov     bx,[bp+DestBufferWidth] ;distance from end of one dest scan
        sub     bx,[bp+CopyWidth]       ; of copy to the next
        mov     ax,[bp+SrcBufferWidth]  ;distance from end of one source scan
        sub     ax,[bp+CopyWidth]       ; of copy to the next
RowLoop3:
        mov     cx,[bp+CopyWidth]       ;# of bytes to copy
        shr     cx,1
        rep     movsw                   ;copy as many words as possible
        adc     cx,cx
        rep     movsb                   ;copy odd byte, if any
        add     si,ax                   ;point to next source scan line
        add     di,bx                   ;point to next dest scan line
        dec     dx                      ;count down rows to fill
        jnz     RowLoop3

        pop     ds
        pop     di
        pop     si
        pop     bp
        ret
_CopyRect       endp
        end
```

Masked Images

Masked images are rendered by drawing an object's pixels through a mask; pixels are actually drawn only where the mask specifies that drawing is allowed. This makes it possible to draw nonrectangular objects that don't improperly interfere with one another when they overlap. Masked images also make it possible to have transparent areas (windows) within objects. Masked images produce far more realistic animation than do rectangular images, and therefore are more desirable. Unfortunately, masked images are also considerably slower to draw—however, a good assembly language implementation can go a long way toward making masked images draw rapidly enough, as illustrated by this chapter's code. (Masked images are also known as *sprites*; some video hardware supports sprites directly, but on the PC it's necessary to handle sprites in software.)

Masked images make it possible to render scenes so that a given image convincingly appears to be in front of or behind other images; that is, so images are displayed in *z-order* (by distance). By consistently drawing images that are supposed to be farther away before drawing nearer images, the nearer images will appear in front of the other images, and because masked images draw only precisely the correct pixels (as opposed to blank pixels in the bounding rectangle), there's no interference between overlapping images to destroy the illusion.

In this chapter, I've used the approach of having separate, paired masks and images. Another, quite different approach to masking is to specify a transparent color for copying, and copy only those pixels that are not the transparent color. This has the advantage of not requiring separate mask data, so it's more compact, and the code to implement this is a little less complex than the full masking I've implemented. On the other hand, the transparent color approach is less flexible because it makes one color undrawable. Also, with a transparent color, it's not possible to keep the same base image but use different masks, because the mask information is embedded in the image data.

Internal Animation

I've added another feature essential to producing convincing animation: *internal animation*, which is the process of changing the appearance of a given object over time, as distinguished from changing only the *location* of a given object. Internal animation makes images look active and alive. I've implemented the simplest possible form of internal animation in Listing 46.1—alternation between two images—but even this level of internal animation greatly improves the feel of the overall animation. You could easily increase the number of images cycled through, simply by increasing the value of **InternalAnimateMax** for a given entity. You could also implement more complex image-selection logic to produce more interesting and less predictable internal-animation effects, such as jumping, ducking, running, and the like.

Dirty-Rectangle Management

As mentioned above, dirty-rectangle animation makes it possible to access display memory a minimum number of times. The previous chapter's code didn't do any of that; instead, it copied all portions of every dirty rectangle to the screen, regardless of overlap between rectangles. The code I've presented in this chapter goes to the other extreme, taking great pains never to draw overlapped portions of rectangles more than once. This is accomplished by checking for overlap whenever a rectangle is to be added to the dirty list. When overlap with an existing rectangle is detected, the new rectangle is reduced to between zero and four nonoverlapping rectangles. Those rectangles are then again considered for addition to the dirty list, and may again be reduced, if additional overlap is detected.

A good deal of code is required to generate a fully nonoverlapped dirty list. Is it worth it? It certainly can be, but in the case of Listing 46.1, probably not. For one thing, you'd need larger, heavily overlapped objects for this approach to pay off big. Besides, this program is mostly in C, and spends a lot of time doing things other than actually accessing display memory. It also takes a fair amount of time just to generate the nonoverlapped list; the overhead of all the looping, intersecting, and calling required to generate the list eats up a lot of the benefits of accessing display memory less often. Nonetheless, fully nonoverlapped drawing can be useful under the right circumstances, and I've implemented it in Listing 46.1 so you'll have something to refer to should you decide to go this route.

There are a couple of additional techniques you might try if you want to wring maximum performance out of dirty-rectangle animation. You could try coalescing rectangles as you generate the dirty-rectangle list. That is, you could detect pairs of rectangles that can be joined together into larger rectangles, so that fewer, larger rectangles would have to be copied. This would boost the efficiency of the low-level copying code, albeit at the cost of some cycles in the dirty-list management code.

You might also try taking advantage of the natural coherence of animated graphics screens. In particular, because the rectangle used to erase an image at its old location often overlaps the rectangle within which the image resides at its new location, you could just directly generate the two or three nonoverlapped rectangles required to copy both the erase rectangle and the new-image rectangle for any single moving image. The calculation of these rectangles could be very efficient, given that you know in advance the direction of motion of your images. Handling this particular overlap case would eliminate most overlapped drawing, at a minimal cost. You might then decide to ignore overlapped drawing between different images, which tends to be both less common and more expensive to identify and handle.

Drawing Order and Visual Quality

A final note on dirty-rectangle animation concerns the quality of the displayed screen image. In the last chapter, we simply stuffed dirty rectangles into a list in the order they became dirty, and then copied all of the rectangles in that same order. Unfortunately, this caused all of the erase rectangles to be copied first, followed by all of the rectangles of the images at their new locations. Consequently, there was a significant delay between the appearance of the erase rectangle for a given image and the appearance of the new rectangle. A byproduct was the fact that a partially complete—part old, part new—image was visible long enough to be noticed. In short, although the pixels ended up correct, they were in an intermediate, incorrect state for a sufficient period of time to make the animation look wrong.

This violated a fundamental rule of animation: *No pixel should ever be displayed in a perceptibly incorrect state.* To correct the problem, I've sorted the dirty rectangles first

by Y coordinate, and secondly by X coordinate. This means the screen updates from the top down, and from left to right, so the several nonoverlapping rectangles copied to draw a given image should be drawn nearly simultaneously. Run the code from the last chapter and then this chapter; you'll see quite a difference in appearance.

Avoid the trap of thinking animation is merely a matter of drawing the right pixels, one after another. Animation is the art of drawing *the right pixels at the right times* so that the eye and brain see what you want them to see. Animation is a lot more challenging than merely cranking out pixels, and it sure as heck isn't a purely linear process.