Chapter 39

Fast Convex Polygons

Chapter 39

Filling Polygons in a Hurry

In the previous chapter, we explored the surprisingly intricate process of filling convex polygons. Now we're going to fill them an order of magnitude or so faster.

Two thoughts may occur to some of you at this point: "Oh, no, he's not going to get into assembly language and device-dependent code, is he?" and, "Why bother with polygon filling—or, indeed, any drawing primitives—anyway? Isn't that what GUIs and third-party libraries are for?"

To which I answer, "Well, yes, I am," and, "If you have to ask, you've missed the magic of microcomputer programming." Actually, both questions ask the same thing, and that is: "Why should I, as a programmer, have any idea how my program actually works?"

Put that way, it sounds a little different, doesn't it?

GUIs, reusable code, portable code written entirely in high-level languages, and object-oriented programming are all the rage now, and promise to remain so for the foreseeable future. The thrust of this technology is to enhance the software development process by offloading as much responsibility as possible to other programmers, and by writing all remaining code in modular, generic form. This modular code then becomes a black box to be reused endlessly without another thought about what actually lies inside. GUIs also reduce development time by making many interface choices for you. That, in turn, makes it possible to create quickly and reliably programs that will be easy for new users to pick up, so software becomes easier to both produce and learn. This is, without question, a Good Thing.

The "black box" approach does not, however, necessarily cause the software itself to become faster, smaller, or more innovative; quite the opposite, I suspect. I'll reserve judgement on whether that is a good thing or not, but I'll make a prediction: In the short run, the aforementioned techniques will lead to noticeably larger, slower programs, as programmers understand less and less of what the key parts of their programs do and rely increasingly on general-purpose code written by other people. (In the long run, programs will be bigger and slower yet, but computers will be so fast and will have so much memory that no one will care.) Over time, PC programs will also come to be more similar to one another—and to programs running on other platforms, such as the Mac—as regards both user interface and performance.

Again, I am not saying that this is bad. It does, however, have major implications for the future nature of PC graphics programming, in ways that will directly affect the means by which many of you earn your livings. Not so very long from now, graphics programming—all programming, for that matter—will become mostly a matter of assembling in various ways components written by other people, and will cease to be the all-inclusively creative, mindbendingly complex pursuit it is today. (Using legally certified black boxes is, by the way, one direction in which the patent lawyers are leading us; legal considerations may be the final nail in the coffin of homegrown code.) For now, though, it's still within your power, as a PC programmer, to understand and even control every single thing that happens on a computer if you so desire, to realize any vision you may have. Take advantage of this unique window of opportunity to create some magic!

Neither does it hurt to understand what's involved in drawing, say, a filled polygon, even if you are using a GUI. You will better understand the performance implications of the available GUI functions, and you will be able to fill in any gaps in the functions provided. You may even find that you can outperform the GUI on occasion by doing your own drawing into a system memory bitmap, then copying the result to the screen; for instance, you can do this under Windows by using the WinG library available from Microsoft. You will also be able to understand why various quirks exist, and will be able to put them to good use. For example, the X Window System follows the polygon drawing rules described in the previous chapter (although it's not obvious from the X Window System documentation); if you understood the previous chapter's discussion, you're in good shape to use polygons under X.

In short, even though doing so runs counter to current trends, it helps to understand how things work, especially when they're very visible parts of the software you develop. That said, let's learn more about filling convex polygons.

Fast Convex Polygon Filling

In addressing the topic of filling convex polygons in the previous chapter, the implementation we came up with met all of our functional requirements. In particular, it met stringent rules that guaranteed that polygons would never overlap or have gaps

at shared edges, an important consideration when building polygon-based images. Unfortunately, the implementation was also slow as molasses. In this chapter we'll work up polygon-filling code that's fast enough to be truly usable.

Our original polygon filling code involved three major tasks, each performed by a separate function:

- Tracing each polygon edge to generate a coordinate list (performed by the function **ScanEdge**);
- Drawing the scanned-out horizontal lines that constitute the filled polygon (**DrawHorizontalLineList**); and
- Characterizing the polygon and coordinating the tracing and drawing (**FillConvexPolygon**).

The amount of time that the previous chapter's sample program spent in each of these areas is shown in Table 39.1. As you can see, half the time was spent drawing and the other half was spent tracing the polygon edges (the time spent in **FillConvexPolygon** was relatively minuscule), so we have our choice of where to begin optimizing.

Fast Drawing

Let's start with drawing, which is easily sped up. The previous chapter's code used a double-nested loop that called a draw-pixel function to plot each pixel in the polygon individually. That's a ridiculous approach in a graphics mode that offers linearly mapped memory, as does VGA mode 13H, the mode in which we're working. At the very least, we could point a far pointer to the left edge of each polygon scan line, then draw each pixel in that scan line in quick succession, using something along the lines of ***ScrPtr++ = FillColor;** inside a loop.

However, it seems silly to use a loop when the x86 has an instruction, **REP STOS**, that's uniquely suited to filling linear memory buffers. There's no way to use **REP STOS** directly in C code, but it's a good bet that the **memset** library function uses **REP STOS**, so you could greatly enhance performance by using **memset** to draw each scan line of the polygon in a single shot. That, however, is easier said than done. The **memset** function linked in from the library is tied to the memory model in use; in small (which includes Tiny, Small, or Medium) data models **memset** accepts only near pointers, so it can't be used to access screen memory. Consequently, a large (which includes Compact, Large, or Huge) data model must be used to allow **memset** to draw to display memory—a clear case of the tail wagging the dog. This is an excellent example of why, although it is possible to use C to do virtually anything, it's sometimes much simpler just to use a little assembly code and be done with it.

At any rate, Listing 39.1 for this chapter shows a version of **DrawHorizontalLineList** that uses memset to draw each scan line of the polygon in a single call. When linked to Chapter 38's test program, Listing 39.1 increases pure drawing speed (disregarding edge tracing and other nondrawing time) by more than an order of magnitude

Implementation	Total Polygon Filling Time	DrawHorizontal LineList	ScanEdge	FillConvex Polygon
Drawing to display memory in mode 13h				
C floating point scan/ DrawPixel drawing code from Chapter 38, (small model)	11.69	5.80 seconds (50% of total)	5.86 (50%)	0.03 (<1%)
C floating point scan/ memset drawing (Listing 39.1, compact model)	6.64	0.49 (7%)	6.11 (92%)	0.04 (<1%)
C integer scan/ memset drawing (Listing 39.1 & Listing 39.2, compact model)	0.60	0.49 (82%)	0.07 (12%)	0.04 (7%)
C integer scan/ ASM drawing (Listing 39.2 & Listing 39.3, small model)	0.45	0.36 (80%)	0.06 (13%)	0.03 (7%)
ASM integer scan/ ASM drawing (Listing 40.3 & Listing 40.4, small model)	0.42	0.36 (86%)	0.03 (7%)	0.03 (7%)
Drawing to system memory				
C integer scan/ memset drawing (Listing 39.1 & Listing 39.2, compact model)	0.31	0.20 (65%)	0.07 (23%)	0.04 (13%)
ASM integer scan/ ASM drawing (Listing 39.3 & Listing 39.4, small model)	0.13	0.07 (54%)	0.03 (23%)	0.03 (23%)

All times are in seconds, as measured with Turbo Profiler on a 20-MHz cached 386 with no math coprocessor installed. Note that time spent in **main()** is not included. C code was compiled with Borland C++ with maximum optimization (-G -O -Z -r -a); assembly language code was assembled with TASM. Percentages of combined times are rounded to the nearest percent, so the sum of the three percentages does not always equal 100.

Table 39.1 Polygon fill performance.

over Chapter 38's draw-pixel-based code, despite the fact that Listing 39.1 requires a large (in this case, the Compact) data model. Listing 39.1 works fine with Borland C++, but may not work with other compilers, for it relies on the aforementioned interaction between **memset** and the selected memory model.

LISTING 39.1 L39-1.C

```
/* Draws all pixels in the list of horizontal lines passed in, in
   mode 13h, the VGA's 320x200 256-color mode. Uses memset to fill
   each line, which is much faster than using DrawPixel but requires
   that a large data model (compact, large, or huge) be in use when
   running in real mode or 286 protected mode.
   All C code tested with Borland C++. */

#include <string.h>
#include <dos.h>
#include "polygon.h"

#define SCREEN_WIDTH    320
#define SCREEN_SEGMENT  0xA000

void DrawHorizontalLineList(struct HLineList * HLineListPtr,
      int Color)
{
   struct HLine *HLinePtr;
   int Length, Width;
   unsigned char far *ScreenPtr;

   /* Point to the start of the first scan line on which to draw */
   ScreenPtr = MK_FP(SCREEN_SEGMENT,
         HLineListPtr->YStart * SCREEN_WIDTH);

   /* Point to the XStart/XEnd descriptor for the first (top)
      horizontal line */
   HLinePtr = HLineListPtr->HLinePtr;
   /* Draw each horizontal line in turn, starting with the top one and
      advancing one line each time */
   Length = HLineListPtr->Length;
   while (Length-- > 0) {
      /* Draw the whole horizontal line if it has a positive width */
      if ((Width = HLinePtr->XEnd - HLinePtr->XStart + 1) > 0)
         memset(ScreenPtr + HLinePtr->XStart, Color, Width);
      HLinePtr++;                /* point to next scan line X info */
      ScreenPtr += SCREEN_WIDTH; /* point to next scan line start */
   }
}
```

At this point, I'd like to mention that benchmarks are notoriously unreliable; the results in Table 39.1 are accurate *only* for the test program, and only when running on a particular system. Results could be vastly different if smaller, larger, or more complex polygons were drawn, or if a faster or slower computer/VGA combination were used. These factors notwithstanding, the test program does fill a variety of polygons of varying complexity sized from large to small and in between, and certainly the order of magnitude difference between Listing 39.1 and the old version of **DrawHorizontalLineList** is a clear indication of which code is superior.

Anyway, Listing 39.1 has the desired effect of vastly improving drawing time. There are cycles yet to be had in the drawing code, but as tracing polygon edges now takes 92 percent of the polygon filling time, it's logical to optimize the tracing code next.

Fast Edge Tracing

There's no secret as to why last chapter's **ScanEdge** was so slow: It used floating point calculations. One secret of fast graphics is using integer or fixed-point calculations, instead. (Sure, the floating point code would run faster if a math coprocessor were installed, but it would still be slower than the alternatives; besides, why require a math coprocessor when you don't have to?) Both integer and fixed-point calculations are fast. In many cases, fixed-point is faster, but integer calculations have one tremendous virtue: They're completely accurate. The tiny imprecision inherent in either fixed or floating-point calculations can result in occasional pixels being one position off from their proper location. This is no great tragedy, but after going to so much trouble to ensure that polygons don't overlap at common edges, why not get it exactly right?

In fact, when I tested out the integer edge tracing code by comparing an integer-based test image to one produced by floating-point calculations, two pixels out of the whole screen differed, leading me to suspect a bug in the integer code. It turned out, however, that's in those two cases, the floating point results were sufficiently imprecise to creep from just under an integer value to just over it, so that the **ceil** function returned a coordinate that was one too large.

Floating point is very accurate—but it is not precise. Integer calculations, properly performed, are.

Listing 39.2 shows a C implementation of integer edge tracing. Vertical and diagonal lines, which are trivial to trace, are special-cased. Other lines are broken into two categories: Y-major (closer to vertical) and X-major (closer to horizontal). The handlers for the Y-major and X-major cases operate on the principle of similar triangles: The number of X pixels advanced per scan line is the same as the ratio of the X delta of the edge to the Y delta. Listing 39.2 is more complex than the original floating point implementation, but not painfully so. In return for that complexity, Listing 39.2 is more than 80 times faster at scanning edges—and, as just mentioned, it's actually more accurate than the floating point code.

Ya gotta love that integer arithmetic.

LISTING 39.2 L39-2.C

```
/* Scan converts an edge from (X1,Y1) to (X2,Y2), not including the
   point at (X2,Y2). If SkipFirst — 1, the point at (X1,Y1) isn't
   drawn; if SkipFirst — 0, it is. For each scan line, the pixel
   closest to the scanned edge without being to the left of the
   scanned edge is chosen. Uses an all-integer approach for speed and
   precision. */
```

```
#include <math.h>
#include "polygon.h"

void ScanEdge(int X1, int Y1, int X2, int Y2, int SetXStart,
      int SkipFirst, struct HLine **EdgePointPtr)
{
   int Y, DeltaX, Height, Width, AdvanceAmt, ErrorTerm, i;
   int ErrorTermAdvance, XMajorAdvanceAmt;
   struct HLine *WorkingEdgePointPtr;

   WorkingEdgePointPtr = *EdgePointPtr; /* avoid double dereference */
   AdvanceAmt = ((DeltaX = X2 - X1) > 0) ? 1 : -1;
                           /* direction in which X moves (Y2 is
                              always > Y1, so Y always counts up) */

   if ((Height = Y2 - Y1) <= 0)  /* Y length of the edge */
      return;     /* guard against 0-length and horizontal edges */

   /* Figure out whether the edge is vertical, diagonal, X-major
      (mostly horizontal), or Y-major (mostly vertical) and handle
      appropriately */
   if ((Width = abs(DeltaX)) == 0) {
      /* The edge is vertical; special-case by just storing the same
         X coordinate for every scan line */
      /* Scan the edge for each scan line in turn */
      for (i = Height - SkipFirst; i-- > 0; WorkingEdgePointPtr++) {
         /* Store the X coordinate in the appropriate edge list */
         if (SetXStart == 1)
            WorkingEdgePointPtr->XStart = X1;
         else
            WorkingEdgePointPtr->XEnd = X1;
      }
   } else if (Width == Height) {
      /* The edge is diagonal; special-case by advancing the X
         coordinate 1 pixel for each scan line */
      if (SkipFirst) /* skip the first point if so indicated */
         X1 += AdvanceAmt; /* move 1 pixel to the left or right */
      /* Scan the edge for each scan line in turn */
      for (i = Height - SkipFirst; i-- > 0; WorkingEdgePointPtr++) {
         /* Store the X coordinate in the appropriate edge list */
         if (SetXStart == 1)
            WorkingEdgePointPtr->XStart = X1;
         else
            WorkingEdgePointPtr->XEnd = X1;
         X1 += AdvanceAmt; /* move 1 pixel to the left or right */
      }
   } else if (Height > Width) {
      /* Edge is closer to vertical than horizontal (Y-major) */
      if (DeltaX >= 0)
         ErrorTerm = 0; /* initial error term going left->right */
      else
         ErrorTerm = -Height + 1;   /* going right->left */
      if (SkipFirst) {   /* skip the first point if so indicated */
         /* Determine whether it's time for the X coord to advance */
         if ((ErrorTerm += Width) > 0) {
            X1 += AdvanceAmt; /* move 1 pixel to the left or right */
            ErrorTerm -= Height; /* advance ErrorTerm to next point */
         }
      }
      /* Scan the edge for each scan line in turn */
      for (i = Height - SkipFirst; i-- > 0; WorkingEdgePointPtr++) {
```

```
            /* Store the X coordinate in the appropriate edge list */
            if (SetXStart == 1)
               WorkingEdgePointPtr->XStart = X1;
            else
               WorkingEdgePointPtr->XEnd = X1;
            /* Determine whether it's time for the X coord to advance */
            if ((ErrorTerm += Width) > 0) {
               X1 += AdvanceAmt; /* move 1 pixel to the left or right */
               ErrorTerm -= Height; /* advance ErrorTerm to correspond */
            }
         }
      } else {
         /* Edge is closer to horizontal than vertical (X-major) */
         /* Minimum distance to advance X each time */
         XMajorAdvanceAmt = (Width / Height) * AdvanceAmt;
         /* Error term advance for deciding when to advance X 1 extra */
         ErrorTermAdvance = Width % Height;
         if (DeltaX >= 0)
            ErrorTerm = 0; /* initial error term going left->right */
         else
            ErrorTerm = -Height + 1;   /* going right->left */
         if (SkipFirst) {   /* skip the first point if so indicated */
            X1 += XMajorAdvanceAmt;    /* move X minimum distance */
            /* Determine whether it's time for X to advance one extra */
            if ((ErrorTerm += ErrorTermAdvance) > 0) {
               X1 += AdvanceAmt;       /* move X one more */
               ErrorTerm -= Height; /* advance ErrorTerm to correspond */
            }
         }
         /* Scan the edge for each scan line in turn */
         for (i = Height - SkipFirst; i-- > 0; WorkingEdgePointPtr++) {
            /* Store the X coordinate in the appropriate edge list */
            if (SetXStart == 1)
               WorkingEdgePointPtr->XStart = X1;
            else
               WorkingEdgePointPtr->XEnd = X1;
            X1 += XMajorAdvanceAmt;    /* move X minimum distance */
            /* Determine whether it's time for X to advance one extra */
            if ((ErrorTerm += ErrorTermAdvance) > 0) {
               X1 += AdvanceAmt;       /* move X one more */
               ErrorTerm -= Height; /* advance ErrorTerm to correspond */
            }
         }
      }

      *EdgePointPtr = WorkingEdgePointPtr;   /* advance caller's ptr */
   }
```

The Finishing Touch: Assembly Language

The C implementation in Listing 39.2 is now nearly 20 times as fast as the original, which is good enough for most purposes. Still, it requires that one of the large data models be used (for **memset**), and it's certainly not the fastest possible code. The obvious next step is assembly language.

Listing 39.3 is an assembly language version of **DrawHorizontalLineList**. In actual use, it proved to be about 36 percent faster than Listing 39.1; better than a poke in the eye with a sharp stick, but just barely. There's more to these timing results than

meets that eye, though. Display memory generally responds much more slowly than system memory, especially in 386 and 486 systems. That means that much of the time taken by Listing 39.3 is actually spent waiting for display memory accesses to complete, with the processor forced to idle by wait states. If, instead, Listing 39.3 drew to a local buffer in system memory or to a particularly fast VGA, the assembly implementation might well display a far more substantial advantage over the C code.

And indeed it does. When the test program is modified to draw to a local buffer, both the C and assembly language versions get 0.29 seconds faster, that being a measure of the time taken by display memory wait states. With those wait states factored out, the assembly language version of **DrawHorizontalLineList** becomes almost three times as fast as the C code.

There is a lesson here. An optimization has no fixed payoff; its value fluctuates according to the context in which it is used. There's relatively little benefit to further optimizing code that already spends half its time waiting for display memory; no matter how good your optimizations, you'll get only a two-times speedup at best, and generally much less than that. There is, on the other hand, potential for tremendous improvement when drawing to system memory, so if that's where most of your drawing will occur, optimizations such as Listing 39.3 are well worth the effort.

Know the environments in which your code will run, and know where the cycles go in those environments.

LISTING 39.3 L39-3.ASM

```
; Draws all pixels in the list of horizontal lines passed in, in
; mode 13h, the VGA's 320x200 256-color mode. Uses REP STOS to fill
; each line.
; C near-callable as:
;     void DrawHorizontalLineList(struct HLineList * HLineListPtr,
;          int Color);
; All assembly code tested with TASM and MASM

SCREEN_WIDTH    equ   320
SCREEN_SEGMENT  equ   0a000h

HLine struc
XStart          dw    ?           ;X coordinate of leftmost pixel in line
XEnd            dw    ?           ;X coordinate of rightmost pixel in line
HLine           ends

HLineList struc
Lngth           dw    ?           ;# of horizontal lines
YStart          dw    ?           ;Y coordinate of topmost line
HLinePtr        dw    ?           ;pointer to list of horz lines
HLineList       ends

Parms struc
                dw    2 dup(?)    ;return address & pushed BP
HLineListPtr    dw    ?           ;pointer to HLineList structure
Color           dw    ?           ;color with which to fill
Parms           ends
```

```
        .model small
        .code
        public _DrawHorizontalLineList
        align   2
_DrawHorizontalLineList   proc
        push  bp                        ;preserve caller's stack frame
        mov   bp,sp                     ;point to our stack frame
        push  si                        ;preserve caller's register variables
        push  di
        cld                             ;make string instructions inc pointers

        mov   ax,SCREEN_SEGMENT
        mov   es,ax                     ;point ES to display memory for REP STOS

        mov   si,[bp+HLineListPtr]      ;point to the line list
        mov   ax,SCREEN_WIDTH           ;point to the start of the first scan
        mul   [si+YStart]               ; line in which to draw
        mov   dx,ax                     ;ES:DX points to first scan line to
                                        ; draw
        mov   bx,[si+HLinePtr]          ;point to the XStart/XEnd descriptor
                                        ; for the first (top) horizontal line
        mov   si,[si+Lngth]             ;# of scan lines to draw
        and   si,si                     ;are there any lines to draw?
        jz    FillDone                  ;no, so we're done
        mov   al,byte ptr [bp+Color]    ;color with which to fill
        mov   ah,al                     ;duplicate color for STOSW
FillLoop:
        mov   di,[bx+XStart]            ;left edge of fill on this line
        mov   cx,[bx+XEnd]              ;right edge of fill
        sub   cx,di
        js    LineFillDone              ;skip if negative width
        inc   cx                        ;width of fill on this line
        add   di,dx                     ;offset of left edge of fill
        test  di,1                      ;does fill start at an odd address?
        jz    MainFill                  ;no
        stosb                           ;yes, draw the odd leading byte to
                                        ; word-align the rest of the fill
        dec   cx                        ;count off the odd leading byte
        jz    LineFillDone              ;done if that was the only byte
MainFill:
        shr   cx,1                      ;# of words in fill
        rep   stosw                     ;fill as many words as possible
        adc   cx,cx                     ;1 if there's an odd trailing byte to
                                        ; do, 0 otherwise
        rep   stosb                     ;fill any odd trailing byte
LineFillDone:
        add   bx,size HLine             ;point to the next line descriptor
        add   dx,SCREEN_WIDTH           ;point to the next scan line
        dec   si                        ;count off lines to fill
        jnz   FillLoop
FillDone:
        pop   di                        ;restore caller's register variables
        pop   si
        pop   bp                        ;restore caller's stack frame
        ret
_DrawHorizontalLineList   endp
        end
```

Maximizing REP STOS

Listing 39.3 doesn't take the easy way out and use **REP STOSB** to fill each scan line; instead, it uses **REP STOSW** to fill as many pixel pairs as possible via word-sized accesses, using **STOSB** only to do odd bytes. Word accesses to odd addresses are always split by the processor into 2-byte accesses. Such word accesses take twice as long as word accesses to even addresses, so Listing 39.3 makes sure that all word accesses occur at even addresses, by performing a leading **STOSB** first if necessary.

Listing 39.3 is another case in which it's worth knowing the environment in which your code will run. Extra code is required to perform aligned word-at-a-time filling, resulting in extra overhead. For very small or narrow polygons, that overhead might overwhelm the advantage of drawing a word at a time, making plain old **REP STOSB** faster.

Faster Edge Tracing

Finally, Listing 39.4 is an assembly language version of **ScanEdge**. Listing 39.4 is a relatively straightforward translation from C to assembly, but is nonetheless about twice as fast as Listing 39.2.

The version of **ScanEdge** in Listing 39.4 could certainly be sped up still further by unrolling the loops. **FillConvexPolygon**, the overall coordination routine, hasn't even been converted to assembly language, so that could be sped up as well. I haven't bothered with these optimizations because all code other than **DrawHorizontalLineList** takes only 14 percent of the overall polygon filling time when drawing to display memory; the potential return on optimizing nondrawing code simply isn't great enough to justify the effort. Part of the value of a profiler is being able to tell when to stop optimizing; with Listings 39.3 and 39.4 in use, more than two-thirds of the time taken to draw polygons is spent waiting for display memory, so optimization is pretty much maxed out. However, further optimization might be worthwhile when drawing to system memory, where wait states are out of the picture and the nondrawing code takes a significant portion (46 percent) of the overall time.

Again, *know where the cycles go.*

By the way, note that all the versions of **ScanEdge** and **FillConvexPolygon** that we've looked at are adapter-independent, and that the C code is also machine-independent; all adapter-specific code is isolated in **DrawHorizontalLineList**. This makes it easy to add support for other graphics systems, such as the 8514/A, the XGA, or, for that matter, a completely non-PC system.

LISTING 39.4 L39-4.ASM

```
; Scan converts an edge from (X1,Y1) to (X2,Y2), not including the
; point at (X2,Y2). If SkipFirst == 1, the point at (X1,Y1) isn't
; drawn; if SkipFirst == 0, it is. For each scan line, the pixel
; closest to the scanned edge without being to the left of the scanned
; edge is chosen. Uses an all-integer approach for speed & precision.
```

```
; C near-callable as:
;    void ScanEdge(int X1, int Y1, int X2, int Y2, int SetXStart,
;     int SkipFirst, struct HLine **EdgePointPtr);
; Edges must not go bottom to top; that is, Y1 must be <= Y2.
; Updates the pointer pointed to by EdgePointPtr to point to the next
;  free entry in the array of HLine structures.

HLine   struc
XStart          dw    ?           ;X coordinate of leftmost pixel in scan line
XEnd            dw    ?           ;X coordinate of rightmost pixel in scan line
HLine   ends

Parms   struc
                dw    2 dup(?)    ;return address & pushed BP
X1              dw    ?           ;X start coord of edge
Y1              dw    ?           ;Y start coord of edge
X2              dw    ?           ;X end coord of edge
Y2              dw    ?           ;Y end coord of edge
SetXStart       dw    ?           ;1 to set the XStart field of each
                                  ; HLine struc, 0 to set XEnd
SkipFirst       dw    ?           ;1 to skip scanning the first point
                                  ; of the edge, 0 to scan first point
EdgePointPtr    dw    ?           ;pointer to a pointer to the array of
                                  ; HLine structures in which to store
                                  ; the scanned X coordinates
Parms   ends

;Offsets from BP in stack frame of local variables.
AdvanceAmt      equ     -2
Height          equ     -4
LOCAL_SIZE      equ      4        ;total size of local variables

      .model small
      .code
      public _ScanEdge
      align  2
_ScanEdge  proc
      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 caller's register variables
      push  di
      mov   di,[bp+EdgePointPtr]
      mov   di,[di]               ;point to the HLine array
      cmp   [bp+SetXStart],1      ;set the XStart field of each HLine
                                  ; struc?
      jz    HLinePtrSet           ;yes, DI points to the first XStart
      add   di,XEnd               ;no, point to the XEnd field of the
                                  ; first HLine struc
HLinePtrSet:
      mov   bx,[bp+Y2]
      sub   bx,[bp+Y1]            ;edge height
      jle   ToScanEdgeExit        ;guard against 0-length & horz edges
      mov   [bp+Height],bx        ;Height = Y2 - Y1
      sub   cx,cx                 ;assume ErrorTerm starts at 0 (true if
                                  ; we're moving right as we draw)
      mov   dx,1                  ;assume AdvanceAmt = 1 (move right)
      mov   ax,[bp+X2]
      sub   ax,[bp+X1]            ;DeltaX = X2 - X1
      jz    IsVertical            ;it's a vertical edge--special case it
```

```
        jns     SetAdvanceAmt           ;DeltaX >= 0
        mov     cx,1                    ;DeltaX < 0 (move left as we draw)
        sub     cx,bx                   ;ErrorTerm = -Height + 1
        neg     dx                      ;AdvanceAmt = -1 (move left)
        neg     ax                      ;Width = abs(DeltaX)
SetAdvanceAmt:
        mov     [bp+AdvanceAmt],dx
; Figure out whether the edge is diagonal, X-major (more horizontal),
; or Y-major (more vertical) and handle appropriately.
        cmp     ax,bx                   ;if Width==Height, it's a diagonal edge
        jz      IsDiagonal              ;it's a diagonal edge--special case
        jb      YMajor                  ;it's a Y-major (more vertical) edge
                                        ;it's an X-major (more horz) edge
        sub     dx,dx                   ;prepare DX:AX (Width) for division
        div     bx                      ;Width/Height
                                        ;DX = error term advance per scan line
        mov     si,ax                   ;SI = minimum # of pixels to advance X
                                        ; on each scan line
        test    [bp+AdvanceAmt],8000h   ;move left or right?
        jz      XMajorAdvanceAmtSet     ;right, already set
        neg     si                      ;left, negate the distance to advance
                                        ; on each scan line
XMajorAdvanceAmtSet:    ;
        mov     ax,[bp+X1]              ;starting X coordinate
                cmp   [bp+SkipFirst],1  ;skip the first point?
        jz      XMajorSkipEntry         ;yes
XMajorLoop:
        mov     [di],ax                 ;store the current X value
        add     di,size HLine           ;point to the next HLine struc
XMajorSkipEntry:
        add     ax,si                   ;set X for the next scan line
        add     cx,dx                   ;advance error term
        jle     XMajorNoAdvance         ;not time for X coord to advance one
                                        ; extra
        add     ax,[bp+AdvanceAmt]      ;advance X coord one extra
        sub     cx,[bp+Height]          ;adjust error term back
XMajorNoAdvance:
        dec     bx                      ;count off this scan line
        jnz     XMajorLoop
        jmp     ScanEdgeDone
        align 2
ToScanEdgeExit:
        jmp     ScanEdgeExit
        align 2
IsVertical:
        mov     ax,[bp+X1]              ;starting (and only) X coordinate
        sub     bx,[bp+SkipFirst]       ;loop count = Height - SkipFirst
        jz      ScanEdgeExit            ;no scan lines left after skipping 1st
VerticalLoop:
        mov     [di],ax                 ;store the current X value
        add     di,size HLine           ;point to the next HLine struc
        dec     bx                      ;count off this scan line
        jnz     VerticalLoop
        jmp     ScanEdgeDone
        align 2
IsDiagonal:
        mov     ax,[bp+X1]              ;starting X coordinate
        cmp     [bp+SkipFirst],1        ;skip the first point?
        jz      DiagonalSkipEntry;yes
```

```
DiagonalLoop:
      mov   [di],ax               ;store the current X value
      add   di,size HLine         ;point to the next HLine struc
DiagonalSkipEntry:
      add   ax,dx                 ;advance the X coordinate
      dec   bx                    ;count off this scan line
      jnz   DiagonalLoop
      jmp   ScanEdgeDone
      align 2
YMajor:
      push  bp                    ;preserve stack frame pointer
      mov   si,[bp+X1]            ;starting X coordinate
      cmp   [bp+SkipFirst],1      ;skip the first point?
      mov   bp,bx                 ;put Height in BP for error term calcs
      jz    YMajorSkipEntry       ;yes, skip the first point
YMajorLoop:
      mov   [di],si               ;store the current X value
      add   di,size HLine         ;point to the next HLine struc
YMajorSkipEntry:
      add   cx,ax                 ;advance the error term
      jle   YMajorNoAdvance       ;not time for X coord to advance
      add   si,dx                 ;advance the X coordinate
      sub   cx,bp                 ;adjust error term back
YMajorNoAdvance:
      dec   bx                    ;count off this scan line
      jnz   YMajorLoop
      pop   bp                    ;restore stack frame pointer
ScanEdgeDone:
      cmp   [bp+SetXStart],1      ;were we working with XStart field?
      jz    UpdateHLinePtr        ;yes, DI points to the next XStart
      sub   di,XEnd               ;no, point back to the XStart field
UpdateHLinePtr:
      mov   bx,[bp+EdgePointPtr]  ;point to pointer to HLine array
      mov   [bx],di               ;update caller's HLine array pointer
ScanEdgeExit:
      pop   di                    ;restore caller's register variables
      pop   si
      mov   sp,bp                 ;deallocate local variables
      pop   bp                    ;restore caller's stack frame
      ret
_ScanEdge   endp
      end
```