Chapter 40

Of Songs, Taxes, and the Simplicity of Complex Polygons

Chapter 40

Dealing with Irregular Polygonal Areas

Every so often, my daughter asks me to sing her to sleep. (If you've ever heard me sing, this may cause you concern about either her hearing or her judgement, but love knows no bounds.) As any parent is well aware, singing a young child to sleep can easily take several hours, or until sunrise, whichever comes last. One night, running low on children's songs, I switched to a Beatles medley, and at long last her breathing became slow and regular. At the end, I softly sang "A Hard Day's Night," then quietly stood up to leave. As I tiptoed out, she said, in a voice not even faintly tinged with sleep, "Dad, what do they mean, 'working like a dog'? Chasing a stick? That doesn't make sense; people don't chase sticks."

That led us into a discussion of idioms, which made about as much sense to her as an explanation of quantum mechanics. Finally, I fell back on my standard explanation of the Universe, which is that a lot of the time it simply doesn't make sense.

As a general principle, that explanation holds up remarkably well. (In fact, having just done my taxes, I think Earth is actually run by blob-creatures from the planet Mrxx, who are helplessly doubled over with laughter at the ridiculous things they can make us do. "Let's make them get Social Security numbers for their pets next year!" they're saying right now, gasping for breath.) Occasionally, however, one has the rare pleasure of finding a corner of the Universe that makes sense, where everything fits together as if preordained.

Filling arbitrary polygons is such a case.

Filling Arbitrary Polygons

In Chapter 38, I described three types of polygons: convex, nonconvex, and complex. *The RenderMan Companion*, a terrific book by Steve Upstill (Addison-Wesley, 1990) has an intuitive definition of *convex*: If a rubber band stretched around a polygon touches all vertices in the order they're defined, then the polygon is convex. If a polygon has intersecting edges, it's complex. If a polygon doesn't have intersecting edges but isn't convex, it's nonconvex. Nonconvex is a special case of complex, and convex is a special case of nonconvex. (Which, I'm well aware, makes nonconvex a lousy name—noncomplex would have been better—but I'm following X Window System nomenclature here.)

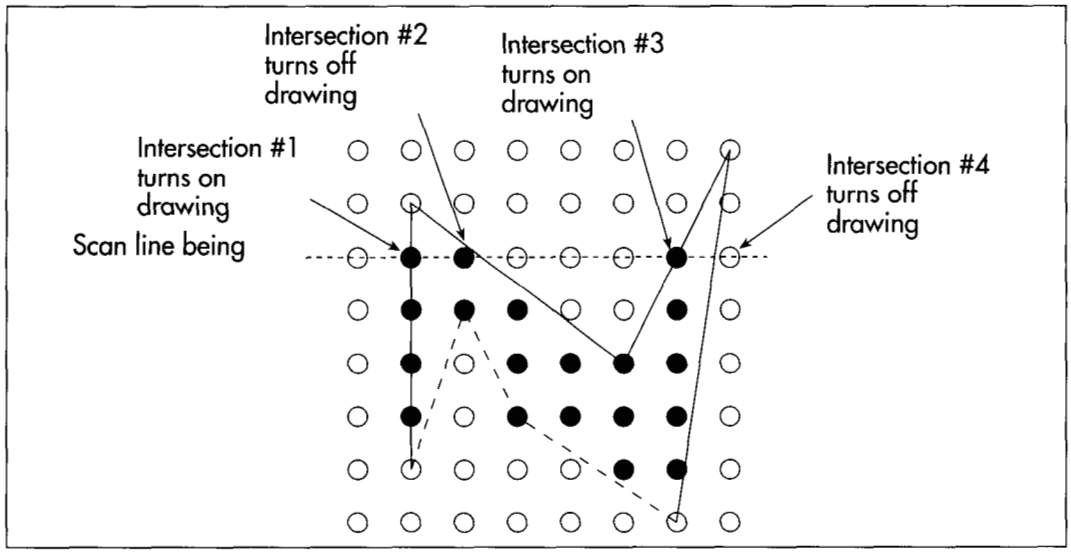
The reason for distinguishing between these three types of polygons is that the more specialized types can be filled with markedly faster approaches. Complex polygons require the slowest approach; however, that approach will serve to fill any polygon of any sort. Nonconvex polygons require less sorting, because edges never cross. Convex polygons can be filled fastest of all by simply scanning the two sides of the polygon, as we saw in Chapter 39.

Before we dive into complex polygon filling, I'd like to point out that the code in this chapter, like all polygon filling code I've ever seen, requires that the caller describe the type of the polygon to be filled. Often, however, the caller doesn't know what type of polygon it's passing, or specifies complex for simplicity, because that will work for all polygons; in such a case, the polygon filler will use the slow complex-fill code even if the polygon is, in fact, a convex polygon. In Chapter 41, I'll discuss one way to improve this situation.

Active Edges

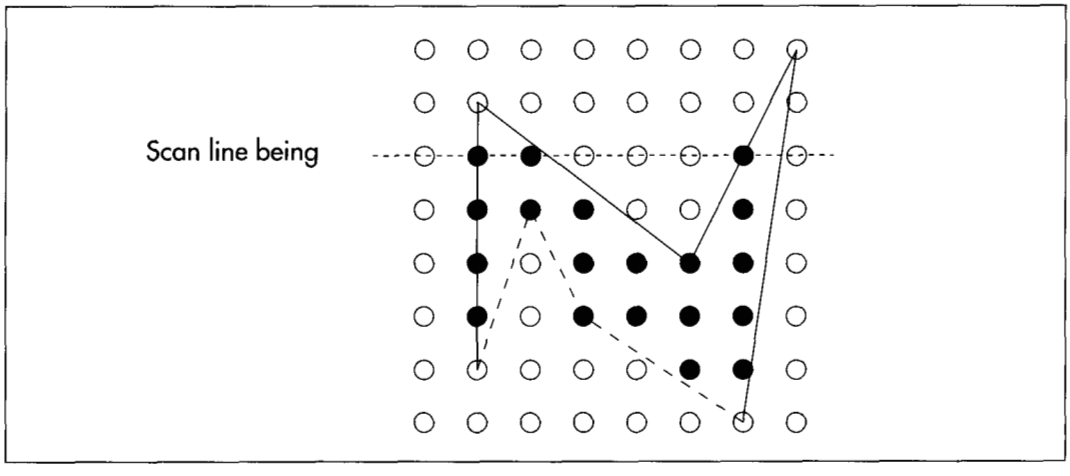
The basic premise of filling a complex polygon is that for a given scan line, we determine all intersections between the polygon's edges and that scan line and then fill the spans between the intersections, as shown in Figure 40.1. (Section 3.6 of Foley and van Dam's *Computer Graphics*, Second Edition provides an overview of this and other aspects of polygon filling.) There are several rules that might be used to determine which spans are drawn and which aren't; we'll use the odd/even rule, which specifies that drawing turns on after odd-numbered intersections (first, third, and so on) and off after even-numbered intersections.

The question then becomes how can we most efficiently determine which edges cross each scan line and where? As it happens, there is a great deal of coherence from one scan line to the next in a polygon edge list, because each edge starts at a given Y coordinate and continues unbroken until it ends. In other words, edges don't leap about and stop and start randomly; the X coordinate of an edge at one scan line is a consistent delta from that edge's X coordinate at the last scan line, and that is consistent for the length of the line.



Filling one scan line by finding intersecting edges.
Figure 40.1

This allows us to reduce the number of edges that must be checked for intersection; on any given scan line, we only need to check for intersections with the currently active edges—edges that start on that scan line, plus all edges that start on earlier (above) scan lines and haven't ended yet—as shown in Figure 40.2. This suggests that we can proceed from the top scan line of the polygon to the bottom, keeping a



Checking currently active edges (solid lines).
Figure 40.2

running list of currently active edges—called the *active edge table* (AET)—with the edges sorted in order of ascending X coordinate of intersection with the current scan line. Then, we can simply fill each scan line in turn according to the list of active edges at that line.

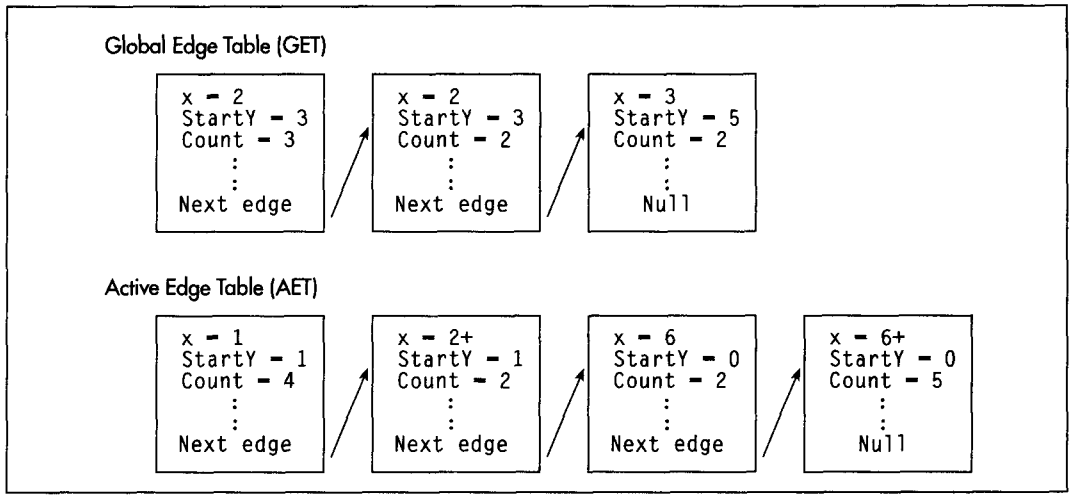
Maintaining the AET from one scan line to the next involves three steps: First, we must add to the AET any edges that start on the current scan line, making sure to keep the AET X-sorted for efficient odd/even scanning. Second, we must remove edges that end on the current scan line. Third, we must advance the X coordinates of active edges with the same sort of error term-based, Bresenham's-like approach we used for convex polygons, again ensuring that the AET is X-sorted after advancing the edges.

Advancing the X coordinates is easy. For each edge, we'll store the current X coordinate and all required error term information, and we'll use that to advance the edge one scan line at a time; then, we'll resort the AET by X coordinate as needed. Removing edges as they end is also easy; we'll just count down the length of each active edge on each scan line and remove an edge when its count reaches zero. Adding edges as their tops are encountered is a tad more complex. While there are a number of ways to do this, one particularly efficient approach is to start out by putting all the edges of the polygon, sorted by increasing Y coordinate, into a single list, called the *global edge table* (GET). Then, as each scan line is encountered, all edges at the start of the GET that begin on the current scan line are moved to the AET; because the GET is Y-sorted, there's no need to search the entire GET. For still greater efficiency, edges in the GET that share common Y coordinates can be sorted by increasing X coordinate; this ensures that no more than one pass through the AET per scan line is ever needed when adding new edges from the GET in such a way as to keep the AET sorted in ascending X order.

What form should the GET and AET take? Linked lists of edge structures, as shown in Figure 40.3. With linked lists, all that's required to move edges from the GET to the AET as they become active, sort the AET, and remove edges that have been fully drawn is the exchanging of a few pointers.

In summary, we'll initially store all the polygon edges in Y-primary/X-secondary sort order in the GET, complete with initial X and Y coordinates, error terms and error term adjustments, lengths, and directions of X movement for each edge. Once the GET is built, we'll do the following:

1. Set the current Y coordinate to the Y coordinate of the first edge in the GET.
2. Move all edges with the current Y coordinate from the GET to the AET, removing them from the GET and maintaining the X-sorted order of the AET.
3. Draw all odd-to-even spans in the AET at the current Y coordinate.
4. Count down the lengths of all edges in the AET, removing any edges that are done, and advancing the X coordinates of all remaining edges in the AET by one scan line.



The global and active edge tables as linked lists.
Figure 40.3

5. Sort the AET in order of ascending X coordinate.
6. Advance the current Y coordinate by one scan line.
7. If either the AET or GET isn't empty, go to step 2.

That's really all there is to it. Compare Listing 40.1 to the fast convex polygon filling code from Chapter 39, and you'll see that, contrary to expectation, complex polygon filling is indeed one of the more sane and sensible corners of the universe.

LISTING 40.1 L40-1.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
   Chapter 38 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 FillConvexPolygon(struct PointListHeader *, int, int, int);
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)
      return(FillConvexPolygon(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;
   }
}
```

Complex Polygon Filling: An Implementation

Listing 40.1 just shown presents a function, **FillPolygon()**, that fills polygons of all shapes. If **CONVEX_FILL_LINKED** is defined, the fast convex fill code from Chapter 39 is linked in and used to draw convex polygons. Otherwise, convex polygons are handled as if they were complex. Nonconvex polygons are also handled as complex, although this is not necessary, as discussed shortly.

Listing 40.1 is a faithful implementation of the complex polygon filling approach just described, with separate functions corresponding to each of the tasks, such as building the GET and X-sorting the AET. Listing 40.2 provides the actual drawing code used to fill spans, built on a draw pixel routine that is the only hardware dependency anywhere in the C code. Listing 40.3 is the header file for the polygon filling code; note that it is an expanded version of the header file used by the fast convex polygon fill code from Chapter 39. (They may have the same name but are *not* the same file!) Listing 40.4 is a sample program that, when linked to Listings 40.1 and 40.2, demonstrates drawing polygons of various sorts.

LISTING 40.2 L40-2.C

```
/* Draws all pixels in the horizontal line segment passed in, from
   (LeftX,Y) to (RightX,Y), in the specified color in mode 13h, the
   VGA's 320x200 256-color mode. Both LeftX and RightX are drawn. No
   drawing will take place if LeftX > RightX. */

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

#define SCREEN_WIDTH    320
#define SCREEN_SEGMENT  0xA000

static void DrawPixel(int, int, int);

void DrawHorizontalLineSeg(Y, LeftX, RightX, Color) {
   int X;

   /* Draw each pixel in the horizontal line segment, starting with
      the leftmost one */
   for (X = LeftX; X <= RightX; X++)
      DrawPixel(X, Y, Color);
}
```

```
/* Draws the pixel at (X, Y) in color Color in VGA mode 13h */
static void DrawPixel(int X, int Y, int Color) {
   unsigned char far *ScreenPtr;

#ifdef __TURBOC__
   ScreenPtr = MK_FP(SCREEN_SEGMENT, Y * SCREEN_WIDTH + X);
#else    /* MSC 5.0 */
   FP_SEG(ScreenPtr) = SCREEN_SEGMENT;
   FP_OFF(ScreenPtr) = Y * SCREEN_WIDTH + X;
#endif
   *ScreenPtr = (unsigned char) Color;
}
```

LISTING 40.3 POLYGON.H

```
/* 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 a series of points (used to store a list of vertices that
   describe a polygon; each vertex connects to the two adjacent
   vertices; the last vertex is assumed to connect to the first) */
struct PointListHeader {
   int Length;               /* # of points */
   struct Point * PointPtr;  /* pointer to list of points */
};
/* Describes the beginning and ending X coordinates of a single
   horizontal line (used only by fast polygon fill code) */
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 a scan-converted polygon to the
   low-level hardware-dependent drawing code) (used only by fast
   polygon fill code). */
struct HLineList {
   int Length;               /* # of horizontal lines */
   int YStart;               /* Y coordinate of topmost line */
   struct HLine * HLinePtr;  /* pointer to list of horz lines */
};
```

LISTING 40.4 L40-4.C

```
/* Sample program to exercise the polygon-filling routines */

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

#define DRAW_POLYGON(PointList,Color,Shape,X,Y)            \
   Polygon.Length = sizeof(PointList)/sizeof(struct Point); \
   Polygon.PointPtr = PointList;                           \
   FillPolygon(&Polygon, Color, Shape, X, Y);
```

```
void main(void);
extern int FillPolygon(struct PointListHeader *, int, int, int, int);

void main() {
   int i, j;
   struct PointListHeader Polygon;
   static struct Point Polygon1[] =
         {{0,0},{100,150},{320,0},{0,200},{220,50},{320,200}};
   static struct Point Polygon2[] =
         {{0,0},{320,0},{320,200},{0,200},{0,0},{50,50},
          {270,50},{270,150},{50,150},{50,50}};
   static struct Point Polygon3[] =
         {{0,0},{10,0},{105,185},{260,30},{15,150},{5,150},{5,140},
          {260,5},{300,5},{300,15},{110,200},{100,200},{0,10}};
   static struct Point Polygon4[] =
         {{0,0},{30,-20},{30,0},{0,20},{-30,0},{-30,-20}};
   static struct Point Triangle1[] = {{30,0},{15,20},{0,0}};
   static struct Point Triangle2[] = {{30,20},{15,0},{0,20}};
   static struct Point Triangle3[] = {{0,20},{20,10},{0,0}};
   static struct Point Triangle4[] = {{20,20},{20,0},{0,10}};
   union REGS regset;

   /* Set the display to VGA mode 13h, 320x200 256-color mode */
   regset.x.ax = 0x0013;
   int86(0x10, &regset, &regset);

   /* Draw three complex polygons */
   DRAW_POLYGON(Polygon1, 15, COMPLEX, 0, 0);
   getch();    /* wait for a keypress */
   DRAW_POLYGON(Polygon2, 5, COMPLEX, 0, 0);
   getch();    /* wait for a keypress */
   DRAW_POLYGON(Polygon3, 3, COMPLEX, 0, 0);
   getch();    /* wait for a keypress */

   /* Draw some adjacent nonconvex polygons */
   for (i=0; i<5; i++) {
      for (j=0; j<8; j++) {
         DRAW_POLYGON(Polygon4, 16+i*8+j, NONCONVEX, 40+(i*60),
               30+(j*20));
      }
   }
   getch();    /* wait for a keypress */

   /* Draw adjacent triangles across the screen */
   for (j=0; j<=80; j+=20) {
      for (i=0; i<290; i += 30) {
         DRAW_POLYGON(Triangle1, 2, CONVEX, i, j);
         DRAW_POLYGON(Triangle2, 4, CONVEX, i+15, j);
      }
   }
   for (j=100; j<=170; j+=20) {
      /* Do a row of pointing-right triangles */
      for (i=0; i<290; i += 20) {
         DRAW_POLYGON(Triangle3, 40, CONVEX, i, j);
      }
      /* Do a row of pointing-left triangles halfway between one row
         of pointing-right triangles and the next, to fit between */
      for (i=0; i<290; i += 20) {
         DRAW_POLYGON(Triangle4, 1, CONVEX, i, j+10);
      }
```

```
    }
    getch();    /* wait for a keypress */

    /* Return to text mode and exit */
    regset.x.ax = 0x0003;
    int86(0x10, &regset, &regset);
}
```

Listing 40.4 illustrates several interesting aspects of polygon filling. The first and third polygons drawn illustrate the operation of the odd/even fill rule. The second polygon drawn illustrates how holes can be created in seemingly solid objects; an edge runs from the outside of the rectangle to the inside, the edges comprising the hole are defined, and then the same edge is used to move back to the outside; because the edges join seamlessly, the rectangle appears to form a solid boundary around the hole.

The set of V-shaped polygons drawn by Listing 40.4 demonstrate that polygons sharing common edges meet but do not overlap. This characteristic, which I discussed at length in Chapter 38, is not a trivial matter; it allows polygons to fit together without fear of overlapping or missed pixels. In general, Listing 40.1 guarantees that polygons are filled such that common boundaries and vertices are drawn once and only once. This has the side-effect for any individual polygon of not drawing pixels that lie exactly on the bottom or right boundaries or at vertices that terminate bottom or right boundaries.

By the way, I have not seen polygon boundary filling handled precisely this way elsewhere. The boundary filling approach in Foley and van Dam is similar, but seems to me to not draw all boundary and vertex pixels once and only once.

More on Active Edges

Edges of zero height—horizontal edges and edges defined by two vertices at the same location—never even make it into the GET in Listing 40.1. A polygon edge of zero height can never be an active edge, because it can never intersect a scan line; it can only run along the scan line, and the span it runs along is defined not by that edge but by the edges that connect to its endpoints.

Performance Considerations

How fast is Listing 40.1? When drawing triangles on a 20-MHz 386, it's less than one-fifth the speed of the fast convex polygon fill code. However, most of that time is spent drawing individual pixels; when Listing 40.2 is replaced with the fast assembly line segment drawing code in Listing 40.5, performance improves by two and one-half times, to about half as fast as the fast convex fill code. Even after conversion to assembly in Listing 40.5, **DrawHorizontalLineSeg** still takes more than half of the total execution time, and the remaining time is spread out fairly evenly over the various subroutines in Listing 40.1. Consequently, there's no single place in which it's possible to greatly improve performance, and the maximum additional improvement

that's possible looks to be a good deal less than two times; for that reason, and because of space limitations, I'm not going to convert the rest of the code to assembly. However, when filling a polygon with a great many edges, and especially one with a great many active edges at one time, relatively more time would be spent traversing the linked lists. In such a case, conversion to assembly (which does a very good job with linked list processing) could pay off reasonably well.

LISTING 40.5 L40-5.ASM

```
; Draws all pixels in the horizontal line segment passed in, from
;  (LeftX,Y) to (RightX,Y), in the specified color in mode 13h, the
;  VGA's 320x200 256-color mode. No drawing will take place if
;  LeftX > RightX. Tested with TASM
; C near-callable as:
;       void DrawHorizontalLineSeg(Y, LeftX, RightX, Color);

SCREEN_WIDTH    equ     320
SCREEN_SEGMENT  equ     0a000h

Parms   struc
        dw      2 dup(?)          ;return address & pushed BP
Y       dw      ?                 ;Y coordinate of line segment to draw
LeftX   dw      ?                 ;left endpoint of the line segment
RightX  dw      ?                 ;right endpoint of the line segment
Color   dw      ?                 ;color in which to draw the line segment
Parms   ends

        .model small
        .code
        public _DrawHorizontalLineSeg
        align   2
_DrawHorizontalLineSeg  proc
        push    bp                ;preserve caller's stack frame
        mov     bp,sp             ;point to our stack frame
        push    di                ;preserve caller's register variable
        cld                       ;make string instructions inc pointers
        mov     ax,SCREEN_SEGMENT
        mov     es,ax             ;point ES to display memory
        mov     di,[bp+LeftX]
        mov     cx,[bp+RightX]
        sub     cx,di             ;width of line
        jl      DrawDone          ;RightX < LeftX; no drawing to do
        inc     cx                ;include both endpoints
        mov     ax,SCREEN_WIDTH
        mul     [bp+Y]            ;offset of scan line on which to draw
        add     di,ax             ;ES:DI points to start of line seg
        mov     al,byte ptr [bp+Color] ;color in which to draw
        mov     ah,al             ;put color in AH for STOSW
        shr     cx,1              ;# of words to fill
        rep     stosw             ;fill a word at a time
        adc     cx,cx
        rep     stosb             ;draw the odd byte, if any
DrawDone:
        pop     di                ;restore caller's register variable
        pop     bp                ;restore caller's stack frame
        ret
_DrawHorizontalLineSeg  endp
        end
```

The algorithm used to X-sort the AET is an interesting performance consideration. Listing 40.1 uses a bubble sort, usually a poor choice for performance. However, bubble sorts perform well when the data are already almost sorted, and because of the X coherence of edges from one scan line to the next, that's generally the case with the AET. An insertion sort might be somewhat faster, depending on the state of the AET when any particular sort occurs, but a bubble sort will generally do just fine.

An insertion sort that scans backward through the AET from the current edge rather than forward from the start of the AET could be quite a bit faster, because edges rarely move more than one or two positions through the AET. However, scanning backward requires a doubly linked list, rather than the singly linked list used in Listing 40.1. I've chosen to use a singly linked list partly to minimize memory requirements (double-linking requires an extra pointer field) and partly because supporting back links would complicate the code a good bit. The main reason, though, is that the potential rewards for the complications of back links and insertion sorting aren't great enough; profiling a variety of polygons reveals that less than ten percent of total time is spent sorting the AET.

The potential 1 to 5 percent speedup gained by optimizing AET sorting just isn't worth it in any but the most demanding application—a good example of the need to keep an overall perspective when comparing the theoretical characteristics of various approaches.

Nonconvex Polygons

Nonconvex polygons can be filled somewhat faster than complex polygons. Because edges never cross or switch positions with other edges once they're in the AET, the AET for a nonconvex polygon needs to be sorted only when new edges are added. In order for this to work, though, edges must be added to the AET in strict left-to-right order. Complications arise when dealing with two edges that start at the same point, because slopes must be compared to determine which edge is leftmost. This is certainly doable, but because of space limitations and limited performance returns, I haven't implemented this in Listing 40.1.

Details, Details

Every so often, a programming demon that I'd thought I'd forever laid to rest arises to haunt me once again. A minor example of this—an imp, if you will—is the use of " = " when I mean " == ," which I've done all too often in the past, and am sure I'll do again. That's minor deviltry, though, compared to the considerably greater evils of one of my personal scourges, of which I was recently reminded anew: too-close attention to detail. Not seeing the forest for the trees. Looking low when I should have looked high. Missing the big picture, if you catch my drift.

Thoreau said it best: "Our life is frittered away by detail....Simplify, simplify." That quote sprang to mind when I received a letter a while back from Anton Treuenfels of Fridley, Minnesota, thanking me for clarifying the principles of filling adjacent convex polygons in my ongoing writings on graphics programming. (You'll find this material in the previous two chapters.) Anton then went on to describe his own method for filling convex polygons.

Anton's approach had its virtues and drawbacks, foremost among the virtues being a simplicity Thoreau would have admired. For instance, in writing my polygon-filling code, I had spent quite some time trying to figure out the best way to identify which edge was the left edge and which the right, finally settling on comparing the slopes of the edges if the top of the polygon wasn't flat, and comparing the starting points of the edges if the top was flat. Anton simplified this tremendously by not bothering to figure out ahead of time which was the right edge of the polygon and which the left, instead scanning out the two edges in whatever order he found them and letting the low-level drawing code test, and if necessary swap, the endpoints of each horizontal line of the fill, so that filling started at the leftmost edge. This is a little slower than my approach (although the difference is almost surely negligible), but it also makes quite a bit of code go away.

What that example, and others like it in Anton's letter, did was kick my mind into a mode that it hadn't—but should have—been in when I wrote the code, a mode in which I began to wonder, "How else can I simplify this code?"; what you might call Occam's Razor mode. You see, I created the convex polygon-drawing code by first writing pseudocode, then writing C code, and finally writing assembly code, and once the pseudocode was finished, I stopped thinking about the interactions of the various portions of the program.

In other words, I became so absorbed in individual details that I forgot to consider the code as a whole. That was a mistake, and an embarrassing one for someone who constantly preaches that programmers should look at their code from a variety of perspectives; the next chapter shows just how much difference thinking about the big picture can make. May my embarrassment be your enlightenment.

The point is not whether, in the final analysis, my code or Anton's code is better; both have their advantages. The point is that I was programming with half a deck because I was so fixated on the details of a single type of implementation; I ended up with relatively hard-to-write, complex code, and missed out on many potentially useful optimizations by being so focused. It's a big world out there, and there are many subtle approaches to any problem, so relax and keep the big picture in mind as you implement your programs. Your code will likely be not only better, but also simpler. And whenever you see me walking across hot coals in this book or elsewhere when there's an easier way to go, please, let me know!

Thanks, Anton.