Chapter 41

Those Way-Down Polygon Nomenclature Blues

Chapter 41

Names Do Matter when You Conceptualize a Data Structure

After I wrote the columns on polygons in *Dr. Dobb's Journal* that became Chapters 38–40, long-time reader Bill Huber wrote to take me to task—and a well-deserved kick in the fanny it was, I might add—for my use of non-standard polygon terminology in those columns. Unix's X-Window System (XWS) defines three categories of polygons: complex, nonconvex, and convex. These three categories, each a specialized subset of the preceding category, not-so-coincidentally map quite nicely to three increasingly fast polygon filling techniques. Therefore, I used the XWS names to describe the sorts of polygons that can be drawn with each of the polygon filling techniques.

The problem is that those names don't accurately describe all the sorts of polygons that the techniques are capable of drawing. Convex polygons are those for which no interior angle is greater than 180 degrees. The "convex" drawing approach described in the previous few chapters actually handles a number of polygons that are not convex; in fact, it can draw any polygon through which no horizontal line can be drawn that intersects the boundary more than twice. (In other words, the boundary reverses the Y direction exactly twice, disregarding polygons that have degenerated into horizontal lines, which I'm going to ignore.)

Bill was kind enough to send me the pages out of *Computational Geometry, An Introduction* (Springer-Verlag, 1988) that describe the correct terminology; such polygons are, in fact, "monotone with respect to a vertical line" (which unfortunately makes a

rather long **#define** variable). Actually, to be a tad more precise, I'd call them "monotone with respect to a vertical line and simple," where "simple" means "not self-intersecting." Similarly, the polygon type I called "nonconvex" is actually "simple," and I suppose what I called "complex" should be referred to as "nonsimple," or maybe just "none of the above."

This may seem like nit-picking, but actually, it isn't; what it's really about is the tremendous importance of having a shared language. In one of his books, Richard Feynman describes having developed his own mathematical framework, complete with his own notation and terminology, in high school. When he got to college and started working with other people who were at his level, he suddenly understood that people can't share ideas effectively unless they speak the same language; otherwise, they waste a great deal of time on misunderstandings and explanation.

Or, as Bill Huber put it, "You are free to adopt your own terminology when it suits your purposes well. But you risk losing or confusing those who could be among your most astute readers—those who already have been trained in the same or a related field." Ditto. Likewise. *D'accord.* And *mea culpa*; I shall endeavor to watch my language in the future.

Nomenclature in Action

Just to show you how much difference proper description and interchange of ideas can make, consider the case of identifying convex polygons. When I was writing about polygons in my column in *DDJ*, a nonfunctional method for identifying such polygons—checking for exactly two X direction changes and two Y direction changes around the perimeter of the polygon—crept into the column by accident. That method, as I noted in a later column, does not work. (That's why you won't find it in this book.) Still, a fast method of checking for convex polygons would be highly desirable, because such polygons can be drawn with the fast code from Chapter 39, rather than the relatively slow, general-purpose code from Chapter 40.

Now consider Bill's point that we're not limited to drawing convex polygons in our "convex fill" code, but can actually handle any simple polygon that's monotone with respect to a vertical line. Additionally, consider Anton Treuenfels's point, made back in Chapter 40, that life gets simpler if we stop worrying about which edge of a polygon is the left edge and which is the right, and instead just scan out each raster line starting at whichever edge is left-most. Now, what do we have?

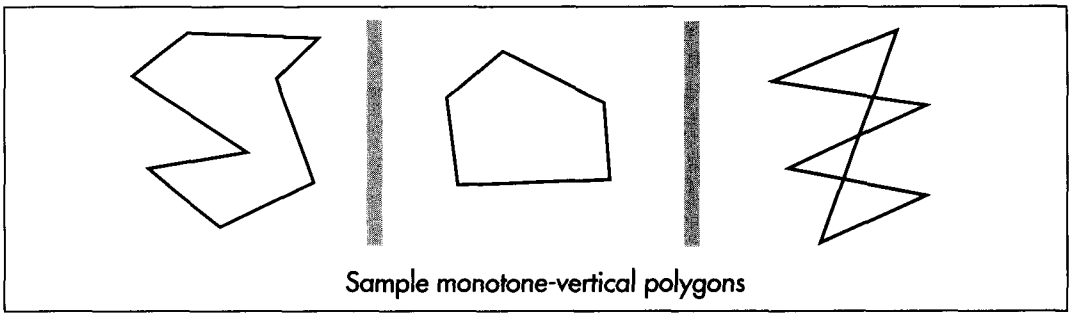
What we have is an approach passed along by Jim Kent, of Autodesk Animator fame. If we modify the low-level code to check which edge is left-most on each scan line and start drawing there, as just described, then we can handle any polygon that's monotone with respect to a vertical line regardless of whether the edges cross. (I'll call this "monotone-vertical" from now on; if anyone wants to correct that terminology, jump right in.) In other words, we can then handle nonsimple polygons that are

monotone-vertical; self-intersection is no longer a problem. We just scan around the polygon's perimeter looking for exactly two direction reversals along the Y axis only, and if that proves to be the case, we can handle the polygon at high speed. Figure 41.1 shows polygons that can be drawn by a monotone-vertical capable filler; Figure 41.2 shows some that cannot. Listing 41.1 shows code to test whether a polygon is appropriately monotone.

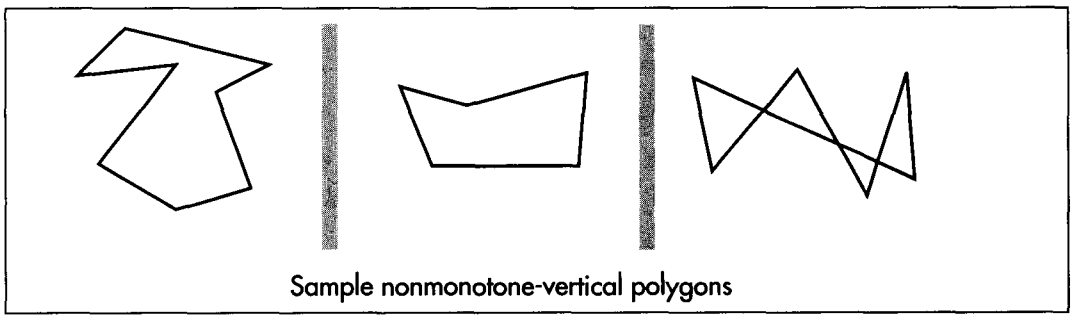
LISTING 41.1 L41-1.C

```
/* Returns 1 if polygon described by passed-in vertex list is monotone with
respect to a vertical line, 0 otherwise. Doesn't matter if polygon is simple
(non-self-intersecting) or not. Tested with Borland C++ in small model. */

#include "polygon.h"

#define SIGNUM(a) ((a>0)?1:((a<0)?-1:0))

int PolygonIsMonotoneVertical(struct PointListHeader * VertexList)
{
   int i, Length, DeltaYSign, PreviousDeltaYSign;
   int NumYReversals = 0;
   struct Point *VertexPtr = VertexList->PointPtr;

   /* Three or fewer points can't make a non-vertical-monotone polygon */
   if ((Length=VertexList->Length) < 4) return(1);

   /* Scan to the first non-horizontal edge */
   PreviousDeltaYSign = SIGNUM(VertexPtr[Length-1].Y - VertexPtr[0].Y);
   i = 0;
   while ((PreviousDeltaYSign == 0) && (i < (Length-1))) {
      PreviousDeltaYSign = SIGNUM(VertexPtr[i].Y - VertexPtr[i+1].Y);
      i++;
   }

   if (i == (Length-1)) return(1);  /* polygon is a flat line */

   /* Now count Y reversals. Might miss one reversal, at the last vertex, but
      because reversal counts must be even, being off by one isn't a problem */
   do {
      if ((DeltaYSign = SIGNUM(VertexPtr[i].Y - VertexPtr[i+1].Y))
            != 0) {
         if (DeltaYSign != PreviousDeltaYSign) {
            /* Switched Y direction; not vertical-monotone if
               reversed Y direction as many as three times */
            if (++NumYReversals > 2) return(0);
            PreviousDeltaYSign = DeltaYSign;
         }
      }
   } while (i++ < (Length-1));
   return(1);  /* it's a vertical-monotone polygon */
}
```

Listings 41.2 and 41.3 are variants of the fast convex polygon fill code from Chapter 39, modified to be able to handle all monotone-vertical polygons, including nonsimple ones; the edge-scanning code (Listing 39.4 from Chapter 39) remains the same, and so is not shown again here.



Monotone-vertical polygons.
Figure 41.1



Non-monotone-vertical polygons.
Figure 41.2

LISTING 41.2 L41-2.C

```
/* Color-fills a convex polygon. All vertices are offset by (XOffset, YOffset).
"Convex" means "monotone with respect to a vertical line"; that is, every
horizontal line drawn through the polygon at any point would cross exactly two
active edges (neither horizontal lines nor zero-length edges count as active
edges; both are acceptable anywhere in the polygon). Right & left edges may
cross (polygons may be nonsimple). Polygons that are not convex according to
this definition won't be drawn properly. (Yes, "convex" is a lousy name for
this type of polygon, but it's convenient; use "monotone-vertical" if it makes
you happier!)
*********************************************************************
NOTE: the low-level drawing routine, DrawHorizontalLineList, must be able to
reverse the edges, if necessary to make the correct edge left edge. It must
also expect right edge to be specified in +1 format (the X coordinate is 1 past
highest coordinate to draw). In both respects, this differs from low-level
drawing routines presented in earlier columns; changes are necessary to make it
possible to draw nonsimple monotone-vertical polygons; that in turn makes it
possible to use Jim Kent's test for monotone-vertical polygons.
*********************************************************************
Returns 1 for success, 0 if memory allocation failed */
```

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

/* Advances the index by one vertex forward through the vertex list,
wrapping at the end of the list */
#define INDEX_FORWARD(Index) \
   Index = (Index + 1) % VertexList->Length;

/* Advances the index by one vertex backward through the vertex list,
wrapping at the start of the list */
#define INDEX_BACKWARD(Index) \
   Index = (Index - 1 + VertexList->Length) % VertexList->Length;

/* Advances the index by one vertex either forward or backward through
the vertex list, wrapping at either end of the list */
#define INDEX_MOVE(Index,Direction)                                    \
   if (Direction > 0)                                                  \
      Index = (Index + 1) % VertexList->Length;                        \
   else                                                                \
      Index = (Index - 1 + VertexList->Length) % VertexList->Length;

extern void ScanEdge(int, int, int, int, int, int, struct HLine **);
extern void DrawHorizontalLineList(struct HLineList *, int);

int FillMonotoneVerticalPolygon(struct PointListHeader * VertexList,
      int Color, int XOffset, int YOffset)
{
   int i, MinIndex, MaxIndex, MinPoint_Y, MaxPoint_Y;
   int NextIndex, CurrentIndex, PreviousIndex;
   struct HLineList WorkingHLineList;
   struct HLine *EdgePointPtr;
   struct Point *VertexPtr;

   /* Point to the vertex list */
   VertexPtr = VertexList->PointPtr;

   /* Scan the list to find the top and bottom of the polygon */
   if (VertexList->Length == 0)
      return(1);  /* reject null polygons */
   MaxPoint_Y = MinPoint_Y = VertexPtr[MinIndex = MaxIndex = 0].Y;
   for (i = 1; i < VertexList->Length; i++) {
      if (VertexPtr[i].Y < MinPoint_Y)
         MinPoint_Y = VertexPtr[MinIndex = i].Y; /* new top */
      else if (VertexPtr[i].Y > MaxPoint_Y)
         MaxPoint_Y = VertexPtr[MaxIndex = i].Y; /* new bottom */
   }

   /* Set the # of scan lines in the polygon, skipping the bottom edge */
   if ((WorkingHLineList.Length = MaxPoint_Y - MinPoint_Y) <= 0)
      return(1);  /* there's nothing to draw, so we're done */
   WorkingHLineList.YStart = YOffset + MinPoint_Y;

   /* Get memory in which to store the line list we generate */
   if ((WorkingHLineList.HLinePtr =
         (struct HLine *) (malloc(sizeof(struct HLine) *
         WorkingHLineList.Length))) == NULL)
      return(0);  /* couldn't get memory for the line list */
```

```
   /* Scan the first edge and store the boundary points in the list */
   /* Initial pointer for storing scan converted first-edge coords */
   EdgePointPtr = WorkingHLineList.HLinePtr;
   /* Start from the top of the first edge */
   PreviousIndex = CurrentIndex = MinIndex;
   /* Scan convert each line in the first edge from top to bottom */
   do {
      INDEX_BACKWARD(CurrentIndex);
      ScanEdge(VertexPtr[PreviousIndex].X + XOffset,
            VertexPtr[PreviousIndex].Y,
            VertexPtr[CurrentIndex].X + XOffset,
            VertexPtr[CurrentIndex].Y, 1, 0, &EdgePointPtr);
      PreviousIndex = CurrentIndex;
   } while (CurrentIndex != MaxIndex);

   /* Scan the second edge and store the boundary points in the list */
   EdgePointPtr = WorkingHLineList.HLinePtr;
   PreviousIndex = CurrentIndex = MinIndex;
   /* Scan convert the second edge, top to bottom */
   do {
      INDEX_FORWARD(CurrentIndex);
      ScanEdge(VertexPtr[PreviousIndex].X + XOffset,
            VertexPtr[PreviousIndex].Y,
            VertexPtr[CurrentIndex].X + XOffset,
            VertexPtr[CurrentIndex].Y, 0, 0, &EdgePointPtr);
      PreviousIndex = CurrentIndex;
   } while (CurrentIndex != MaxIndex);

   /* Draw the line list representing the scan converted polygon */
   DrawHorizontalLineList(&WorkingHLineList, Color);

   /* Release the line list's memory and we're successfully done */
   free(WorkingHLineList.HLinePtr);
   return(1);
}
```

LISTING 41.3 L41-3.ASM

```
; Draws all pixels in list of horizontal lines passed in, in mode 13h, VGA's
; 320x200 256-color mode. Uses REP STOS to fill each line.
; ****************************************************************
; NOTE: is able to reverse the X coords for a scan line, if necessary, to make
; XStart < XEnd. Expects whichever edge is rightmost on any scan line to be in
; +1 format; that is, XEnd is 1 greater than rightmost pixel to draw. If
; XStart == XEnd, nothing is drawn on that scan 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
        cmp     di,cx           ;is XStart > XEnd?
        jle     NoSwap          ;no, we're all set
        xchg    di,cx           ;yes, so swap edges
NoSwap:
        sub     cx,di           ;width of fill on this line
        jz      LineFillDone    ;skip if zero width
        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
```

Listing 41.4 is almost identical to Listing 40.1 from Chapter 40. I've modified Listing 40.1 to employ the vertical-monotone detection test we've been talking about and use the fast vertical-monotone drawing code whenever possible; that's what Listing 41.4 is. Note well that Listing 40.5 from Chapter 40 is also required in order for this code to link. Listing 41.5 is an appropriately updated version of the POLYGON.H header file.

LISTING 41.4 L41-4.C

```
/* Color-fills an arbitrarily-shaped polygon described by VertexList.
If the first and last points in VertexList are not the same, the path
around the polygon is automatically closed. All vertices are offset
by (XOffset, YOffset). Returns 1 for success, 0 if memory allocation
failed. All C code tested with Borland C++.

If the polygon shape is known in advance, speedier processing may be
enabled by specifying the shape as follows: "convex" - a rubber band
stretched around the polygon would touch every vertex in order;
"nonconvex" - the polygon is not self-intersecting, but need not be
convex; "complex" - the polygon may be self-intersecting, or, indeed,
any sort of polygon at all. Complex will work for all polygons; convex
is fastest. Undefined results will occur if convex is specified for a
nonconvex or complex polygon.

Define CONVEX_CODE_LINKED if the fast convex polygon filling code from
the February 1991 column is linked in. Otherwise, convex polygons are
handled by the complex polygon filling code.
Nonconvex is handled as complex in this implementation. See text for a
discussion of faster nonconvex handling. */

#include <stdio.h>
#include <math.h>
#ifdef __TURBOC__
#include <alloc.h>
#else    /* MSC */
#include <malloc.h>
#endif
#include "polygon.h"

#define SWAP(a,b) {temp = a; a = b; b = temp;}

struct EdgeState {
   struct EdgeState *NextEdge;
   int X;
   int StartY;
   int WholePixelXMove;
   int XDirection;
   int ErrorTerm;
   int ErrorTermAdjUp;
   int ErrorTermAdjDown;
   int Count;
};
```

```
extern void DrawHorizontalLineSeg(int, int, int, int);
extern int FillMonotoneVerticalPolygon(struct PointListHeader *,
   int, int, int);
extern int PolygonIsMonotoneVertical(struct PointListHeader *);
static void BuildGET(struct PointListHeader *, struct EdgeState *,
   int, int);
static void MoveXSortedToAET(int);
static void ScanOutAET(int, int);
static void AdvanceAET(void);
static void XSortAET(void);

/* Pointers to global edge table (GET) and active edge table (AET) */
static struct EdgeState *GETPtr, *AETPtr;

int FillPolygon(struct PointListHeader * VertexList, int Color,
      int PolygonShape, int XOffset, int YOffset)
{
   struct EdgeState *EdgeTableBuffer;
   int CurrentY;


#ifdef CONVEX_CODE_LINKED
   /* Pass convex polygons through to fast convex polygon filler */
   if ((PolygonShape == CONVEX) ||
         PolygonIsMonotoneVertical(VertexList))
      return(FillMonotoneVerticalPolygon(VertexList, Color, XOffset,
            YOffset));
#endif

   /* It takes a minimum of 3 vertices to cause any pixels to be
      drawn; reject polygons that are guaranteed to be invisible */
   if (VertexList->Length < 3)
      return(1);
   /* Get enough memory to store the entire edge table */
   if ((EdgeTableBuffer =
         (struct EdgeState *) (malloc(sizeof(struct EdgeState) *
         VertexList->Length))) == NULL)
      return(0);  /* couldn't get memory for the edge table */
   /* Build the global edge table */
   BuildGET(VertexList, EdgeTableBuffer, XOffset, YOffset);
   /* Scan down through the polygon edges, one scan line at a time,
      so long as at least one edge remains in either the GET or AET */
   AETPtr = NULL;    /* initialize the active edge table to empty */
   CurrentY = GETPtr->StartY; /* start at the top polygon vertex */
   while ((GETPtr != NULL) || (AETPtr != NULL)) {
      MoveXSortedToAET(CurrentY);  /* update AET for this scan line */
      ScanOutAET(CurrentY, Color); /* draw this scan line from AET */
      AdvanceAET();                /* advance AET edges 1 scan line */
      XSortAET();                  /* resort on X */
      CurrentY++;                  /* advance to the next scan line */
   }
   /* Release the memory we've allocated and we're done */
   free(EdgeTableBuffer);
   return(1);
}

/* Creates a GET in the buffer pointed to by NextFreeEdgeStruc from
   the vertex list. Edge endpoints are flipped, if necessary, to
   guarantee all edges go top to bottom. The GET is sorted primarily
   by ascending Y start coordinate, and secondarily by ascending X
   start coordinate within edges with common Y coordinates. */
```

```
static void BuildGET(struct PointListHeader * VertexList,
      struct EdgeState * NextFreeEdgeStruc, int XOffset, int YOffset)
{
   int i, StartX, StartY, EndX, EndY, DeltaY, DeltaX, Width, temp;
   struct EdgeState *NewEdgePtr;
   struct EdgeState *FollowingEdge, **FollowingEdgeLink;
   struct Point *VertexPtr;

   /* Scan through the vertex list and put all non-0-height edges into
      the GET, sorted by increasing Y start coordinate */
   VertexPtr = VertexList->PointPtr;   /* point to the vertex list */
   GETPtr = NULL;    /* initialize the global edge table to empty */
   for (i = 0; i < VertexList->Length; i++) {
      /* Calculate the edge height and width */
      StartX = VertexPtr[i].X + XOffset;
      StartY = VertexPtr[i].Y + YOffset;
      /* The edge runs from the current point to the previous one */
      if (i == 0) {
         /* Wrap back around to the end of the list */
         EndX = VertexPtr[VertexList->Length-1].X + XOffset;
         EndY = VertexPtr[VertexList->Length-1].Y + YOffset;
      } else {
         EndX = VertexPtr[i-1].X + XOffset;
         EndY = VertexPtr[i-1].Y + YOffset;
      }
      /* Make sure the edge runs top to bottom */
      if (StartY > EndY) {
         SWAP(StartX, EndX);
         SWAP(StartY, EndY);
      }
      /* Skip if this can't ever be an active edge (has 0 height) */
      if ((DeltaY = EndY - StartY) != 0) {
         /* Allocate space for this edge's info, and fill in the
            structure */
         NewEdgePtr = NextFreeEdgeStruc++;
         NewEdgePtr->XDirection =   /* direction in which X moves */
               ((DeltaX = EndX - StartX) > 0) ? 1 : -1;
         Width = abs(DeltaX);
         NewEdgePtr->X = StartX;
         NewEdgePtr->StartY = StartY;
         NewEdgePtr->Count = DeltaY;
         NewEdgePtr->ErrorTermAdjDown = DeltaY;
         if (DeltaX >= 0)  /* initial error term going L->R */
            NewEdgePtr->ErrorTerm = 0;
         else              /* initial error term going R->L */
            NewEdgePtr->ErrorTerm = -DeltaY + 1;
         if (DeltaY >= Width) {     /* Y-major edge */
            NewEdgePtr->WholePixelXMove = 0;
            NewEdgePtr->ErrorTermAdjUp = Width;
         } else {                   /* X-major edge */
            NewEdgePtr->WholePixelXMove =
                  (Width / DeltaY) * NewEdgePtr->XDirection;
            NewEdgePtr->ErrorTermAdjUp = Width % DeltaY;
         }
         /* Link the new edge into the GET so that the edge list is
            still sorted by Y coordinate, and by X coordinate for all
            edges with the same Y coordinate */
         FollowingEdgeLink = &GETPtr;
         for (;;) {
            FollowingEdge = *FollowingEdgeLink;
```

```
            if ((FollowingEdge == NULL) ||
                  (FollowingEdge->StartY > StartY) ||
                  ((FollowingEdge->StartY == StartY) &&
                  (FollowingEdge->X >= StartX))) {
               NewEdgePtr->NextEdge = FollowingEdge;
               *FollowingEdgeLink = NewEdgePtr;
               break;
            }
            FollowingEdgeLink = &FollowingEdge->NextEdge;
         }
      }
   }
}

/* Sorts all edges currently in the active edge table into ascending
   order of current X coordinates */
static void XSortAET() {
   struct EdgeState *CurrentEdge, **CurrentEdgePtr, *TempEdge;
   int SwapOccurred;

   /* Scan through the AET and swap any adjacent edges for which the
      second edge is at a lower current X coord than the first edge.
      Repeat until no further swapping is needed */
   if (AETPtr != NULL) {
      do {
         SwapOccurred = 0;
         CurrentEdgePtr = &AETPtr;
         while ((CurrentEdge = *CurrentEdgePtr)->NextEdge != NULL) {
            if (CurrentEdge->X > CurrentEdge->NextEdge->X) {
               /* The second edge has a lower X than the first;
                  swap them in the AET */
               TempEdge = CurrentEdge->NextEdge->NextEdge;
               *CurrentEdgePtr = CurrentEdge->NextEdge;
               CurrentEdge->NextEdge->NextEdge = CurrentEdge;
               CurrentEdge->NextEdge = TempEdge;
               SwapOccurred = 1;
            }
            CurrentEdgePtr = &(*CurrentEdgePtr)->NextEdge;
         }
      } while (SwapOccurred != 0);
   }
}

/* Advances each edge in the AET by one scan line.
   Removes edges that have been fully scanned. */
static void AdvanceAET() {
   struct EdgeState *CurrentEdge, **CurrentEdgePtr;

   /* Count down and remove or advance each edge in the AET */
   CurrentEdgePtr = &AETPtr;
   while ((CurrentEdge = *CurrentEdgePtr) != NULL) {
      /* Count off one scan line for this edge */
      if ((--(CurrentEdge->Count)) == 0) {
         /* This edge is finished, so remove it from the AET */
         *CurrentEdgePtr = CurrentEdge->NextEdge;
      } else {
         /* Advance the edge's X coordinate by minimum move */
         CurrentEdge->X += CurrentEdge->WholePixelXMove;
         /* Determine whether it's time for X to advance one extra */
         if ((CurrentEdge->ErrorTerm +=
               CurrentEdge->ErrorTermAdjUp) > 0) {
```

```
            CurrentEdge->X += CurrentEdge->XDirection;
            CurrentEdge->ErrorTerm -= CurrentEdge->ErrorTermAdjDown;
         }
         CurrentEdgePtr = &CurrentEdge->NextEdge;
      }
   }
}

/* Moves all edges that start at the specified Y coordinate from the
   GET to the AET, maintaining the X sorting of the AET. */
static void MoveXSortedToAET(int YToMove) {
   struct EdgeState *AETEdge, **AETEdgePtr, *TempEdge;
   int CurrentX;

   /* The GET is Y sorted. Any edges that start at the desired Y
      coordinate will be first in the GET, so we'll move edges from
      the GET to AET until the first edge left in the GET is no longer
      at the desired Y coordinate. Also, the GET is X sorted within
      each Y coordinate, so each successive edge we add to the AET is
      guaranteed to belong later in the AET than the one just added. */
   AETEdgePtr = &AETPtr;
   while ((GETPtr != NULL) && (GETPtr->StartY == YToMove)) {
      CurrentX = GETPtr->X;
      /* Link the new edge into the AET so that the AET is still
         sorted by X coordinate */
      for (;;) {
         AETEdge = *AETEdgePtr;
         if ((AETEdge == NULL) || (AETEdge->X >= CurrentX)) {
            TempEdge = GETPtr->NextEdge;
            *AETEdgePtr = GETPtr;  /* link the edge into the AET */
            GETPtr->NextEdge = AETEdge;
            AETEdgePtr = &GETPtr->NextEdge;
            GETPtr = TempEdge;   /* unlink the edge from the GET */
            break;
         } else {
            AETEdgePtr = &AETEdge->NextEdge;
         }
      }
   }
}

/* Fills the scan line described by the current AET at the specified Y
   coordinate in the specified color, using the odd/even fill rule */
static void ScanOutAET(int YToScan, int Color) {
   int LeftX;
   struct EdgeState *CurrentEdge;

   /* Scan through the AET, drawing line segments as each pair of edge
      crossings is encountered. The nearest pixel on or to the right
      of left edges is drawn, and the nearest pixel to the left of but
      not on right edges is drawn */
   CurrentEdge = AETPtr;
   while (CurrentEdge != NULL) {
      LeftX = CurrentEdge->X;
      CurrentEdge = CurrentEdge->NextEdge;
      DrawHorizontalLineSeg(YToScan, LeftX, CurrentEdge->X-1, Color);
      CurrentEdge = CurrentEdge->NextEdge;
   }
}
```

LISTING 41.5 POLYGON.H

```
/* Header file for polygon-filling code */

#define CONVEX    0
#define NONCONVEX 1
#define COMPLEX   2

/* Describes a single point (used for a single vertex) */
struct Point {
   int X;   /* X coordinate */
   int Y;   /* Y coordinate */
};

/* Describes series of points (used to store a list of vertices that describe
a polygon; each vertex is assumed to connect to the two adjacent vertices, and
last vertex is assumed to connect to the first) */
struct PointListHeader {
   int Length;               /* # of points */
   struct Point * PointPtr;  /* pointer to list of points */
};

/* Describes beginning and ending X coordinates of a single horizontal line */
struct HLine {
   int XStart; /* X coordinate of leftmost pixel in line */
   int XEnd;   /* X coordinate of rightmost pixel in line */
};

/* Describes a Length-long series of horizontal lines, all assumed to be on
contiguous scan lines starting at YStart and proceeding downward (used to
describe scan-converted polygon to low-level hardware-dependent drawing code) */
struct HLineList {
   int Length;               /* # of horizontal lines */
   int YStart;               /* Y coordinate of topmost line */
   struct HLine * HLinePtr;  /* pointer to list of horz lines */
};

/* Describes a color as an RGB triple, plus one byte for other info */
struct RGB { unsigned char Red, Green, Blue, Spare; };
```

Is monotone-vertical polygon detection worth all this trouble? Under the right circumstances, you bet. In a situation where a great many polygons are being drawn, and the application either doesn't know whether they're monotone-vertical or has no way to tell the polygon filler that they are, performance can be increased considerably if most polygons are, in fact, monotone-vertical. This potential performance advantage is helped along by the surprising fact that Jim's test for monotone-vertical status is simpler and faster than my original, nonfunctional test for convexity.

See what accurate terminology and effective communication can do?