Chapter 38

The Polygon Primeval

Chapter 38

Drawing Polygons Efficiently and Quickly

"Give me but one firm spot on which to stand, and I will move the Earth."
—Archimedes

Were Archimedes alive today, he might say, "Give me but one fast polygon-fill routine on which to call, and I will draw the Earth." Programmers often think of pixel drawing as being the basic graphics primitive, but filled polygons are equally fundamental and far more useful. Filled polygons can be used for constructs as diverse as a single pixel or a 3-D surface, and virtually everything in between.

I'll spend some time in this chapter and the next several developing routines to draw filled polygons and building more sophisticated graphics operations atop those routines. Once we have that foundation, I'll get into 2-D manipulation and animation of polygon-based entities as preface to an exploration of 3-D graphics. You can't get there from here without laying some groundwork, though, so in this chapter I'll begin with the basics of filling a polygon. In the next chapter, we'll see how to draw a polygon considerably faster. That's my general approach for this sort of topic: High-level exploration of a graphics topic first, followed by a speedy hardware-specific implementation for the IBM PC/VGA combination, the most widely used graphics system around. Abstract, machine-independent graphics is a thing of beauty, but only by understanding graphics at all levels, including the hardware, can you boost performance into the realm of the sublime.

And slow computer graphics is scarcely worth the bother.

Filled Polygons

A polygon is simply a shape formed by lines laid end to end to form a continuous, closed path. A polygon is filled by setting all pixels within the polygon's boundaries to a color or pattern. For now, we'll work only with polygons filled with solid colors.

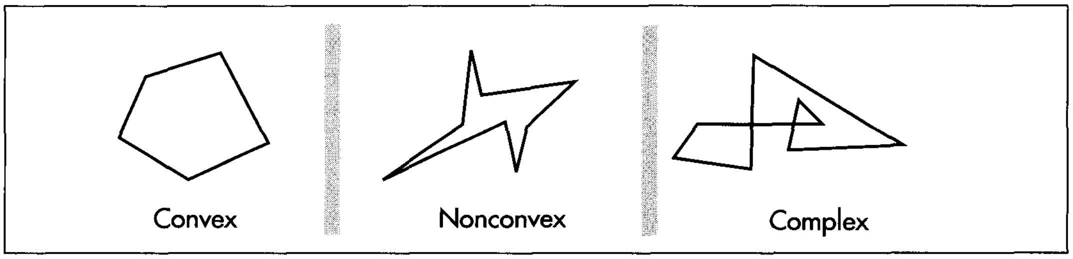
You can divide polygons into three categories: convex, nonconvex, and complex, as shown in Figure 38.1. Convex polygons include what you'd normally think of as "convex" and more; as far as we're concerned, a convex polygon is one for which any horizontal line drawn through the polygon encounters the right edge exactly once and the left edge exactly once, excluding horizontal and zero-length edge segments. Put another way, neither the right nor left edge of a convex polygon ever reverses direction from up to down, or vice-versa. Also, the right and left edges of a convex polygon may not cross one another, although they may touch so long as the right edge never crosses over to the left side of the left edge. (Check out the second polygon drawn in Listing 38.3, which certainly isn't convex in the normal sense.) The boundaries of nonconvex polygons, on the other hand, can go in whatever directions they please, so long as they never cross. Complex polygons can have any boundaries you might imagine, which makes for interesting problems in deciding which interior spaces to fill and which not to fill. Each category is a superset of the previous one.

(See Chapter 41 for a more detailed discussion of polygon types and naming.)

Why bother to distinguish between convex, nonconvex, and complex polygons? Easy: performance, especially when it comes to filling convex polygons. We're going to start with filled convex polygons; they're widely useful and will serve well to introduce some of the subtler complexities of polygon drawing, not the least of which is the slippery concept of "inside."

Which Side Is Inside?

The basic principle of polygon filling is decomposing each polygon into a series of horizontal lines, one for each horizontal row of pixels, or scan line, within the polygon (a process I'll call *scan conversion*), and drawing the horizontal lines. I'll refer to the



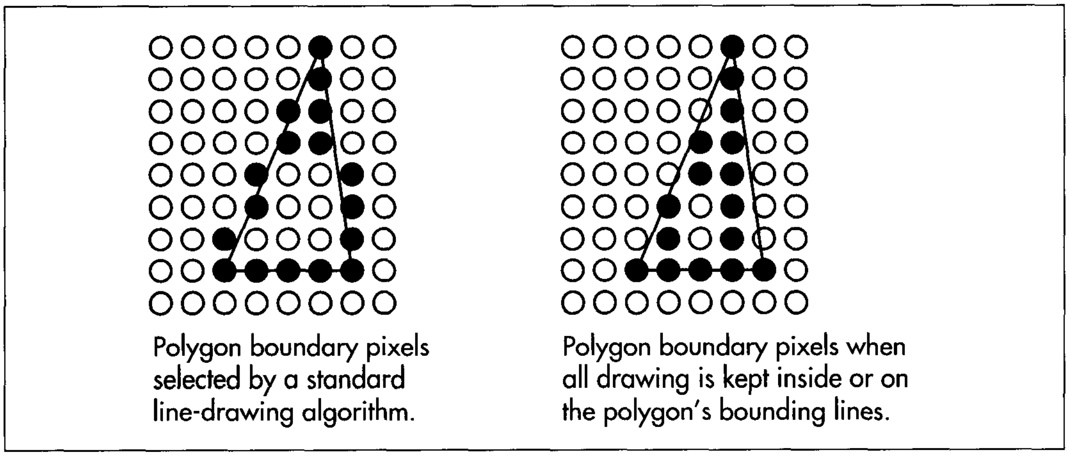
Convex, nonconvex, and complex polygons.
Figure 38.1

entire process as rasterization. Rasterization of convex polygons is easily done by starting at the top of the polygon and tracing down the left and right sides, one scan line (one vertical pixel) at a time, filling the extent between the two edges on each scan line, until the bottom of the polygon is reached. At first glance, rasterization does not seem to be particularly complicated, although it should be apparent that this simple approach is inadequate for nonconvex polygons.

There are a couple of complications, however. The lesser complication is how to rasterize the polygon efficiently, given that it's difficult to write fast code that simultaneously traces two edges and fills the space between them. The solution is to decouple the process of scan-converting the polygon into a list of horizontal lines from that of drawing the horizontal lines. One device-independent routine can trace along the two edges and build a list of the beginning and end coordinates of the polygon on each raster line. Then a second, device-specific, routine can draw from the list after the entire polygon has been scanned. We'll see this in action shortly.

The second, greater complication arises because the definition of which pixels are "within" a polygon is a more complicated matter than you might imagine. You might think that scan-converting an edge of a polygon is analogous to drawing a line from one vertex to the next, but this is not so. A line by itself is a one-dimensional construct, and as such is approximated on a display by drawing the pixels nearest to the line on either side of the true line. A line serving as a polygon boundary, on the other hand, is part of a two-dimensional object. When filling a polygon, we want to draw the pixels within the polygon, but a standard vertex-to-vertex line-drawing algorithm will draw many pixels outside the polygon, as shown in Figure 38.2.

It's no crime to use standard lines to trace out a polygon, rather than drawing only interior pixels. In fact, there are certain advantages: For example, the edges of a


Polygon boundary pixels selected by a standard line-drawing algorithm.
Polygon boundary pixels when all drawing is kept inside or on the polygon's bounding lines.

Drawing polygons with standard line-drawing algorithms.
Figure 38.2

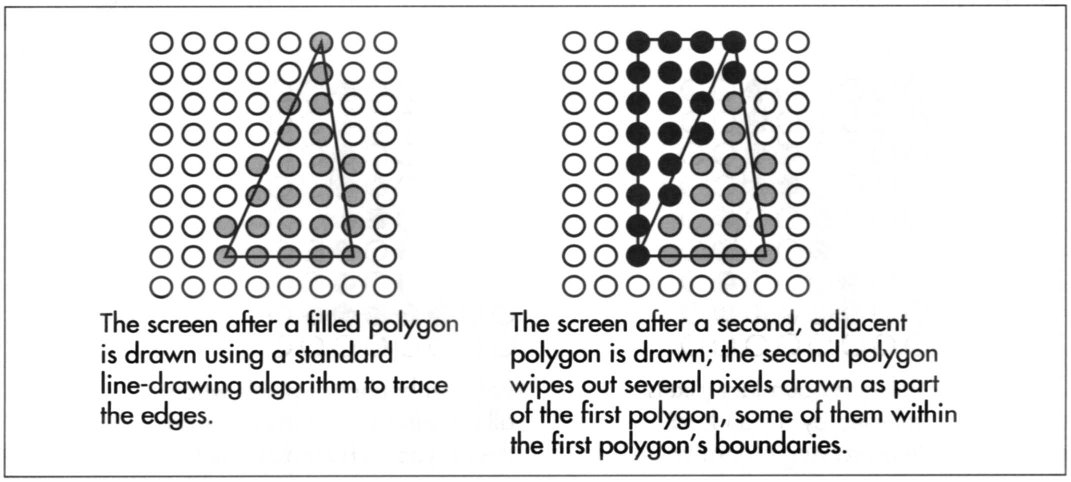
filled polygon will match the edges of the same polygon drawn unfilled. Such polygons will look pretty much as they're supposed to, and all drawing on raster displays is, after all, only an approximation of an ideal.

There's one great drawback to tracing polygons with standard lines, however: Adjacent polygons won't fit together properly, as shown in Figure 38.3. If you use six equilateral triangles to make a hexagon, for example, the edges of the triangles will overlap when traced with standard lines, and more recently drawn triangles will wipe out portions of their predecessors. Worse still, odd color effects will show up along the polygon boundaries if XOR drawing is used. Consequently, filling out to the boundary lines just won't do for drawing images composed of fitted-together polygons. And because fitting polygons together is exactly what I have in mind, we need a different approach.

How Do You Fit Polygons Together?

How, then, do you fit polygons together? *Very* carefully. First, the line-tracing algorithm must be adjusted so that it selects only those pixels that are truly inside the polygon. This basically requires shifting a standard line-drawing algorithm horizontally by one half-pixel toward the polygon's interior. That leaves the issue of how to handle points that are exactly on the boundary, and points that lie at vertices, so that those points are drawn once and only once. To deal with that, we're going to adopt the following rules:

- Points located exactly on nonhorizontal edges are drawn only if the interior of the polygon is directly to the right (left edges are drawn, right edges aren't).


The screen after a filled polygon is drawn using a standard line-drawing algorithm to trace the edges.
The screen after a second, adjacent polygon is drawn; the second polygon wipes out several pixels drawn as part of the first polygon, some of them within the first polygon's boundaries.

The adjacent polygons problem.
Figure 38.3

- Points located exactly on horizontal edges are drawn only if the interior of the polygon is directly below them (horizontal top edges are drawn, horizontal bottom edges aren't).
- A vertex is drawn only if all lines ending at that point meet the above conditions (no right or bottom edges end at that point).

All edges of a polygon except those that are flat tops or flat bottoms will be considered either right edges or left edges, regardless of slope. The left edge is the one that starts with the leftmost line down from the top of the polygon.

These rules ensure that no pixel is drawn more than once when adjacent polygons are filled, and that if polygons cover the full 360-degree range around a pixel, then that pixel will be drawn once and only once—just what we need in order to be able to fit filled polygons together seamlessly.

This sort of non-overlapping polygon filling isn't ideal for all purposes. Polygons are skewed toward the top and left edges, which not only introduces drawing error relative to the ideal polygon but also means that a filled polygon won't match the same polygon drawn unfilled. Narrow wedges and one-pixel-wide polygons will show up spottily. All in all, the choice of polygon-filling approach depends entirely on the ways in which the filled polygons must be used.

For our purposes, nonoverlapping polygons are the way to go, so let's have at them.

Filling Non-Overlapping Convex Polygons

Without further ado, Listing 38.1 contains a function, **FillConvexPolygon**, that accepts a list of points that describe a convex polygon, with the last point assumed to connect to the first, and scans it into a list of lines to fill, then passes that list to the function **DrawHorizontalLineList** in Listing 38.2. Listing 38.3 is a sample program that calls **FillConvexPolygon** to draw polygons of various sorts, and Listing 38.4 is a header file included by the other listings. Here are the listings; we'll pick up discussion on the other side.

LISTING 38.1 L38-1.C

```
/* Color-fills a convex polygon. All vertices are offset by (XOffset,
   YOffset). "Convex" means that 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), and that the
   right & left edges never cross. (It's OK for them to touch, though,
   so long as the right edge never crosses over to the left of the
   left edge.) Nonconvex polygons won't be drawn properly. Returns 1
   for success, 0 if memory allocation failed. */

#include <stdio.h>
#include <math.h>
#ifdef __TURBOC__
```

```
#include <alloc.h>
#else    /* MSC */
#include <malloc.h>
#endif
#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 DrawHorizontalLineList(struct HLineList *, int);
static void ScanEdge(int, int, int, int, int, int, struct HLine **);

int FillConvexPolygon(struct PointListHeader * VertexList, int Color,
      int XOffset, int YOffset)
{
   int i, MinIndexL, MaxIndex, MinIndexR, SkipFirst, Temp;
   int MinPoint_Y, MaxPoint_Y, TopIsFlat, LeftEdgeDir;
   int NextIndex, CurrentIndex, PreviousIndex;
   int DeltaXN, DeltaYN, DeltaXP, DeltaYP;
   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[MinIndexL = MaxIndex = 0].Y;
   for (i = 1; i < VertexList->Length; i++) {
      if (VertexPtr[i].Y < MinPoint_Y)
         MinPoint_Y = VertexPtr[MinIndexL = i].Y; /* new top */
      else if (VertexPtr[i].Y > MaxPoint_Y)
         MaxPoint_Y = VertexPtr[MaxIndex = i].Y; /* new bottom */
   }
   if (MinPoint_Y == MaxPoint_Y)
      return(1);  /* polygon is 0-height; avoid infinite loop below */

   /* Scan in ascending order to find the last top-edge point */
   MinIndexR = MinIndexL;
   while (VertexPtr[MinIndexR].Y == MinPoint_Y)
      INDEX_FORWARD(MinIndexR);
   INDEX_BACKWARD(MinIndexR); /* back up to last top-edge point */
```

```
/* Now scan in descending order to find the first top-edge point */
while (VertexPtr[MinIndexL].Y == MinPoint_Y)
   INDEX_BACKWARD(MinIndexL);
INDEX_FORWARD(MinIndexL); /* back up to first top-edge point */

/* Figure out which direction through the vertex list from the top
   vertex is the left edge and which is the right */
LeftEdgeDir = -1; /* assume left edge runs down thru vertex list */
if ((TopIsFlat = (VertexPtr[MinIndexL].X !=
      VertexPtr[MinIndexR].X) ? 1 : 0) == 1) {
   /* If the top is flat, just see which of the ends is leftmost */
   if (VertexPtr[MinIndexL].X > VertexPtr[MinIndexR].X) {
      LeftEdgeDir = 1;  /* left edge runs up through vertex list */
      Temp = MinIndexL;       /* swap the indices so MinIndexL   */
      MinIndexL = MinIndexR;  /* points to the start of the left */
      MinIndexR = Temp;       /* edge, similarly for MinIndexR   */
   }
} else {
   /* Point to the downward end of the first line of each of the
      two edges down from the top */
   NextIndex = MinIndexR;
   INDEX_FORWARD(NextIndex);
   PreviousIndex = MinIndexL;
   INDEX_BACKWARD(PreviousIndex);
   /* Calculate X and Y lengths from the top vertex to the end of
      the first line down each edge; use those to compare slopes
      and see which line is leftmost */
   DeltaXN = VertexPtr[NextIndex].X - VertexPtr[MinIndexL].X;
   DeltaYN = VertexPtr[NextIndex].Y - VertexPtr[MinIndexL].Y;
   DeltaXP = VertexPtr[PreviousIndex].X - VertexPtr[MinIndexL].X;
   DeltaYP = VertexPtr[PreviousIndex].Y - VertexPtr[MinIndexL].Y;
   if (((long)DeltaXN * DeltaYP - (long)DeltaYN * DeltaXP) < 0L) {
      LeftEdgeDir = 1;  /* left edge runs up through vertex list */
      Temp = MinIndexL;       /* swap the indices so MinIndexL   */
      MinIndexL = MinIndexR;  /* points to the start of the left */
      MinIndexR = Temp;       /* edge, similarly for MinIndexR   */
   }
}

/* Set the # of scan lines in the polygon, skipping the bottom edge
   and also skipping the top vertex if the top isn't flat because
   in that case the top vertex has a right edge component, and set
   the top scan line to draw, which is likewise the second line of
   the polygon unless the top is flat */
if ((WorkingHLineList.Length =
      MaxPoint_Y - MinPoint_Y - 1 + TopIsFlat) <= 0)
   return(1);  /* there's nothing to draw, so we're done */
WorkingHLineList.YStart = YOffset + MinPoint_Y + 1 - TopIsFlat;

/* 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 left edge and store the boundary points in the list */
/* Initial pointer for storing scan converted left-edge coords */
EdgePointPtr = WorkingHLineList.HLinePtr;
/* Start from the top of the left edge */
PreviousIndex = CurrentIndex = MinIndexL;
```

```
   /* Skip the first point of the first line unless the top is flat;
      if the top isn't flat, the top vertex is exactly on a right
      edge and isn't drawn */
   SkipFirst = TopIsFlat ? 0 : 1;
   /* Scan convert each line in the left edge from top to bottom */
   do {
      INDEX_MOVE(CurrentIndex,LeftEdgeDir);
      ScanEdge(VertexPtr[PreviousIndex].X + XOffset,
            VertexPtr[PreviousIndex].Y,
            VertexPtr[CurrentIndex].X + XOffset,
            VertexPtr[CurrentIndex].Y, 1, SkipFirst, &EdgePointPtr);
      PreviousIndex = CurrentIndex;
      SkipFirst = 0; /* scan convert the first point from now on */
   } while (CurrentIndex != MaxIndex);

   /* Scan the right edge and store the boundary points in the list */
   EdgePointPtr = WorkingHLineList.HLinePtr;
   PreviousIndex = CurrentIndex = MinIndexR;
   SkipFirst = TopIsFlat ? 0 : 1;
   /* Scan convert the right edge, top to bottom. X coordinates are
      adjusted 1 to the left, effectively causing scan conversion of
      the nearest points to the left of but not exactly on the edge */
   do {
      INDEX_MOVE(CurrentIndex,-LeftEdgeDir);
      ScanEdge(VertexPtr[PreviousIndex].X + XOffset - 1,
            VertexPtr[PreviousIndex].Y,
            VertexPtr[CurrentIndex].X + XOffset - 1,
            VertexPtr[CurrentIndex].Y, 0, SkipFirst, &EdgePointPtr);
      PreviousIndex = CurrentIndex;
      SkipFirst = 0; /* scan convert the first point from now on */
   } 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);
}

/* Scan converts an edge from (X1,Y1) to (X2,Y2), not including the
   point at (X2,Y2). This avoids overlapping the end of one line with
   the start of the next, and causes the bottom scan line of the
   polygon not to be drawn. If SkipFirst != 0, the point at (X1,Y1)
   isn't drawn. For each scan line, the pixel closest to the scanned
   line without being to the left of the scanned line is chosen. */
static void ScanEdge(int X1, int Y1, int X2, int Y2, int SetXStart,
      int SkipFirst, struct HLine **EdgePointPtr)
{
   int Y, DeltaX, DeltaY;
   double InverseSlope;
   struct HLine *WorkingEdgePointPtr;

   /* Calculate X and Y lengths of the line and the inverse slope */
   DeltaX = X2 - X1;
   if ((DeltaY = Y2 - Y1) <= 0)
      return;     /* guard against 0-length and horizontal edges */
   InverseSlope = (double)DeltaX / (double)DeltaY;
   /* Store the X coordinate of the pixel closest to but not to the
      left of the line for each Y coordinate between Y1 and Y2, not
      including Y2 and also not including Y1 if SkipFirst != 0 */
   WorkingEdgePointPtr = *EdgePointPtr; /* avoid double dereference */
   for (Y = Y1 + SkipFirst; Y < Y2; Y++, WorkingEdgePointPtr++) {
      /* Store the X coordinate in the appropriate edge list */
      if (SetXStart == 1)
         WorkingEdgePointPtr->XStart =
               X1 + (int)(ceil((Y-Y1) * InverseSlope));
      else
         WorkingEdgePointPtr->XEnd =
               X1 + (int)(ceil((Y-Y1) * InverseSlope));
   }
   *EdgePointPtr = WorkingEdgePointPtr;   /* advance caller's ptr */
}
```

LISTING 38.2 L38-2.C

```
/* Draws all pixels in the list of horizontal lines passed in, in
   mode 13h, the VGA's 320x200 256-color mode. Uses a slow pixel-by-
   pixel approach, which does have the virtue of being easily ported
   to any environment. */

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

#define SCREEN_WIDTH    320
#define SCREEN_SEGMENT  0xA000

static void DrawPixel(int, int, int);

void DrawHorizontalLineList(struct HLineList * HLineListPtr,
      int Color)
{
   struct HLine *HLinePtr;
   int Y, X;

   /* 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 */
   for (Y = HLineListPtr->YStart; Y < (HLineListPtr->YStart +
         HLineListPtr->Length); Y++, HLinePtr++) {
      /* Draw each pixel in the current horizontal line in turn,
         starting with the leftmost one */
      for (X = HLinePtr->XStart; X <= HLinePtr->XEnd; 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 38.3 L38-3.C

```
/* Sample program to exercise the polygon-filling routines. This code
   and all polygon-filling code has been tested with Borland and
   Microsoft compilers. */

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

/* Draws the polygon described by the point list PointList in color
   Color with all vertices offset by (X,Y) */
#define DRAW_POLYGON(PointList,Color,X,Y)                  \
   Polygon.Length = sizeof(PointList)/sizeof(struct Point); \
   Polygon.PointPtr = PointList;                           \
   FillConvexPolygon(&Polygon, Color, X, Y);

void main(void);
extern int FillConvexPolygon(struct PointListHeader *, int, int, int);

void main() {
   int i, j;
   struct PointListHeader Polygon;
   static struct Point ScreenRectangle[] =
         {{0,0},{320,0},{320,200},{0,200}};
   static struct Point ConvexShape[] =
         {{0,0},{121,0},{320,0},{200,51},{301,51},{250,51},{319,143},
         {320,200},{22,200},{0,200},{50,180},{20,160},{50,140},
         {20,120},{50,100},{20,80},{50,60},{20,40},{50,20}};
   static struct Point Hexagon[] =
         {{90,-50},{0,-90},{-90,-50},{-90,50},{0,90},{90,50}};
   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;   /* AH = 0 selects mode set function,
                              AL = 0x13 selects mode 0x13
                              when set as parameters for INT 0x10 */
   int86(0x10, &regset, &regset);

   /* Clear the screen to cyan */
   DRAW_POLYGON(ScreenRectangle, 3, 0, 0);

   /* Draw an irregular shape that meets our definition of convex but
      is not convex by any normal description */
   DRAW_POLYGON(ConvexShape, 6, 0, 0);
   getch();    /* wait for a keypress */

   /* Draw adjacent triangles across the top half of the screen */
   for (j=0; j<=80; j+=20) {
      for (i=0; i<290; i += 30) {
         DRAW_POLYGON(Triangle1, 2, i, j);
         DRAW_POLYGON(Triangle2, 4, i+15, j);
      }
   }
   /* Draw adjacent triangles across the bottom half of the screen */
   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, 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, i, j+10);
      }
   }
   getch();    /* wait for a keypress */

   /* Finally, draw a series of concentric hexagons of approximately
      the same proportions in the center of the screen */
   for (i=0; i<16; i++) {
      DRAW_POLYGON(Hexagon, i, 160, 100);
      for (j=0; j<sizeof(Hexagon)/sizeof(struct Point); j++) {
         /* Advance each vertex toward the center */
         if (Hexagon[j].X != 0) {
            Hexagon[j].X -= Hexagon[j].X >= 0 ? 3 : -3;
            Hexagon[j].Y -= Hexagon[j].Y >= 0 ? 2 : -2;
         } else {
            Hexagon[j].Y -= Hexagon[j].Y >= 0 ? 3 : -3;
         }
      }
   }
   getch();    /* wait for a keypress */

   /* Return to text mode and exit */
   regset.x.ax = 0x0003;   /* AL = 3 selects 80x25 text mode */
   int86(0x10, &regset, &regset);
}
```

LISTING 38.4 POLYGON.H

```
/* POLYGON.H: Header file for polygon-filling code */

/* 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 is assumed to connect to the two
   adjacent vertices, and 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 */
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) */
struct HLineList {
   int Length;                /* # of horizontal lines */
   int YStart;                /* Y coordinate of topmost line */
   struct HLine * HLinePtr;   /* pointer to list of horz lines */
};
```

Listing 38.2 isn't particularly interesting; it merely draws each horizontal line in the passed-in list in the simplest possible way, one pixel at a time. (No, that doesn't make the pixel the fundamental primitive; in the next chapter I'll replace Listing 38.2 with a much faster version that doesn't bother with individual pixels at all.)

Listing 38.1 is where the action is in this chapter. Our goal is to scan out the left and right edges of each polygon so that all points inside and no points outside the polygon are drawn, and so that all points located exactly on the boundary are drawn only if they are not on right or bottom edges. That's precisely what Listing 38.1 does. Here's how:

Listing 38.1 first finds the top and bottom of the polygon, then works out from the top point to find the two ends of the top edge. If the ends are at different locations, the top is flat, which has two implications. First, it's easy to find the starting vertices and directions through the vertex list for the left and right edges. (To scan-convert them properly, we must first determine which edge is which.) Second, the top scan line of the polygon should be drawn without the rightmost pixel, because only the rightmost pixel of the horizontal edge that makes up the top scan line is part of a right edge.

If, on the other hand, the ends of the top edge are at the same location, the top is pointed. In that case, the top scan line of the polygon isn't drawn; it's part of the right-edge line that starts at the top vertex. (It's part of a left-edge line, too, but the right edge overrides.) When the top isn't flat, it's more difficult to tell in which direction through the vertex list the right and left edges go, because both edges start at the top vertex. The solution is to compare the slopes from the top vertex to the ends of the two lines coming out of it in order to see which is leftmost. The calculations in Listing 38.1 involving the various deltas do this, using a rearranged form of the slope-based equation:

```
(DeltaYN/DeltaXN)>(DeltaYP/DeltaXP)
```

Once we know where the left edge starts in the vertex list, we can scan-convert it a line segment at a time until the bottom vertex is reached. Each point is stored as the starting X coordinate for the corresponding scan line in the list we'll pass to **DrawHorizontalLineList**. The nearest X coordinate on each scan line that's on or to the right of the left edge is selected. The last point of each line segment making up the left edge isn't scan-converted, producing two desirable effects. First, it avoids drawing each vertex twice; two lines come into every vertex, but we want to scan-convert each vertex only once. Second, not scan-converting the last point of each line causes the bottom scan line of the polygon not to be drawn, as required by our rules. The first scan line of the polygon is also skipped if the top isn't flat.

Now we need to scan-convert the right edge into the ending X coordinate fields of the line list. This is performed in the same manner as for the left edge, except that every line in the right edge is moved one pixel to the left before being scan-converted. Why? We want the nearest point to the left of but not on the right edge, so that the right edge itself isn't drawn. As it happens, drawing the nearest point on or to the right of a line moved one pixel to the left is exactly the same as drawing the nearest point to the left of but not on that line in its original location. Sketch it out and you'll see what I mean.

Once the two edges are scan-converted, the whole line list is passed to **DrawHorizontalLineList**, and the polygon is drawn.

Finis.

Oddball Cases

Listing 38.1 handles zero-length segments (multiple vertices at the same location) by ignoring them, which will be useful down the road because scaled-down polygons can end up with nearby vertices moved to the same location. Horizontal line segments are fine anywhere in a polygon, too. Basically, Listing 38.1 scan-converts between active edges (the edges that define the extent of the polygon on each scan line) and both horizontal and zero-length lines are non-active; neither advances to another scan line, so they don't affect the edges being scanned.

I've limited this chapter's code to merely demonstrating the principles of filling convex polygons, and the listings given are by no means fast. In the next chapter, we'll spice things up by eliminating the floating point calculations and pixel-at-a-time drawing and tossing a little assembly language into the mix.