Chapter 37

Dead Cats and Lightning Lines

Chapter 37

Optimizing Run-Length Slice Line Drawing in a Major Way

As I write this, the wife, the kid, and I are in the throes of yet another lightning-quick transcontinental move, this time to Redmond, Washington, to work for You Know Who. Moving is never fun, but what makes it worse for us is the pets. Getting them into kennels and to the airport is hard; there's always the possibility that they might not be allowed to fly because of the weather; and, worst of all, they might not make it. Animals don't usually end up injured or dead, but it does happen.

In a (not notably successful) effort to cheer me up about the prospect of shipping my animals, a friend told me the following story, which he swears actually happened to a friend of his. I don't know—to me, it has the ring of an urban legend, which is to say it makes a good story, but you can never track down the person it really happened to; it's always a friend of a friend. But maybe it is true, and anyway, it's a good story.

This friend of a friend (henceforth referred to as FOF), worked in an air-freight terminal. Consequently, he handled a lot of animals, which was fine by him, because he liked animals; in fact, he had quite a few cats at home. You can imagine his dismay when, one day, he took a kennel off the plane to find that the cat it carried was quite thoroughly dead. (No, it wasn't resting, nor pining for the fjords; this cat was bloody *deceased.*)

FOF knew how upset the owner would be, and came up with a plan to make everything better. At home, he had a cat of the same size, shape, and markings. He would substitute that cat, and since all cats treat all humans with equal disdain, the owner would never know the difference, and would never suffer the trauma of the loss of her cat. So FOF drove home, got his cat, put it in the kennel, and waited for the owner to show up—at which point, she took one look at the kennel and said, "This isn't my cat. My cat is dead."

As it turned out, she had shipped her recently deceased feline home to be buried. History does not record how our FOF dug himself out of this one.

Okay, but what's the point? The point is, if it isn't broken, don't fix it. And if it is broken, maybe that's all right, too. Which brings us, neat as a pin, to the topic of drawing lines in a serious hurry.

Fast Run-Length Slice Line Drawing

In the last chapter, we examined the principles of run-length slice line drawing, which draws lines a run at a time rather than a pixel at a time, a run being a series of pixels along the major (longer) axis. It's time to turn theory into useful practice by developing a fast assembly version. Listing 37.1 is the assembly version, in a form that's plug-compatible with the C code from the previous chapter.

LISTING 37.1 L37-1.ASM

```
; Fast run-length slice line drawing implementation for mode 0x13, the VGA's
; 320x200 256-color mode.
; Draws a line between the specified endpoints in color Color.
; C near-callable as:
;  void LineDraw(int XStart, int YStart, int XEnd, int YEnd, int Color)
; Tested with TASM

SCREEN_WIDTH        equ 320
SCREEN_SEGMENT      equ 0a000h
    .model  small
    .code

; Parameters to call.
parms   struc
           dw    ?                        ;pushed BP
           dw    ?                        ;pushed return address
XStart     dw    ?                        ;X start coordinate of line
YStart     dw    ?                        ;Y start coordinate of line
XEnd       dw    ?                        ;X end coordinate of line
YEnd       dw    ?                        ;Y end coordinate of line
Color      db    ?                        ;color in which to draw line
           db    ?                        ;dummy byte because Color is really a word
parms ends

; Local variables.
AdjUp               equ -2     ;error term adjust up on each advance
AdjDown             equ -4     ;error term adjust down when error term turns over
WholeStep           equ -6     ;minimum run length
XAdvance            equ -8     ;1 or -1, for direction in which X advances
LOCAL_SIZE          equ  8
    public  _LineDraw
```

```
_LineDraw   proc  near
      cld
      push       bp                        ;preserve caller's stack frame
      mov        bp,sp                     ;point to our stack frame
      sub sp,    LOCAL_SIZE                ;allocate space for local variables
      push       si                        ;preserve C register variables
      push       di
      push       ds                        ;preserve caller's DS
; We'll draw top to bottom, to reduce the number of cases we have to handle,
; and to make lines between the same endpoints always draw the same pixels.
      mov        ax,[bp].YStart
      cmp        ax,[bp].YEnd
      jle        LineIsTopToBottom
      xchg       [bp].YEnd,ax              ;swap endpoints
      mov        [bp].YStart,ax
      mov        bx,[bp].XStart
      xchg       [bp].XEnd,bx
      mov        [bp].XStart,bx
LineIsTopToBottom:
; Point DI to the first pixel to draw.
      mov        dx,SCREEN_WIDTH
      mul        dx    '                   ;YStart * SCREEN_WIDTH
      mov        si,[bp].XStart
      mov        di,si
      add        di,ax                     ;DI = YStart * SCREEN_WIDTH + XStart
                                           ; = offset of initial pixel
; Figure out how far we're going vertically (guaranteed to be positive).
      mov        cx,[bp].YEnd
      sub        cx,[bp].YStart            ;CX = YDelta
; Figure out whether we're going left or right, and how far we're going
; horizontally. In the process, special-case vertical lines, for speed and
; to avoid nasty boundary conditions and division by 0.
      mov        dx,[bp].XEnd
      sub        dx,si                     ;XDelta
      jnz        NotVerticalLine           ;XDelta == 0 means vertical line
                                           ;it is a vertical line
                                           ;yes, special case vertical line
      mov        ax,SCREEN_SEGMENT
      mov        ds,ax                     ;point DS:DI to the first byte to draw
      mov        al,[bp].Color
VLoop:
      mov        [di],al
      add        di,SCREEN_WIDTH
      dec        cx
      jns        VLoop
      jmp        Done
; Special-case code for horizontal lines.
      align      2
IsHorizontalLine:
      mov        ax,SCREEN_SEGMENT
      mov        es,ax                     ;point ES:DI to the first byte to draw
      mov        al,[bp].Color
      mov        ah,al                     ;duplicate in high byte for word access
      and        bx,bx                     ;left to right?
      jns        DirSet                    ;yes
      sub        di,dx                     ;currently right to left, point to left
                                           ; end so we can go left to right
                                           ; (avoids unpleasantness withright to
                                           ;  left REP STOSW)
```

```
DirSet:
     mov       cx,dx
     inc       cx                     ;# of pixels to draw
     shr       cx,1                   ;# of words to draw
     rep       stosw                  ;do as many words as possible
     adc       cx,cx
     rep       stosb                  ;do the odd byte, if there is one
     jmp       Done
; Special-case code for diagonal lines.
     align     2
IsDiagonalLine:
     mov       ax,SCREEN_SEGMENT
     mov       ds,ax                  ;point DS:DI to the first byte to draw
     mov       al,[bp].Color
     add       bx,SCREEN_WIDTH        ;advance distance from one pixel to next
DLoop:
     mov       [di],al
     add       di,bx
     dec       cx
     jns       DLoop
     jmp       Done

     align     2
NotVerticalLine:
     mov       bx,1                   ;assume left to right, so XAdvance = 1
                                      ;***leaves flags unchanged***
     jns       LeftToRight            ;left to right, all set
     neg       bx                     ;right to left, so XAdvance = -1
     neg       dx                     ;|XDelta|
LeftToRight:
; Special-case horizontal lines.
          and       cx,cx             ;YDelta == 0?
          jz        IsHorizontalLine  ;yes
; Special-case diagonal lines.
     cmp       cx,dx                  ;YDelta == XDelta?
     jz        IsDiagonalLine         ;yes
; Determine whether the line is X or Y major, and handle accordingly.
     cmp       dx,cx
     jae       XMajor
          jmp       YMajor
; X-major (more horizontal than vertical) line.
     align     2
XMajor:
     mov       ax,SCREEN_SEGMENT
     mov       es,ax                  ;point ES:DI to the first byte to draw
     and       bx,bx                  ;left to right?
     jns       DFSet                  ;yes, CLD is already set
     std                              ;right to left, so draw backwards
DFSet:
     mov       ax,dx                  ;XDelta
     sub       dx,dx                  ;prepare for division
     div       cx                     ;AX = XDelta/YDelta
                                      ; (minimum # of pixels in a run in this line)
                                      ;DX = XDelta % YDelta
     mov       bx,dx                  ;error term adjust each time Y steps by 1;
     add       bx,bx                  ; used to tell when one extra pixel should be
     mov       [bp].AdjUp,bx          ; drawn as part of a run, to account for
                                      ; fractional steps along the X axis per
                                      ; 1-pixel steps along Y
     mov       si,cx                  ;error term adjust when the error term turns
```

```
        add     si,si                   ; over, used to factor out the X step made at
        mov     [bp].AdjDown,si         ; that time
; Initial error term; reflects an initial step of 0.5 along the Y axis.
        sub     dx,si                   ;(XDelta % YDelta) - (YDelta * 2)
                                        ;DX = initial error term
; The initial and last runs are partial, because Y advances only 0.5 for
; these runs, rather than 1. Divide one full run, plus the initial pixel,
; between the initial and last runs.
        mov     si,cx                   ;SI = YDelta
        mov     cx,ax                   ;whole step (minimum run length)
        shr     cx,1
        inc     cx                      ;initial pixel count = (whole step / 2) + 1;
                                        ; (may be adjusted later). This is also the
                                        ; final run pixel count
        push    cx                      ;remember final run pixel count for later
; If the basic run length is even and there's no fractional advance, we have
; one pixel that could go to either the initial or last partial run, which
; we'll arbitrarily allocate to the last run.
; If there is an odd number of pixels per run, we have one pixel that can't
; be allocated to either the initial or last partial run, so we'll add 0.5 to
; the error term so this pixel will be handled by the normal full-run loop.
        add     dx,si                   ;assume odd length, add YDelta to error term
                                        ; (add 0.5 of a pixel to the error term)
        test    al,1                    ;is run length even?
        jnz     XMajorAdjustDone        ;no, already did work for odd case, all set
        sub     dx,si                   ;length is even, undo odd stuff we just did
        and     bx,bx                   ;is the adjust up equal to 0?
        jnz     XMajorAdjustDone        ;no (don't need to check for odd length,
                                        ; because of the above test)
        dec     cx                      ;both conditions met; make initial run 1
                                        ; shorter
XMajorAdjustDone:
        mov     [bp].WholeStep,ax;whole step (minimum run length)
        mov     al,[bp].Color           ;AL = drawing color
; Draw the first, partial run of pixels.
        rep     stosb                   ;draw the final run
        add     di,SCREEN_WIDTH         ;advance along the minor axis (Y)
; Draw all full runs.
        cmp     si,1                    ;are there more than 2 scans, so there are
                                        ; some full runs? (SI = # scans - 1)
        jna     XMajorDrawLast          ;no, no full runs
        dec     dx                      ;adjust error term by -1 so we can use
                                        ; carry test
        shr     si,1                    ;convert from scan to scan-pair count
        jnc     XMajorFullRunsOddEntry ;if there is an odd number of scans,
                                        ; do the odd scan now
XMajorFullRunsLoop:
        mov     cx,[bp].WholeStep;run is at least this long
        add     dx,bx                   ;advance the error term and add an extra
        jnc     XMajorNoExtra           ; pixel if the error term so indicates
        inc     cx                      ;one extra pixel in run
        sub     dx,[bp].AdjDown         ;reset the error term
XMajorNoExtra:
        rep     stosb                   ;draw this scan line's run
        add     di,SCREEN_WIDTH         ;advance along the minor axis (Y)
XMajorFullRunsOddEntry:                 ;enter loop here if there is an odd number
                                        ; of full runs
        mov     cx,[bp].WholeStep;run is at least this long
        add     dx,bx                   ;advance the error term and add an extra
        jnc     XMajorNoExtra2          ; pixel if the error term so indicates
```

```
        inc         cx                      ;one extra pixel in run
        sub         dx,[bp].AdjDown         ;reset the error term
XMajorNoExtra2:
        rep         stosb                   ;draw this scan line's run
        add         di,SCREEN_WIDTH         ;advance along the minor axis (Y)

        dec         si
        jnz         XMajorFullRunsLoop
; Draw the final run of pixels.
XMajorDrawLast:
        pop         cx                      ;get back the final run pixel length
        rep         stosb                   ;draw the final run

        cld                                 ;restore normal direction flag
        jmp         Done
; Y-major (more vertical than horizontal) line.
        align       2
YMajor:
        mov         [bp].XAdvance,bx        ;remember which way X advances
        mov         ax,SCREEN_SEGMENT
        mov         ds,ax                   ;point DS:DI to the first byte to draw
        mov         ax,cx                   ;YDelta
        mov         cx,dx                   ;XDelta
        sub         dx,dx                   ;prepare for division
        div         cx                      ;AX = YDelta/XDelta
                                            ; (minimum # of pixels in a run in this line)
                                            ;DX = YDelta % XDelta
        mov         bx,dx                   ;error term adjust each time X steps by 1;
        add         bx,bx                   ; used to tell when one extra pixel should be
        mov         [bp].AdjUp,bx           ; drawn as part of a run, to account for
                                            ; fractional steps along the Y axis per
                                            ; 1-pixel steps along X
        mov         si,cx                   ;error term adjust when the error term turns
        add         si,si                   ; over, used to factor out the Y step made at
        mov         [bp].AdjDown,si         ; that time

; Initial error term; reflects an initial step of 0.5 along the X axis.
        sub         dx,si                   ;(YDelta % XDelta) - (XDelta * 2)
                                            ;DX = initial error term
; The initial and last runs are partial, because X advances only 0.5 for
; these runs, rather than 1. Divide one full run, plus the initial pixel,
; between the initial and last runs.
        mov         si,cx                   ;SI = XDelta
        mov         cx,ax                   ;whole step (minimum run length)
        shr         cx,1
        inc         cx                      ;initial pixel count = (whole step / 2) + 1;
                                            ; (may be adjusted later)
        push        cx                      ;remember final run pixel count for later

; If the basic run length is even and there's no fractional advance, we have
; one pixel that could go to either the initial or last partial run, which
; we'll arbitrarily allocate to the last run.
; If there is an odd number of pixels per run, we have one pixel that can't
; be allocated to either the initial or last partial run, so we'll add 0.5 to
; the error term so this pixel will be handled by the normal full-run loop.
        add         dx,si                   ;assume odd length, add XDelta to error term
        test        al,1                    ;is run length even?
        jnz         YMajorAdjustDone        ;no, already did work for odd case, all set
        sub         dx,si                   ;length is even, undo odd stuff we just did
        and         bx,bx                   ;is the adjust up equal to 0?
```

```
     jnz        YMajorAdjustDone        ;no (don't need to check for odd length,
                                        ; because of the above test)
     dec        cx                      ;both conditions met; make initial run 1
                                        ; shorter
YMajorAdjustDone:
     mov        [bp].WholeStep,ax       ;whole step (minimum run length)
     mov        al,[bp].Color           ;AL = drawing color
     mov        bx,[bp].XAdvance        ;which way X advances
; Draw the first, partial run of pixels.
YMajorFirstLoop:
     mov        [di],al                 ;draw the pixel
     add        di,SCREEN_WIDTH         ;advance along the major axis (Y)
     dec        cx
     jnz        YMajorFirstLoop
     add        di,bx                   ;advance along the minor axis (X)
   ; Draw all full runs.
     cmp        si,1                    ;# of full runs. Are there more than 2
                                        ; columns, so there are some full runs?
                                        ; (SI = # columns - 1)
     jna        YMajorDrawLast          ;no, no full runs
     dec        dx                      ;adjust error term by -1 so we can use
                                        ; carry test
     shr        si,1                    ;convert from column to column-pair count
     jnc        YMajorFullRunsOddEntry  ;if there is an odd number of
                                        ; columns, do the odd column now
YMajorFullRunsLoop:
     mov        cx,[bp].WholeStep       ;run is at least this long
     add        dx,[bp].AdjUp           ;advance the error term and add an extra
     jnc        YMajorNoExtra           ; pixel if the error term so indicates
     inc        cx                      ;one extra pixel in run
     sub        dx,[bp].AdjDown         ;reset the error term
YMajorNoExtra:
  ;draw the run
YMajorRunLoop:
     mov        [di],al                 ;draw the pixel
     add        di,SCREEN_WIDTH         ;advance along the major axis (Y)
     dec        cx
     jnz        YMajorRunLoop
     add        di,bx                   ;advance along the minor axis (X)
YMajorFullRunsOddEntry:                 ;enter loop here if there is an odd number
                                        ; of full runs
     mov        cx,[bp].WholeStep       ;run is at least this long
     add        dx,[bp].AdjUp           ;advance the error term and add an extra
     jnc        YMajorNoExtra2          ; pixel if the error term so indicates
     inc        cx                      ;one extra pixel in run
     sub        dx,[bp].AdjDown         ;reset the error term
YMajorNoExtra2:
  ;draw the run
YMajorRunLoop2:
     mov        [di],al                 ;draw the pixel
     add        di,SCREEN_WIDTH         ;advance along the major axis (Y)
     dec        cx
     jnz        YMajorRunLoop2
     add        di,bx                   ;advance along the minor axis (X)

     dec        si
     jnz        YMajorFullRunsLoop
; Draw the final run of pixels.
YMajorDrawLast:
     pop        cx                      ;get back the final run pixel length
```

```
YMajorLastLoop:
     mov       [di],al              ;draw the pixel
     add       di,SCREEN_WIDTH      ;advance along the major axis (Y)
     dec       cx
     jnz       YMajorLastLoop
Done:
     pop       ds                   ;restore caller's DS
     pop       di
     pop       si                   ;restore C register variables
     mov       sp,bp                ;deallocate local variables
     pop       bp                   ;restore caller's stack frame
     ret
  _LineDraw   endp
     end
```

How Fast Is Fast?

Your first question is likely to be the following: Just how fast is Listing 37.1? Is it optimized to the hilt or just pretty fast? The quick answer is: It's *fast.* Listing 37.1 draws lines at a rate of nearly 1 million pixels per second on my 486/33, and is capable of still faster drawing, as I'll discuss shortly. (The heavily optimized AutoCAD line-drawing code that I mentioned in the last chapter drew 150,000 pixels per second on an EGA in a 386/16, and I thought I had died and gone to Heaven. Such is progress.) The full answer is a more complicated one, and ties in to the principle that if it is broken, maybe that's okay—and to the principle of looking before you leap, also known as profiling before you optimize.

When I went to speed up run-length slice lines, I initially manually converted the last chapter's C code into assembly. Then I streamlined the register usage and used **REP STOS** wherever possible. Listing 37.1 is that code. At that point, line drawing was surely faster, although I didn't know exactly how much faster. Equally surely, there were significant optimizations yet to be made, and I was itching to get on to them, for they were bound to be a lot more interesting than a basic C-to-assembly port.

Ego intervened at this point, however. I wanted to know how much of a speed-up I had already gotten, so I timed the performance of the C code and compared it to the assembly code. To my horror, I found that I had not gotten even a two-times improvement! I couldn't understand how that could be—the C code was decidedly unoptimized—until I hit on the idea of measuring the maximum memory speed of the VGA to which I was drawing.

Bingo. The Paradise VGA in my 486/33 is fast for a single display-memory write, because it buffers the data, lets the CPU go on its merry way, and finishes the write when display memory is ready. However, the maximum rate at which data can be written to the adapter turns out to be no more than one byte every microsecond. Put another way, you can only write one byte to this adapter every 33 clock cycles on a 486/33. Therefore, no matter how fast I made the line-drawing code, it could never draw more than 1,000,000 pixels per second in 256-color mode in my system. The C code was already drawing at about half that rate, so the potential speed-up for the assembly code was limited to a maximum of two times, which is pretty close to what Listing 37.1 did, in fact, achieve. When I compared the C and assembly implementations drawing to normal system (nondisplay) memory, I found that the assembly code was actually four times as fast as the C code.

In fact, Listing 37.1 draws VGA lines at about 92 percent of the maximum possible rate in my system—that is, it draws very nearly as fast as the VGA hardware will allow. All the optimization in the world would get me less than 10 percent faster line drawing—and only if I eliminated all overhead, an unlikely proposition at best. The code isn't fully optimized, but so what?

Now it's true that faster line-drawing code would likely be more beneficial on faster VGAs, especially local-bus VGAs, and in slower systems. For that reason, I'll list a variety of potential optimizations to Listing 37.1. On the other hand, it's also true that Listing 37.1 is capable of drawing lines at a rate of 2.2 million pixels per second on a 486/ 33, given fast enough VGA memory, so it should be able to drive almost any non-local-bus VGA at nearly full speed. In short, Listing 37.1 is very fast, and, in many systems, further optimization is basically a waste of time.

Profile before you optimize.

Further Optimizations

Following is a quick tour of some of the many possible further optimizations to Listing 37.1.

The run-handling loops could be unrolled more than the current two times. However, bear in mind that a two-times unrolling gets more than half the maximum unrolling benefit with less overhead than a more heavily unrolled loop.

BX could be freed up in the Y-major code by breaking out separate loops for X advances of 1 and –1. DX could be freed up by using AH as the counter for the run loops, although this would limit the maximum line length that could be handled. The freed registers could be used to keep more of the whole-step and error variables in registers. Alternatively, the freed registers could be used to implement more esoteric approaches like unrolling the Y-major inner loop; such unrolling could take advantage of the knowledge that only two run lengths are possible for any given line. Strangely enough, on the 486 it might also be worth unrolling the X-major inner loop, which consists of **REP STOSB**, because of the slow start-up time of **REP** relative to the speed of branching on that processor.

Special code could be implemented for lines with integral slopes, because all runs are exactly the same length in such lines. Also, the X-major code could try to write an aligned word at a time to display memory whenever possible; this would improve the maximum possible performance on some 16-bit VGAs.

One weakness of Listing 37.1 is that for lines with slopes between 0.5 and 2, the average run length is less than two, rendering run-length slicing ineffective. This can be remedied by viewing lines in that range as being composed of diagonal, rather than horizontal or vertical runs. I haven't space to take this idea any further in this book, but it's not very complicated, and it guarantees a minimum run length of 2, which renders run drawing considerably more efficient, and makes techniques such as unrolling the inner run-drawing loops more attractive.

Finally, be aware that run-length slice drawing is best for long lines, because it has more and slower setup than a standard Bresenham's line draw, including a divide. Run-length slice is great for 100-pixel lines, but not necessarily for 20-pixel lines, and it's a sure thing that it's not terrific for 3-pixel lines. Both approaches will work, but if line-drawing performance is critical, whether you'll want to use run-length slice or standard Bresenham's depends on the typical lengths of the lines you'll be drawing. For lines of widely varying lengths, you might want to implement both approaches, and choose the best one for each line, depending on the line length—assuming, of course, that your display memory is fast enough and your application demanding enough to make that level of optimization worthwhile.

If your code looks broken from a performance perspective, think before you fix it; that particular cat may be dead for a perfectly good reason. I'll say it again: *Profile before you optimize.*