Chapter 35

Bresenham Is Fast, and Fast Is Good

Chapter 35

Implementing and Optimizing Bresenham's Line-Drawing Algorithm

For all the complexity of graphics design and programming, surprisingly few primitive functions lie at the heart of most graphics software. Heavily used primitives include routines that draw dots, circles, area fills, bit block logical transfers, and, of course, lines. For many years, computer graphics were created primarily with specialized line-drawing hardware, so lines are in a way the *lingua franca* of computer graphics. Lines are used in a wide variety of microcomputer graphics applications today, notably CAD/CAM and computer-aided engineering.

Probably the best-known formula for drawing lines on a computer display is called Bresenham's line-drawing algorithm. (We have to be specific here because there is also a less-well-known Bresenham's circle-drawing algorithm.) In this chapter, I'll present two implementations for the EGA and VGA of Bresenham's line-drawing algorithm, which provides decent line quality and excellent drawing speed.

The first implementation is in rather plain C, with the second in not-so-plain assembly, and they're both pretty good code. The assembly implementation is damned good code, in fact, but if you want to know whether it's the fastest Bresenham's implementation possible, I must tell you that it isn't. First of all, the code could be sped up a bit by shuffling and combining the various error-term manipulations, but that results in *truly* cryptic code. I wanted you to be able to relate the original algorithm to the final code, so I skipped those optimizations. Also, write mode 3, which is unique to the VGA, could be used for considerably faster drawing. I've described write mode 3 in earlier chapters, and I strongly recommend its use in VGA-only line drawing.

Second, horizontal, vertical, and diagonal lines could be special-cased, since those particular lines require little calculation and can be drawn very rapidly. (This is especially true of horizontal lines, which can be drawn 8 pixels at a time.)

Third, run-length slice line drawing could be used to significantly reduce the number of calculations required per pixel, as I'll demonstrate in the next two chapters.

Finally, unrolled loops and/or duplicated code could be used to eliminate most of the branches in the final assembly implementation, and because x86 processors are notoriously slow at branching, that would make quite a difference in overall performance. If you're interested in unrolled loops and similar assembly techniques, I refer you to the first part of this book.

That brings us neatly to my final point: Even if I didn't know that there were further optimizations to be made to my line-drawing implementation, I'd *assume* that there were. As I'm sure the experienced assembly programmers among you know, there are dozens of ways to tackle any problem in assembly, and someone else always seems to have come up with a trick that never occurred to you. I've incorporated a suggestion made by Jim Mackraz in the code in this chapter, and I'd be most interested in hearing of any other tricks or tips you may have.

Notwithstanding, the line-drawing implementation in Listing 35.3 is plenty fast enough for most purposes, so let's get the discussion underway.

The Task at Hand

There are two important characteristics of any line-drawing function. First, it must draw a reasonable approximation of a line. A computer screen has limited resolution, and so a line-drawing function must actually approximate a straight line by drawing a series of pixels in what amounts to a jagged pattern that generally proceeds in the desired direction. That pattern of pixels must reliably suggest to the human eye the true line it represents. Second, to be usable, a line-drawing function must be *fast*. Minicomputers and mainframes generally have hardware that performs line drawing, but most microcomputers offer no such assistance. True, nowadays graphics accelerators such as the S3 and ATI chips have line drawing hardware, but some other accelerators don't; when drawing lines on the latter sort of chip, when drawing on the CGA, EGA, and VGA, and when drawing sorts of lines not supported by line-drawing hardware as well, the PC's CPU must draw lines on its own, and, as many users of graphics-oriented software know, that can be a slow process indeed.

Line drawing quality and speed derive from two factors: The algorithm used to draw the line and the implementation of that algorithm. The first implementation (written in Borland C++) that I'll be presenting in this chapter illustrates the workings of the algorithm and draws lines at a good rate. The second implementation, written in assembly language and callable directly from Borland C++, draws lines at extremely high speed, on the order of three to six times faster than the C version. Between them, the two implementations illuminate Bresenham's line-drawing algorithm and provide high-performance line-drawing capability.

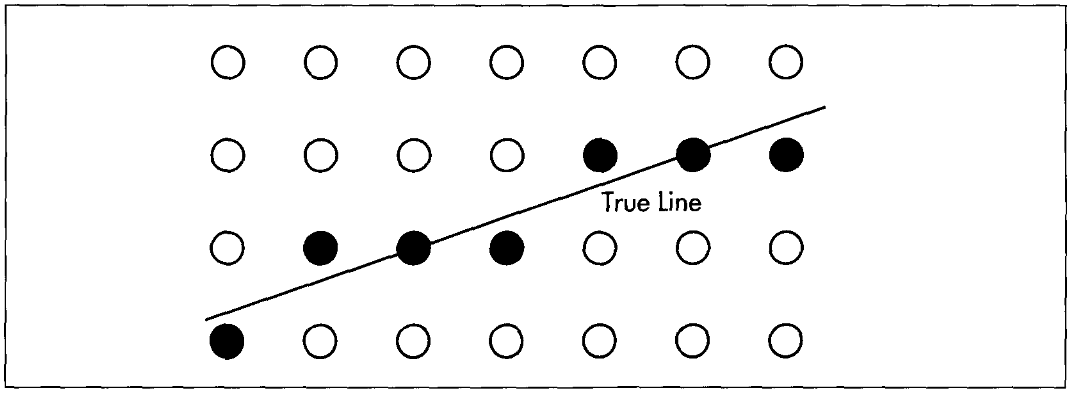
The difficulty in drawing a line lies in generating a set of pixels that, taken together, are a reasonable facsimile of a true line. Only horizontal, vertical, and 1:1 diagonal lines can be drawn precisely along the true line being represented; all other lines must be approximated from the array of pixels that a given video mode supports, as shown in Figure 35.1.

Considerable thought has gone into the design of line-drawing algorithms, and a number of techniques for drawing high-quality lines have been developed. Unfortunately, most of these techniques were developed for powerful, expensive graphics workstations and require very high resolution, a large color palette, and/or floating-point hardware. These techniques tend to perform poorly and produce less visually impressive results on all but the best-endowed PCs.

Bresenham's line-drawing algorithm, on the other hand, is uniquely suited to microcomputer implementation in that it requires no floating-point operations, no divides, and no multiplies inside the line-drawing loop. Moreover, it can be implemented with surprisingly little code.

Bresenham's Line-Drawing Algorithm

The key to grasping Bresenham's algorithm is to understand that when drawing an approximation of a line on a finite-resolution display, each pixel drawn will lie either exactly on the true line or to one side or the other of the true line. The amount by which the pixel actually drawn deviates from the true line is the *error* of the line



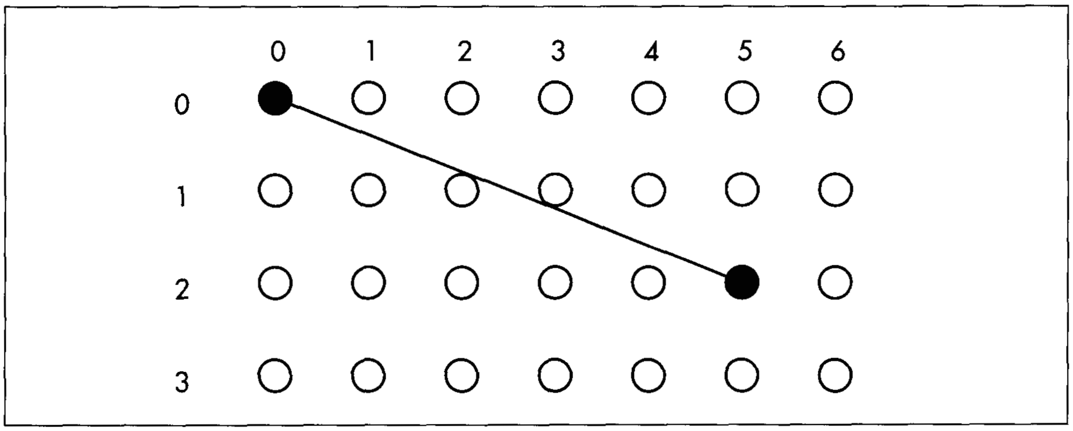
Approximating a true line from a pixel array.
Figure 35.1

drawing at that point. As the drawing of the line progresses from one pixel to the next, the error can be used to tell when, given the resolution of the display, a more accurate approximation of the line can be drawn by placing a given pixel one unit of screen resolution away from its predecessor in either the horizontal or the vertical direction, or both.

Let's examine the case of drawing a line where the horizontal, or X length of the line is greater than the vertical, or Y length, and both lengths are greater than 0. For example, suppose we are drawing a line from (0,0) to (5,2), as shown in Figure 35.2. Note that Figure 35.2 shows the upper-left-hand corner of the screen as (0,0), rather than placing (0,0) at its more traditional lower-left-hand corner location. Due to the way in which the PC's graphics are mapped to memory, it is simpler to work within this framework, although a translation of Y from increasing downward to increasing upward could be effected easily enough by simply subtracting the Y coordinate from the screen height minus 1; if you are more comfortable with the traditional coordinate system, feel free to modify the code in Listings 35.1 and 35.3.

In Figure 35.2, the endpoints of the line fall exactly on displayed pixels. However, no other part of the line squarely intersects the center of a pixel, meaning that all other pixels will have to be plotted as approximations of the line. The approach to approximation that Bresenham's algorithm takes is to move exactly 1 pixel along the major dimension of the line each time a new pixel is drawn, while moving 1 pixel along the minor dimension each time the line moves more than halfway between pixels along the minor dimension.

In Figure 35.2, the X dimension is the major dimension. This means that 6 dots, one at each of X coordinates 0, 1, 2, 3, 4, and 5, will be drawn. The trick, then, is to decide on the correct Y coordinates to accompany those X coordinates.



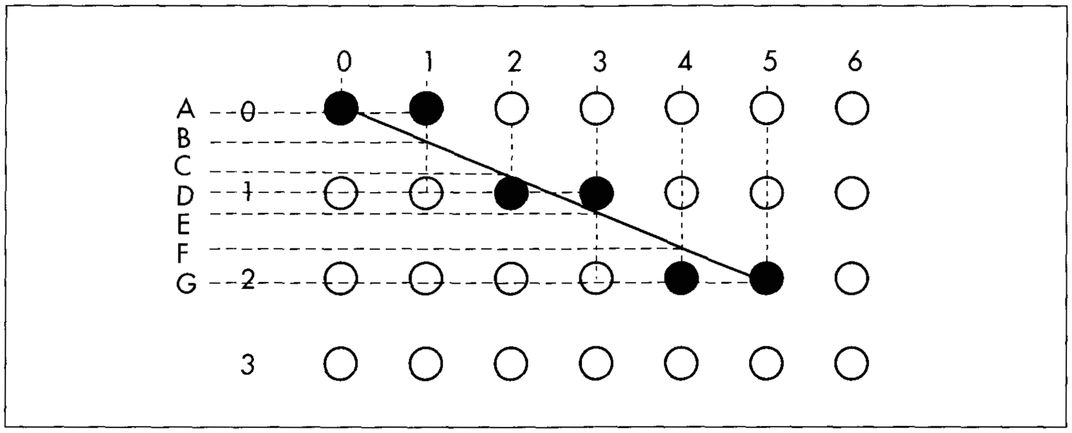
Drawing between two pixel endpoints.
Figure 35.2

It's easy enough to select the Y coordinates by eye in Figure 35.2. The appropriate Y coordinates are 0, 0, 1, 1, 2, 2, based on the Y coordinate closest to the line for each X coordinate. Bresenham's algorithm makes the same selections, based on the same criterion. The manner in which it does this is by keeping a running record of the error of the line—that is, how far from the true line the current Y coordinate is—at each X coordinate, as shown in Figure 35.3. When the running error of the line indicates that the current Y coordinate deviates from the true line to the extent that the adjacent Y coordinate would be closer to the line, then the current Y coordinate is changed to that adjacent Y coordinate.

Let's take a moment to follow the steps Bresenham's algorithm would go through in drawing the line in Figure 35.3. The initial pixel is drawn at (0,0), the starting point of the line. At this point the error of the line is 0.

Since X is the major dimension, the next pixel has an X coordinate of 1. The Y coordinate of this pixel will be whichever of 0 (the last Y coordinate) or 1 (the adjacent Y coordinate in the direction of the end point of the line) the true line at this X coordinate is closer to. The running error at this point is B minus A, as shown in Figure 35.3. This amount is less than 1/2 (that is, less than halfway to the next Y coordinate), so the Y coordinate does not change at X equal to 1. Consequently, the second pixel is drawn at (1,0).

The third pixel has an X coordinate of 2. The running error at this point is C minus A, which is greater than 1/2 and therefore closer to the next than to the current Y coordinate. The third pixel is drawn at (2,1), and 1 is subtracted from the running error to compensate for the adjustment of one pixel in the current Y coordinate. The running error of the pixel actually drawn at this point is C minus D.



The error term in Bresenham's algorithm.
Figure 35.3

The fourth pixel has an X coordinate of 3. The running error at this point is E minus D; since this is less than 1/2, the current Y coordinate doesn't change. The fourth pixel is drawn at (3,1).

The fifth pixel has an X coordinate of 4. The running error at this point is F minus D; since this is greater than 1/2, the current Y coordinate advances. The third pixel is drawn at (4,2), and 1 is subtracted from the running error. The error of the pixel drawn at this point is G minus F.

Finally, the sixth pixel is the end point of the line. This pixel has an X coordinate of 5. The running error at this point is G minus G, or 0, indicating that this point is squarely on the true line, as of course it should be given that it's the end point, so the current Y coordinate remains the same. The end point of the line is drawn at (5,2), and the line is complete.

That's really all there is to Bresenham's algorithm. The algorithm is a process of drawing a pixel at each possible coordinate along the major dimension of the line, each with the closest possible coordinate along the minor dimension. The running error is used to keep track of when the coordinate along the minor dimension must change in order to remain as close as possible to the true line. The above description of the case where X is the major dimension, Y is the minor dimension, and both dimensions are greater than zero is readily generalized to all eight octants in which lines could be drawn, as we will see in the C implementation.

The above discussion summarizes the nature rather than the exact mechanism of Bresenham's line-drawing algorithm. I'll provide a brief seat-of-the-pants discussion of the algorithm in action when we get to the C implementation of the algorithm; for a full mathematical treatment, I refer you to pages 433–436 of Foley and Van Dam's *Fundamentals of Interactive Computer Graphics* (Addison-Wesley, 1982), or pages 72–78 of the second edition of that book, which was published under the name *Computer Graphics: Principles and Practice* (Addison-Wesley, 1990). These sources provide the derivation of the integer-only, divide-free version of the algorithm, as well as Pascal code for drawing lines in one of the eight possible octants.

Strengths and Weaknesses

The overwhelming strength of Bresenham's line-drawing algorithm is speed. With no divides, no floating-point operations, and no need for variables that won't fit in 16 bits, it is perfectly suited for PCs.

The weakness of Bresenham's algorithm is that it produces relatively low-quality lines by comparison with most other line-drawing algorithms. In particular, lines generated with Bresenham's algorithm can tend to look a little jagged. On the PC, however, jagged lines are an inevitable consequence of relatively low resolution and a small color set, so lines drawn with Bresenham's algorithm don't look all that much different from lines drawn in other ways. Besides, in most applications, users are far more interested in the overall picture than in the primitive elements from which that picture is built. As a general rule, any collection of pixels that trend from point A to point B in a straight fashion is accepted by the eye as a line. Bresenham's algorithm is successfully used by many current PC programs, and by the standard of this wide acceptance the algorithm is certainly good enough.

Then, too, users hate waiting for their computer to finish drawing. By any standard of drawing performance, Bresenham's algorithm excels.

An Implementation in C

It's time to get down and look at some actual working code. Listing 35.1 is a C implementation of Bresenham's line-drawing algorithm for modes 0EH, 0FH, 10H, and 12H of the VGA, called as function **EVGALine**. Listing 35.2 is a sample program to demonstrate the use of **EVGALine**.

LISTING 35.1 L35-1.C

```
/*
 * C implementation of Bresenham's line drawing algorithm
 * for the EGA and VGA. Works in modes 0xE, 0xF, 0x10, and 0x12.
 *
 * Compiled with Borland C++
 *
 * By Michael Abrash
 */

#include <dos.h>        /* contains MK_FP macro */

#define EVGA_SCREEN_WIDTH_IN_BYTES     80
                                       /* memory offset from start of
                                          one row to start of next */
#define EVGA_SCREEN_SEGMENT            0xA000
                                       /* display memory segment */
#define GC_INDEX                       0x3CE
                                       /* Graphics Controller
                                          Index register port */
#define GC_DATA                        0x3CF
                                       /* Graphics Controller
                                          Data register port */
#define SET_RESET_INDEX                0  /* indexes of needed */
#define ENABLE_SET_RESET_INDEX         1  /* Graphics Controller */
#define BIT_MASK_INDEX                 8  /* registers */

/*
 * Draws a dot at (X0,Y0) in whatever color the EGA/VGA hardware is
 * set up for. Leaves the bit mask set to whatever value the
 * dot required.
 */
void EVGADot(X0, Y0)
unsigned int X0;     /* coordinates at which to draw dot, with */
unsigned int Y0;     /* (0,0) at the upper left of the screen */
{
   unsigned char far *PixelBytePtr;
   unsigned char PixelMask;
   /* Calculate the offset in the screen segment of the byte in
      which the pixel lies */
   PixelBytePtr = MK_FP(EVGA_SCREEN_SEGMENT,
      ( Y0 * EVGA_SCREEN_WIDTH_IN_BYTES ) + ( X0 / 8 ));

   /* Generate a mask with a 1 bit in the pixel's position within the
      screen byte */
   PixelMask = 0x80 >> ( X0 & 0x07 );

   /* Set up the Graphics Controller's Bit Mask register to allow
      only the bit corresponding to the pixel being drawn to
      be modified */
   outportb(GC_INDEX, BIT_MASK_INDEX);
   outportb(GC_DATA, PixelMask);

   /* Draw the pixel. Because of the operation of the set/reset
      feature of the EGA/VGA, the value written doesn't matter.
      The screen byte is ORed in order to perform a read to latch the
      display memory, then perform a write in order to modify it. */
   *PixelBytePtr |= 0xFE;
}

/*
 * Draws a line in octant 0 or 3 ( |DeltaX| >= DeltaY ).
 */
void Octant0(X0, Y0, DeltaX, DeltaY, XDirection)
unsigned int X0, Y0;          /* coordinates of start of the line */
unsigned int DeltaX, DeltaY;  /* length of the line (both > 0) */
int XDirection;               /* 1 if line is drawn left to right,
                                 -1 if drawn right to left */
{
   int DeltaYx2;
   int DeltaYx2MinusDeltaXx2;
   int ErrorTerm;

   /* Set up initial error term and values used inside drawing loop */
   DeltaYx2 = DeltaY * 2;
   DeltaYx2MinusDeltaXx2 = DeltaYx2 - (int) ( DeltaX * 2 );
   ErrorTerm = DeltaYx2 - (int) DeltaX;

   /* Draw the line */
   EVGADot(X0, Y0);              /* draw the first pixel */
   while ( DeltaX-- ) {
      /* See if it's time to advance the Y coordinate */
      if ( ErrorTerm >= 0 ) {
         /* Advance the Y coordinate & adjust the error term
            back down */
         Y0++;
         ErrorTerm += DeltaYx2MinusDeltaXx2;
      } else {
         /* Add to the error term */
         ErrorTerm += DeltaYx2;
      }
      X0 += XDirection;          /* advance the X coordinate */
      EVGADot(X0, Y0);           /* draw a pixel */
   }
}

/*
 * Draws a line in octant 1 or 2 ( |DeltaX| < DeltaY ).
 */
```

```
void Octant1(X0, Y0, DeltaX, DeltaY, XDirection)
unsigned int X0, Y0;          /* coordinates of start of the line */
unsigned int DeltaX, DeltaY;  /* length of the line (both > 0) */
int XDirection;               /* 1 if line is drawn left to right,
                                 -1 if drawn right to left */
{
   int DeltaXx2;
   int DeltaXx2MinusDeltaYx2;
   int ErrorTerm;

   /* Set up initial error term and values used inside drawing loop */
   DeltaXx2 = DeltaX * 2;
   DeltaXx2MinusDeltaYx2 = DeltaXx2 - (int) ( DeltaY * 2 );
   ErrorTerm = DeltaXx2 - (int) DeltaY;

   EVGADot(X0, Y0);           /* draw the first pixel */
   while ( DeltaY-- ) {
      /* See if it's time to advance the X coordinate */
      if ( ErrorTerm >= 0 ) {
         /* Advance the X coordinate & adjust the error term
            back down */
         X0 += XDirection;
         ErrorTerm += DeltaXx2MinusDeltaYx2;
      } else {
         /* Add to the error term */
         ErrorTerm += DeltaXx2;
      }
      Y0++;                   /* advance the Y coordinate */
      EVGADot(X0, Y0);        /* draw a pixel */
   }
}

/*
 * Draws a line on the EGA or VGA.
 */
void EVGALine(X0, Y0, X1, Y1, Color)
int X0, Y0;    /* coordinates of one end of the line */
int X1, Y1;    /* coordinates of the other end of the line */
char Color;    /* color to draw line in */
{
   int DeltaX, DeltaY;
   int Temp;

   /* Set the drawing color */

   /* Put the drawing color in the Set/Reset register */
   outportb(GC_INDEX, SET_RESET_INDEX);
   outportb(GC_DATA, Color);
   /* Cause all planes to be forced to the Set/Reset color */
   outportb(GC_INDEX, ENABLE_SET_RESET_INDEX);
   outportb(GC_DATA, 0xF);

   /* Save half the line-drawing cases by swapping Y0 with Y1
      and X0 with X1 if Y0 is greater than Y1. As a result, DeltaY
      is always > 0, and only the octant 0-3 cases need to be
      handled. */
   if ( Y0 > Y1 ) {
      Temp = Y0;
      Y0 = Y1;
      Y1 = Temp;
      Temp = X0;
      X0 = X1;
      X1 = Temp;
   }

   /* Handle as four separate cases, for the four octants in which
      Y1 is greater than Y0 */
   DeltaX = X1 - X0;    /* calculate the length of the line
                           in each coordinate */
   DeltaY = Y1 - Y0;
   if ( DeltaX > 0 ) {
      if ( DeltaX > DeltaY ) {
         Octant0(X0, Y0, DeltaX, DeltaY, 1);
      } else {
         Octant1(X0, Y0, DeltaX, DeltaY, 1);
      }
   } else {
      DeltaX = -DeltaX;            /* absolute value of DeltaX */
      if ( DeltaX > DeltaY ) {
         Octant0(X0, Y0, DeltaX, DeltaY, -1);
      } else {
         Octant1(X0, Y0, DeltaX, DeltaY, -1);
      }
   }

   /* Return the state of the EGA/VGA to normal */
   outportb(GC_INDEX, ENABLE_SET_RESET_INDEX);
   outportb(GC_DATA, 0);
   outportb(GC_INDEX, BIT_MASK_INDEX);
   outportb(GC_DATA, 0xFF);
}
```

LISTING 35.2 L35-2.C

```
/*
 * Sample program to illustrate EGA/VGA line drawing routines.
 *
 * Compiled with Borland C++
 *
 * By Michael Abrash
 */

#include <dos.h>     /* contains geninterrupt */

#define GRAPHICS_MODE   0x10
#define TEXT_MODE       0x03
#define BIOS_VIDEO_INT  0x10
#define X_MAX           640      /* working screen width */
#define Y_MAX           348      /* working screen height */

extern void EVGALine();

/*
 * Subroutine to draw a rectangle full of vectors, of the specified
 * length and color, around the specified rectangle center.
 */
void VectorsUp(XCenter, YCenter, XLength, YLength, Color)
int XCenter, YCenter;   /* center of rectangle to fill */
int XLength, YLength;   /* distance from center to edge
                           of rectangle */
int Color;              /* color to draw lines in */
{
   int WorkingX, WorkingY;
   /* Lines from center to top of rectangle */
   WorkingX = XCenter - XLength;
   WorkingY = YCenter - YLength;
   for ( ; WorkingX < ( XCenter + XLength ); WorkingX++ )
      EVGALine(XCenter, YCenter, WorkingX, WorkingY, Color);

   /* Lines from center to right of rectangle */
   WorkingX = XCenter + XLength - 1;
   WorkingY = YCenter - YLength;
   for ( ; WorkingY < ( YCenter + YLength ); WorkingY++ )
      EVGALine(XCenter, YCenter, WorkingX, WorkingY, Color);

   /* Lines from center to bottom of rectangle */
   WorkingX = XCenter + XLength - 1;
   WorkingY = YCenter + YLength - 1;
   for ( ; WorkingX >= ( XCenter - XLength ); WorkingX-- )
      EVGALine(XCenter, YCenter, WorkingX, WorkingY, Color);

   /* Lines from center to left of rectangle */
   WorkingX = XCenter - XLength;
   WorkingY = YCenter + YLength - 1;
   for ( ; WorkingY >= ( YCenter - YLength ); WorkingY-- )
      EVGALine(XCenter, YCenter, WorkingX, WorkingY, Color );
}

/*
 * Sample program to draw four rectangles full of lines.
 */
void main()
{
   char temp;

   /* Set graphics mode */
   _AX = GRAPHICS_MODE;
   geninterrupt(BIOS_VIDEO_INT);

   /* Draw each of four rectangles full of vectors */
   VectorsUp(X_MAX / 4, Y_MAX / 4, X_MAX / 4,
      Y_MAX / 4, 1);
   VectorsUp(X_MAX * 3 / 4, Y_MAX / 4, X_MAX / 4,
      Y_MAX / 4, 2);
   VectorsUp(X_MAX / 4, Y_MAX * 3 / 4, X_MAX / 4,
      Y_MAX / 4, 3);
   VectorsUp(X_MAX * 3 / 4, Y_MAX * 3 / 4, X_MAX / 4,
      Y_MAX / 4, 4);

   /* Wait for the enter key to be pressed */
   scanf("%c", &temp);

   /* Return back to text mode */
   _AX = TEXT_MODE;
   geninterrupt(BIOS_VIDEO_INT);
}
```

Looking at EVGALine

The **EVGALine** function itself performs four operations. **EVGALine** first sets up the VGA's hardware so that all pixels drawn will be in the desired color. This is accomplished by setting two of the VGA's registers, the Enable Set/Reset register and the

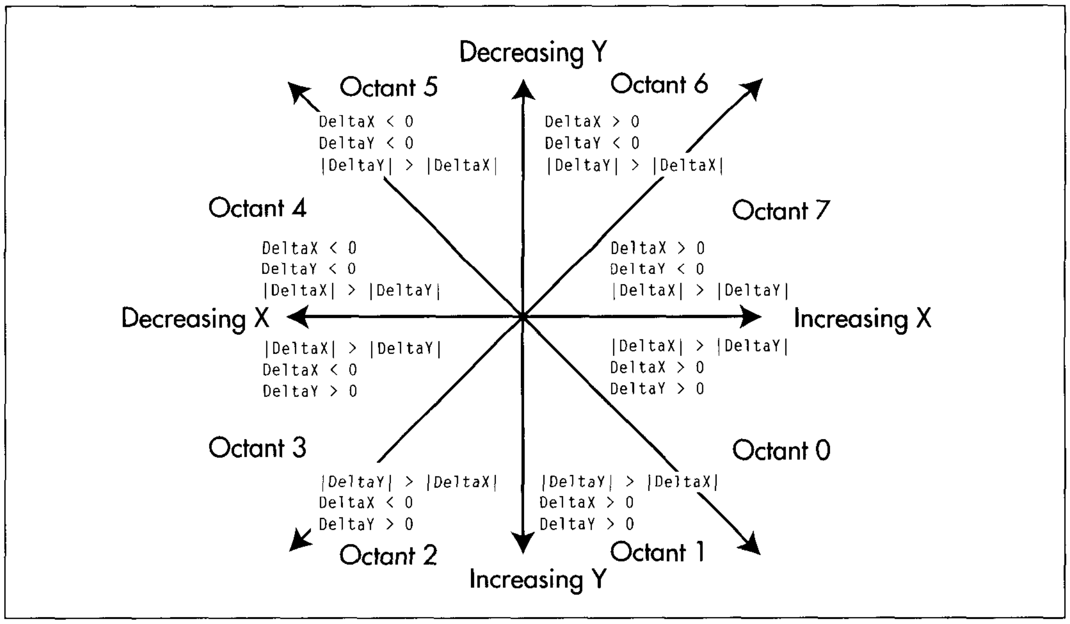
Set/Reset register. Setting the Enable Set/Reset to the value 0FH, as is done in **EVGALine**, causes all drawing to produce pixels in the color contained in the Set/Reset register. Setting the Set/Reset register to the passed color, in conjunction with the Enable Set/Reset setting of 0FH, causes all drawing done by **EVGALine** and the functions it calls to generate the passed color. In summary, setting up the Enable Set/Reset and Set/Reset registers in this way causes the remainder of **EVGALine** to draw a line in the specified color.

EVGALine next performs a simple check to cut in half the number of line orientations that must be handled separately. Figure 35.4 shows the eight possible line orientations among which a Bresenham's algorithm implementation must distinguish. (In interpreting Figure 35.4, assume that lines radiate outward from the center of the figure, falling into one of eight octants delineated by the horizontal and vertical axes and the two diagonals.) The need to categorize lines into these octants falls out of the major/minor axis nature of the algorithm; the orientations are distinguished by which coordinate forms the major axis and by whether each of X and Y increases or decreases from the line start to the line end.

A moment of thought will show, however, that four of the line orientations are redundant. Each of the four orientations for which ***DeltaY****, the Y component of the line, is less than 0 (that is, for which the line start Y coordinate is greater than the line end Y coordinate) can be transformed into one of the four orientations for which the line start Y coordinate is less than the line end Y coordinate simply by reversing the line start and end coordinates, so that the line is drawn in the other direction.* ***EVGALine*** *does this by swapping (X0,Y0) (the line start coordinates) with (X1,Y1) (the line end coordinates) whenever Y0 is greater than Y1.*

This accomplished, **EVGALine** must still distinguish among the four remaining line orientations. Those four orientations form two major categories, orientations for which the X dimension is the major axis of the line and orientations for which the Y dimension is the major axis. As shown in Figure 35.4, octants 1 (where X increases from start to finish) and 2 (where X decreases from start to finish) fall into the latter category, and differ in only one respect, the direction in which the X coordinate moves when it changes. Handling of the running error of the line is exactly the same for both cases, as one would expect given the symmetry of lines differing only in the sign of **DeltaX**, the X coordinate of the line. Consequently, for those cases where **DeltaX** is less than zero, the direction of X movement is made negative, and the absolute value of **DeltaX** is used for error term calculations.

Similarly, octants 0 (where X increases from start to finish) and 3 (where X decreases from start to finish) differ only in the direction in which the X coordinate moves when it changes. The difference between line drawing in octants 0 and 3 and line drawing in octants 1 and 2 is that in octants 0 and 3, since X is the major axis, the X coordinate changes on every pixel of the line and the Y coordinate changes only



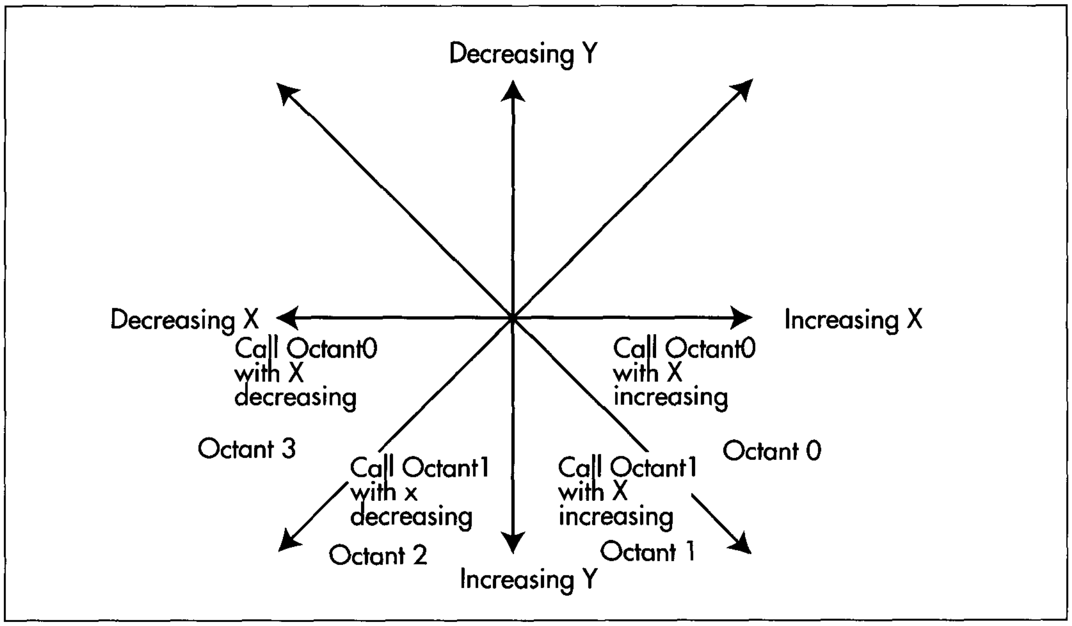
Bresenham's eight possible line orientations.
Figure 35.4

when the running error of the line dictates. In octants 1 and 2, the Y coordinate changes on every pixel and the X coordinate changes only when the running error dictates, since Y is the major axis.

There is one line-drawing function for octants 0 and 3, **Octant0**, and one line-drawing function for octants 1 and 2, **Octant1**. A single function with **if** statements could certainly be used to handle all four octants, but at a significant performance cost. There is, on the other hand, very little performance cost to grouping octants 0 and 3 together and octants 1 and 2 together, since the two octants in each pair differ only in the direction of change of the X coordinate.

EVGALine determines which line-drawing function to call and with what value for the direction of change of the X coordinate based on two criteria: whether **DeltaX** is negative or not, and whether the absolute value of **DeltaX** (|**DeltaX**|) is less than **DeltaY** or not, as shown in Figure 35.5. Recall that the value of **DeltaY**, and hence the direction of change of the Y coordinate, is guaranteed to be non-negative as a result of the earlier elimination of four of the line orientations.

After calling the appropriate function to draw the line (more on those functions shortly), **EVGALine** restores the state of the Enable Set/Reset register to its default of zero. In this state, the Set/Reset register has no effect, so it is not necessary to restore the state of the Set/Reset register as well. **EVGALine** also restores the state of



Decreasing Y
Decreasing X
Increasing X
Call Octant0 with X decreasing
Call Octant0 with X increasing
Octant 3
Octant 0
Call Octant1 with x decreasing
Call Octant1 with X increasing
Octant 2
Octant 1
Increasing Y

EVGALine's decision logic.
Figure 35.5

the Bit Mask register (which, as we will see, is modified by **EVGADot**, the pixel-drawing routine actually used to draw each pixel of the lines produced by **EVGALine**) to its default of 0FFH. While it would be more modular to have **EVGADot** restore the state of the Bit Mask register after drawing each pixel, it would also be considerably slower to do so. The same could be said of having **EVGADot** set the Enable Set/Reset and Set/Reset registers for each pixel: While modularity would improve, speed would suffer markedly.

Drawing Each Line

The **Octant0** and **Octant1** functions draw lines for which |**DeltaX**| is greater than **DeltaY** and lines for which |**DeltaX**| is less than or equal to **DeltaY**, respectively. The parameters to **Octant0** and **Octant1** are the starting point of the line, the length of the line in each dimension, and **XDirection**, the amount by which the X coordinate should be changed when it moves. **XDirection** must be either 1 (to draw toward the right edge of the screen) or –1 (to draw toward the left edge of the screen). No value is required for the amount by which the Y coordinate should be changed; since **DeltaY** is guaranteed to be positive, the Y coordinate always changes by 1 pixel.

Octant0 draws lines for which |**DeltaX**| is greater than **DeltaY**. For such lines, the X coordinate of each pixel drawn differs from the previous pixel by either 1 or –1, depending on the value of **XDirection**. (This makes it possible for **Octant0** to draw lines in both octant 0 and octant 3.) Whenever **ErrorTerm** becomes non-negative, indicating that the next Y coordinate is a better approximation of the line being drawn, the Y coordinate is increased by 1.

Octant1 draws lines for which |**DeltaX**| is less than or equal to **DeltaY**. For these lines, the Y coordinate of each pixel drawn is 1 greater than the Y coordinate of the previous pixel. Whenever **ErrorTerm** becomes non-negative, indicating that the next X coordinate is a better approximation of the line being drawn, the X coordinate is advanced by either 1 or –1, depending on the value of **XDirection**. (This makes it possible for **Octant1** to draw lines in both octant 1 and octant 2.)

Drawing Each Pixel

At the core of **Octant0** and **Octant1** is a pixel-drawing function, **EVGADot**. **EVGADot** draws a pixel at the specified coordinates in whatever color the hardware of the VGA happens to be set up for. As described earlier, since the entire line drawn by **EVGALine** is of the same color, line-drawing performance is improved by setting the VGA's hardware up once in **EVGALine** before the line is drawn, and then drawing all the pixels in the line in the same color via **EVGADot**.

EVGADot makes certain assumptions about the screen. First, it assumes that the address of the byte controlling the pixels at the start of a given row on the screen is 80 bytes after the start of the row immediately above it. In other words, this implementation of **EVGADot** only works for screens configured to be 80 bytes wide. Since this is the standard configuration of all of the modes **EVGALine** is designed to work in, the assumption of 80 bytes per row should be no problem. If it is a problem, however, **EVGADot** could easily be modified to retrieve the BIOS integer variable at address 0040:004A, which contains the number of bytes per row for the current video mode.

Second, **EVGADot** assumes that screen memory is organized as a linear bitmap starting at address A000:0000, with the pixel at the upper left of the screen controlled by bit 7 of the byte at offset 0, the next pixel to the right controlled by bit 6, the ninth pixel controlled by bit 7 of the byte at offset 1, and so on. Further, it assumes that the graphics adapter's hardware is configured such that setting the Bit Mask register to allow modification of only the bit controlling the pixel of interest and then ORing a value of 0FEH with display memory will draw that pixel correctly without affecting any other dots. (Note that 0FEH is used rather than 0FFH or 0 because some optimizing compilers turn ORs with the latter values into simpler operations or optimize them away entirely. As explained later, however, it's not the value that's ORed that matters, given the way we've set up the VGA's hardware; it's the act of ORing itself, and the value 0FEH forces the compiler to perform the OR operation.) Again, this is the normal way in which modes 0EH, 0FH, 10H, and 12H operate. As described earlier, **EVGADot** also assumes that the VGA is set up so that each pixel drawn in the above-mentioned manner will be drawn in the correct color.

Given those assumptions, **EVGADot** becomes a surprisingly simple function. First, **EVGADot** builds a far pointer that points to the byte of display memory controlling the pixel to be drawn. Second, a mask is generated consisting of zeros for all bits except the bit controlling the pixel to be drawn. Third, the Bit Mask register is set to that mask, so that when display memory is read and then written, all bits except the one that controls the pixel to be drawn will be left unmodified.

Finally, 0FEH is ORed with the display memory byte controlling the pixel to be drawn. ORing with 0FEH first reads display memory, thereby loading the VGA's internal latches with the contents of the display memory byte controlling the pixel to be drawn, and then writes to display memory with the value 0FEH. Because of the unusual way in which the VGA's data paths work and the way in which **EVGALine** sets up the VGA's Enable Set/Reset and Set/Reset registers, the value that is written by the **OR** instruction is ignored. Instead, the value that actually gets placed in display memory is the color that was passed to **EVGALine** and placed in the Set/Reset register. The Bit Mask register, which was set up in step three above, allows only the single bit controlling the pixel to be drawn to be set to this color value. For more on the various machineries the VGA brings to bear on graphics data, look back to Chapter 25.

The result of all this is simply a single pixel drawn in the color set up in **EVGALine**. **EVGADot** may seem excessively complex for a function that does nothing more that draw one pixel, but programming the VGA isn't trivial (as we've seen in the early chapters of this part). Besides, while the explanation of **EVGADot** is lengthy, the code itself is only five lines long.

Line drawing would be somewhat faster if the code of **EVGADot** were made an inline part of **Octant0** and **Octant1**, thereby saving the overhead of preparing parameters and calling the function. Feel free to do this if you wish; I maintained **EVGADot** as a separate function for clarity and for ease of inserting a pixel-drawing function for a different graphics adapter, should that be desired. If you do install a pixel-drawing function for a different adapter, or a fundamentally different mode such as a 256-color SuperVGA mode, remember to remove the hardware-dependent **outportb** lines in **EVGALine** itself.

Comments on the C Implementation

EVGALine does no error checking whatsoever. My assumption in writing **EVGALine** was that it would be ultimately used as the lowest-level primitive of a graphics software package, with operations such as error checking and clipping performed at a higher level. Similarly, **EVGALine** is tied to the VGA's screen coordinate system of (0,0) to (639,199) (in mode 0EH), (0,0) to (639,349) (in modes 0FH and 10H), or (0,0) to (639,479) (in mode 12H), with the upper left corner considered to be (0,0). Again, transformation from any coordinate system to the coordinate system used by **EVGALine** can be performed at a higher level. **EVGALine** is specifically designed to do one thing: draw lines into the display memory of the VGA. Additional functionality can be supplied by the code that calls **EVGALine**.

The version of **EVGALine** shown in Listing 35.1 is reasonably fast, but it is not as fast as it might be. Inclusion of **EVGADot** directly into **Octant0** and **Octant1**, and, indeed, inclusion of **Octant0** and **Octant1** directly into **EVGALine** would speed execution by saving the overhead of calling and parameter passing. Handpicked register variables might speed performance as well, as would the use of word **OUT**s rather than byte **OUT**s. A more significant performance increase would come from eliminating separate calculation of the address and mask for each pixel. Since the location of each pixel relative to the previous pixel is known, the address and mask could simply be adjusted from one pixel to the next, rather than recalculated from scratch.

These enhancements are not incorporated into the code in Listing 35.1 for a couple of reasons. One reason is that it's important that the workings of the algorithm be clearly visible in the code, for learning purposes. Once the implementation is understood, rewriting it for improved performance would certainly be a worthwhile exercise. Another reason is that when flat-out speed is needed, assembly language is the best way to go. Why produce hard-to-understand C code to boost speed a bit when assembly-language code can perform the same task at two or more times the speed?

Given which, a high-speed assembly language version of **EVGALine** would seem to be a logical next step.

Bresenham's Algorithm in Assembly

Listing 35.3 is a high-performance implementation of Bresenham's algorithm, written entirely in assembly language. The code is callable from C just as is Listing 35.1, with the same name, **EVGALine**, and with the same parameters. Either of the two can be linked to any program that calls **EVGALine**, since they appear to be identical to the calling program. The only difference between the two versions is that the sample program in Listing 35.2 runs over three times as fast on a 486 with an ISA-bus VGA when calling the assembly-language version of **EVGALine** as when calling the C version, and the difference would be considerably greater yet on a local bus, or with the use of write mode 3. Link each version with Listing 35.2 and compare performance—the difference is startling.

LISTING 35.3 L35-3.ASM

```
; Fast assembler implementation of Bresenham's line-drawing algorithm
; for the EGA and VGA. Works in modes 0Eh, 0Fh, 10h, and 12h.
; Borland C++ near-callable.
; Bit mask accumulation technique when |DeltaX| >= |DeltaY|
;  suggested by Jim Mackraz.
;
; Assembled with TASM
;
; By Michael Abrash
;
```

```
;******************************************************************
; C-compatible line-drawing entry point at _EVGALine.            •
; Near C-callable as:                                             *
;       EVGALine(X0, Y0, X1, Y1, Color);                          *
;******************************************************************
;

     model small
     .code

;
; Equates.
;
EVGA_SCREEN_WIDTH_IN_BYTES   equ  80          ;memory offset from start of
                                              ; one row to start of next
                                              ; in display memory
EVGA_SCREEN_SEGMENT          equ   0a000h     ;display memory segment
GC_INDEX                     equ   3ceh       ;Graphics Controller
                                              ; Index register port
SET_RESET_INDEX              equ   0          ;indexes of needed
ENABLE_SET_RESET_INDEX       equ   1          ; Graphics Controller
BIT_MASK_INDEX               equ   8          ; registers

;
; Stack frame.
;
EVGALineParms   struc
                dw      ?               ;pushed BP
                dw      ?               ;pushed return address (make double
                                        ; word for far call)
X0              dw      ?               ;starting X coordinate of line
Y0              dw      ?               ;starting Y coordinate of line
X1              dw      ?               ;ending X coordinate of line
Y1              dw      ?               ;ending Y coordinate of line
Color           db      ?               ;color of line
                db      ?               ;dummy to pad to word size
EVGALineParms   ends

;******************************************************************
; Line drawing macros.                                            *
;******************************************************************

;
; Macro to loop through length of line, drawing each pixel in turn.
; Used for case of |DeltaX| >= |DeltaY|.
; Input:
;       MOVE_LEFT: 1 if DeltaX < 0, 0 else
;       AL: pixel mask for initial pixel
;       BX: |DeltaX|
;       DX: address of GC data register, with index register set to
;               index of Bit Mask register
;       SI: DeltaY
;       ES:DI: display memory address of byte containing initial
;               pixel
;
LINE1   macro   MOVE_LEFT
        local   LineLoop, MoveXCoord, NextPixel, Line1End
        local   MoveToNextByte, ResetBitMaskAccumulator
        mov     cx,bx                   ;# of pixels in line
        jcxz    Line1End                ;done if there are no more pixels
                                        ; (there's always at least the one pixel
                                        ; at the start location)
        shl     si,1                    ;DeltaY * 2
        mov     bp,si                   ;error term
        sub     bp,bx                   ;error term starts at DeltaY * 2 - DeltaX
        shl     bx,1                    ;DeltaX * 2
        sub     si,bx                   ;DeltaY * 2 - DeltaX * 2 (used in loop)
        add     bx,si                   ;DeltaY * 2 (used in loop)
        mov     ah,al                   ;set aside pixel mask for initial pixel
                                        ; with AL (the pixel mask accumulator) set
                                        ; for the initial pixel
LineLoop:
;
; See if it's time to advance the Y coordinate yet.
;
        and     bp,bp                   ;see if error term is negative
        js      MoveXCoord              ;yes, stay at the same Y coordinate
;
; Advance the Y coordinate, first writing all pixels in the current
; byte, then move the pixel mask either left or right, depending
; on MOVE_LEFT.
;
        out     dx,al                   ;set up bit mask for pixels in this byte
        xchg    byte ptr [di],al
                                        ;load latches and write pixels, with bit mask
                                        ; preserving other latched bits. Because
                                        ; set/reset is enabled for all planes, the
                                        ; value written actually doesn't matter
        add     di,EVGA_SCREEN_WIDTH_IN_BYTES   ;increment Y coordinate
        add     bp,si                   ;adjust error term back down
;
; Move pixel mask one pixel (either right or left, depending
; on MOVE_LEFT), adjusting display memory address when pixel mask wraps.
;
if MOVE_LEFT
        rol     ah,1                    ;move pixel mask 1 pixel to the left
else
        ror     ah,1                    ;move pixel mask 1 pixel to the right
endif
        jnc     ResetBitMaskAccumulator ;didn't wrap to next byte
        jmp     short MoveToNextByte    ;did wrap to next byte
;
; Move pixel mask one pixel (either right or left, depending
; on MOVE_LEFT), adjusting display memory address and writing pixels
; in this byte when pixel mask wraps.
;
MoveXCoord:
        add     bp,bx           ;increment error term & keep same
if MOVE_LEFT
        rol     ah,1            ;move pixel mask 1 pixel to the left
else
        ror     ah,1            ;move pixel mask 1 pixel to the right
endif
        jnc     NextPixel       ;if still in same byte, no need to
                                ; modify display memory yet
        out     dx,al           ;set up bit mask for pixels in this byte.
        xchg    byte ptr [di],al
```

```
                                ;load latches and write pixels, with bit mask
                                ; preserving other latched bits. Because
                                ; set/reset is enabled for all planes, the
                                ; value written actually doesn't matter
MoveToNextByte:
if MOVE_LEFT
        dec     di              ;next pixel is in byte to left
else
        inc     di              ;next pixel is in byte to right
endif
ResetBitMaskAccumulator:
        sub     al,al           ;reset pixel mask accumulator
NextPixel:
        or      al,ah           ;add the next pixel to the pixel mask
                                ; accumulator
        loop    LineLoop
;
; Write the pixels in the final byte.
;
Line1End:
        out     dx,al           ;set up bit mask for pixels in this byte
        xchg    byte ptr [di],al
                                ;load latches and write pixels, with bit mask
                                ; preserving other latched bits. Because
                                ; set/reset is enabled for all planes, the
                                ; value written actually doesn't matter
        endm

;
; Macro to loop through length of line, drawing each pixel in turn.
; Used for case of DeltaX < DeltaY.
; Input:
;       MOVE_LEFT: 1 if DeltaX < 0, 0 else
;       AL: pixel mask for initial pixel
;       BX: |DeltaX|
;       DX: address of GC data register, with index register set to
;               index of Bit Mask register
;       SI: DeltaY
;       ES:DI: display memory address of byte containing initial
;               pixel
;
LINE2   macro   MOVE_LEFT
        local   LineLoop, MoveYCoord, ETermAction, Line2End
        mov     cx,si           ;# of pixels in line
        jcxz    Line2End        ;done if there are no more pixels
        shl     bx,1            ;DeltaX * 2
        mov     bp,bx           ;error term
        sub     bp,si           ;error term starts at DeltaX * 2 - DeltaY
        shl     si,1            ;DeltaY * 2
        sub     bx,si           ;DeltaX * 2 - DeltaY * 2 (used in loop)
        add     si,bx           ;DeltaX * 2 (used in loop)
;
; Set up initial bit mask & write initial pixel.
;
        out     dx,al
        xchg    byte ptr [di],ah
                                ;load latches and write pixel, with bit mask
                                ; preserving other latched bits. Because
                                ; set/reset is enabled for all planes, the
                                ; value written actually doesn't matter
```

```
LineLoop:
;
; See if it's time to advance the X coordinate yet.
;
        and     bp,bp                   ;see if error term is negative
        jns     ETermAction             ;no, advance X coordinate
        add     bp,si                   ;increment error term & keep same
        jmp     short MoveYCoord        ; X coordinate
ETermAction:
;
; Move pixel mask one pixel (either right or left, depending
; on MOVE_LEFT), adjusting display memory address when pixel mask wraps.
;
if MOVE_LEFT
        rol     al,1
        sbb     di,0
else
        ror     al,1
        adc     di,0
endif
        out     dx,al                   ;set new bit mask
        add     bp,bx                   ;adjust error term back down
;
; Advance Y coordinate.
;
MoveYCoord:
        add     di,EVGA_SCREEN_WIDTH_IN_BYTES
;
; Write the next pixel.
;
        xchg    byte ptr [di],ah
                                        ;load latches and write pixel, with bit mask
                                        ; preserving other latched bits. Because
                                        ; set/reset is enabled for all planes, the
                                        ; value written actually doesn't matter
;
        loop    LineLoop
Line2End:
        endm

;****************************************************************
; Line drawing routine.                                         *
;****************************************************************

        public  _EVGALine
_EVGALine       proc    near
        push    bp
        mov     bp,sp
        push    si                      ;preserve register variables
        push    di
        push    ds
;
; Point DS to display memory.
;
        mov     ax,EVGA_SCREEN_SEGMENT
        mov     ds,ax
;
; Set the Set/Reset and Set/Reset Enable registers for
; the selected color.
;
```

```
        mov     dx,GC_INDEX
        mov     al,SET_RESET_INDEX
        out     dx,al
        inc     dx
        mov     al,[bp+Color]
        out     dx,al
        dec     dx
        mov     al,ENABLE_SET_RESET_INDEX
        out     dx,al
        inc     dx
        mov     al,0ffh
        out     dx,al
;
; Get DeltaY.
;
        mov     si,[bp+Y1]              ;line Y start
        mov     ax,[bp+Y0]              ;line Y end, used later in
                                        ;calculating the start address
        sub     si,ax                   ;calculate DeltaY
        jns     CalcStartAddress        ;if positive, we're set
;
; DeltaY is negative -- swap coordinates so we're always working
; with a positive DeltaY.
;
        mov     ax,[bp+Y1]              ;set line start to Y1, for use
                                        ; in calculating the start address
        mov     dx,[bp+X0]
        xchg    dx,[bp+X1]
        mov     [bp+X0],dx              ;swap X coordinates
        neg     si                      ;convert to positive DeltaY
;
; Calculate the starting address in display memory of the line.
; Hardwired for a screen width of 80 bytes.
;
CalcStartAddress:
        shl     ax,1                    ;Y0 * 2 ;Y0 is already in AX
        shl     ax,1                    ;Y0 * 4
        shl     ax,1                    ;Y0 * 8
        shl     ax,1                    ;Y0 * 16
        mov     di,ax
        shl     ax,1                    ;Y0 * 32
        shl     ax,1                    ;Y0 * 64
        add     di,ax                   ;Y0 * 80
        mov     dx,[bp+X0]
        mov     cl,dl                   ;set aside lower 3 bits of column for
        and     cl,7                    ; pixel masking
        shr     dx,1
        shr     dx,1
        shr     dx,1                    ;get byte address of column (X0/8)
        add     di,dx                   ;offset of line start in display segment
;
; Set up GC Index register to point to the Bit Mask register.
;
        mov     dx,GC_INDEX
        mov     al,BIT_MASK_INDEX
        out     dx,al
        inc     dx                      ;leave DX pointing to the GC Data register
;
; Set up pixel mask (in-byte pixel address).
;
```

```
        mov     al,80h
        shr     al,cl
;
; Calculate DeltaX.
;
        mov     bx,[bp+X1]
        sub     bx,[bp+X0]
;
; Handle correct one of four octants.
;
        js      NegDeltaX
        cmp     bx,si
        jb      Octant1
;
; DeltaX >= DeltaY >= 0.
;
        LINE1   0
        jmp     EVGALineDone
;
; DeltaY > DeltaX >= 0.
;
Octant1:
        LINE2   0
        jmp     short EVGALineDone
;
NegDeltaX:
        neg     bx      ;|DeltaX|
        cmp     bx,si
        jb      Octant2
;
; |DeltaX| >= DeltaY and DeltaX < 0.
;
        LINE1   1
        jmp     short EVGALineDone
;
; |DeltaX| < DeltaY and DeltaX < 0.
;
Octant2:
        LINE2   1
;
EVGALineDone:
;
; Restore EVGA state.
;
        mov     al,0ffh
        out     dx,al                   ;set Bit Mask register to 0ffh
        dec     dx
        mov     al,ENABLE_SET_RESET_INDEX
        out     dx,al
        inc     dx
        sub     al,al
        out     dx,al                   ;set Enable Set/Reset register to 0
;
        pop     ds
        pop     di
        pop     si
        pop     bp
        ret
_EVGALine       endp

        end
```

An explanation of the workings of the code in Listing 35.3 would be a lengthy one, and would be redundant since the basic operation of the code in Listing 35.3 is no different from that of the code in Listing 35.1, although the implementation is much changed due to the nature of assembly language and also due to designing for speed rather than for clarity. Given that you thoroughly understand the C implementation in Listing 35.1, the assembly language implementation in Listing 35.3, which is well-commented, should speak for itself.

One point I do want to make is that Listing 35.3 incorporates a clever notion for which credit is due Jim Mackraz, who described the notion in a letter written in response to an article I wrote long ago in the late and lamented *Programmer's Journal.* Jim's suggestion was that when drawing lines for which |**DeltaX**| is greater than |**DeltaY**|, bits set to 1 for each of the pixels controlled by a given byte can be accumulated in a register, rather than drawing each pixel individually. All the pixels controlled by that byte can then be drawn at once, with a single access to display memory, when all pixel processing associated with that byte has been completed. This approach can save many **OUT**s and many display memory reads and writes when drawing nearly-horizontal lines, and that's important because EGAs and VGAs hold the CPU up for a considerable period of time on each I/O operation and display memory access.

All too many PC programmers fall into the high-level-language trap of thinking that a good algorithm guarantees good performance. Not so: As our two implementations of Bresenham's algorithm graphically illustrate (pun not originally intended, but allowed to stand once recognized), truly great PC code requires both a good algorithm *and* a good assembly implementation. In Listing 35.3, we've got both—and my-oh-my, isn't it fun?