Chapter E

Ellipses That Rip

Chapter E

Optimizing Ellipse Drawing with a Draw List for Each Octant

Do you believe in coincidence? You don't have to, you know; there's no rule that says there has to be any such thing, and many people choose to believe otherwise. They think that all events are interrelated; what appears to be coincidence actually reflects deeper meaning in the universe. Personally, I prefer to believe in random coincidence, because if coincidences are meaningful, the universe has been trying to send me a *very* meaningful message lately—and while I can't imagine what that message could possibly be, I'm not sure I want to find out!

To wit: In the previous chapter, I began by telling you that I seemed to be encountering faster circle implementations almost daily, and I mentioned Hal Hardenbergh's extremely fast, 16-bit- integer-only circle-drawing approach. A few days after I wrote the original article from which that chapter was derived, in what was either one hell of a coincidence or a telegram from the Twilight Zone, an article by Tim Paterson in the July, 1990 *Dr. Dobb's Journal* crossed my desk. The topic of the article: drawing circles. Fast. With no multiplies and no divides and plain old 16-bit integers.

Sound familiar?

Tim covers ground that we've already been over, but he looks at circle drawing from a different and interesting perspective. Although the code is in C, you could convert it to assembly language easily enough. The message? One (which pointedly includes

that fellow I) should never be too sure that he or she has in fact nailed any topic for all time, from all perspectives.

With that lesson firmly in mind, let's draw some ellipses. Fast. *Very* fast, in fact. Nonetheless, I'm sure there's something faster yet; in my experience, there always is.

Ellipses, Continued

Last time, we learned how to draw ellipses without using any multiplies, divides, or floating-point numbers, thanks to an integer-only incremental algorithm. This time, I'll better attune that code to the hardware of the EGA and VGA by eliminating the separate calculation of the screen address for each and every point. Instead, I'll generate a *draw list* for an octant, that is, a set of commands to either advance or not advance along the minor axis each time drawing advances one pixel along the major axis. Then I'll draw the four symmetries of that draw list (the four octants which share the same major axis—that is, advance more rapidly along the same axis) one at a time, using specialized code that calculates the bit mask and offset for each pixel as a function of the last pixel, thereby eliminating all multiplications and multi-bit rotations. I'll then repeat the process for the four octants in which the other coordinate is the major axis. Finally, I'll rewrite in assembly language the code that generates draw lists, and I'll do the same for the code that draws octants from draw lists.

Ellipse Drawing Made Fast

Without further ado, let's get to the code. Listing E.1 is C ellipse-drawing code, differing from Listing D.4 in that Listing E.1 generates a draw list and draws the four symmetries of that list, then does the whole thing again for the other axis, as described above. That change produces a performance improvement of about 35 percent, as shown in Table E.1. Listing E.2 is a sample C program that can call any of the ellipse-drawing functions in this chapter and the last; Listing E.2 was used for timing the various implementations. (Because this chapter brings together a lot of familiar stuff, I made the sample program a little more interesting than the standard concentric ellipses. It's not a big deal—it's kind of a circle with pointy ears—but it gives you an idea of the sorts of shapes that are easily created with sets of ellipses; run it yourself and see.)

LISTING E.1 LE-1.C

```
/*
 * Draws an ellipse of the specified X and Y axis radii and color,
 * using a fast integer-only & square-root-free approach, and
 * generating the arc for one octant into a buffer, drawing four
 * symmetries from that buffer, then doing the same for the other
 * axis.
 * Compiles with either Borland or Microsoft.
 * VGA or EGA.
 */
#include <dos.h>
```

```
/* Handle differences between Borland and Microsoft. Note that Borland accepts
   outp as a synonym for outportb, but not outpw for outport */
#ifdef __TURBOC__
#define outpw  outport
#endif

#define SCREEN_WIDTH_IN_BYTES     80          /* # of bytes across one scan
                                                 line in mode 12h */
#define SCREEN_SEGMENT            0xA000      /* mode 12h display memory seg */
#define GC_INDEX                  0x3CE       /* Graphics Controller port */
#define SET_RESET_INDEX           0           /* Set/Reset reg index in GC */
#define SET_RESET_ENABLE_INDEX    1           /* Set/Reset Enable reg index
                                                 in GC */
#define BIT_MASK_INDEX            8           /* Bit Mask reg index in GC */

unsigned char PixList[SCREEN_WIDTH_IN_BYTES*8/2];
                                              /* maximum major axis length is
                                                 1/2 screen width, because we're
                                                 assuming no clipping is needed */

/* Draws the arc for an octant in which Y is the major axis. (X,Y) is the
   starting point of the arc. HorizontalMoveDirection selects whether the
   arc advances to the left or right horizontally (0=left, 1=right).
   RowOffset contains the offset in bytes from one scan line to the next,
   controlling whether the arc is drawn up or down. Length is the
   vertical length in pixels of the arc, and DrawList is a list
   containing 0 for each point if the next point is vertically aligned,
   and 1 if the next point is 1 pixel diagonally to the left or right. */
void DrawVOctant(int X, int Y, int Length, int RowOffset,
   int HorizontalMoveDirection, unsigned char *DrawList)
{
   unsigned char far *ScreenPtr, BitMask;

   /* Point to the byte the initial pixel is in. */
#ifdef __TURBOC__
   ScreenPtr = MK_FP(SCREEN_SEGMENT,
      (Y * SCREEN_WIDTH_IN_BYTES) + (X / 8));
#else
   FP_SEG(ScreenPtr) = SCREEN_SEGMENT;
   FP_OFF(ScreenPtr) =(Y * SCREEN_WIDTH_IN_BYTES) + (X / 8);
#endif
   /* Set the initial bit mask */
   BitMask = 0x80 >> (X & 0x07);

   /* Draw all points in DrawList */
   while ( Length-- ) {
      /* Set the bit mask for the pixel */
      outp(GC_INDEX + 1, BitMask);
      /* Draw the pixel. ORed to force read/write to load latches.
         Data written doesn't matter, because set/reset is enabled
         for all planes. Note: don't OR with 0; MSC optimizes that
         statement to no operation */
      *ScreenPtr |= 0xFE;
      /* Now advance to the next pixel based on DrawList. */
      if ( *DrawList++ ) {
         /* Advance horizontally to produce a diagonal move. Rotate
            the bit mask, advancing one byte horizontally if the bit
            mask wraps */
         if ( HorizontalMoveDirection == 1 ) {
            /* Move right */
```

```
            if ( (BitMask >>= 1) == 0 ) {
               BitMask = 0x80;   /* wrap the mask */
               ScreenPtr++;      /* advance 1 byte to the right */
            }
         } else {
            /* Move left */
            if ( (BitMask <<= 1) == 0 ) {
               BitMask = 0x01;   /* wrap the mask */
               ScreenPtr--;      /* advance 1 byte to the left */
            }
         }
      }
      ScreenPtr += RowOffset; /* advance to the next scan line */
   }
}

/* Draws the arc for an octant in which X is the major axis. (X,Y) is the
   starting point of the arc. HorizontalMoveDirection selects whether the
   arc advances to the left or right horizontally (0=left, 1=right).
   RowOffset contains the offset in bytes from one scan line to the next,
   controlling whether the arc is drawn up or down. Length is the
   horizontal length in pixels of the arc, and DrawList is a list
   containing 0 for each point if the next point is horizontally aligned,
   and 1 if the next point is 1 pixel above or below diagonally. */
void DrawHOctant(int X, int Y, int Length, int RowOffset,
   int HorizontalMoveDirection, unsigned char *DrawList)
{
   unsigned char far *ScreenPtr, BitMask;

   /* Point to the byte the initial pixel is in */
#ifdef __TURBOC__
   ScreenPtr = MK_FP(SCREEN_SEGMENT,
      (Y * SCREEN_WIDTH_IN_BYTES) + (X / 8));
#else
   FP_SEG(ScreenPtr) = SCREEN_SEGMENT;
   FP_OFF(ScreenPtr) =(Y * SCREEN_WIDTH_IN_BYTES) + (X / 8);
#endif
   /* Set the initial bit mask */
   BitMask = 0x80 >> (X & 0x07);

   /* Draw all points in DrawList */
   while ( Length-- ) {
      /* Set the bit mask for the pixel */
      outp(GC_INDEX + 1, BitMask);
      /* Draw the pixel (see comments above for details) */
      *ScreenPtr |= 0xFE;
      /* Now advance to the next pixel based on DrawList */
      if ( *DrawList++ ) {
         /* Advance vertically to produce a diagonal move */
         ScreenPtr += RowOffset; /* advance to the next scan line */
      }
      /* Advance horizontally. Rotate the bit mask, advancing one
         byte horizontally if the bit mask wraps */
      if ( HorizontalMoveDirection == 1 ) {
         /* Move right */
         if ( (BitMask >>= 1) == 0 ) {
            BitMask = 0x80;   /* wrap the mask */
            ScreenPtr++;      /* advance 1 byte to the right */
         }
      } else {
```

```
         /* Move left */
         if ( (BitMask <<= 1) == 0 ) {
            BitMask = 0x01;   /* wrap the mask */
            ScreenPtr--;      /* advance 1 byte to the left */
         }
      }
   }
}

/* Draws an ellipse of X axis radius A and Y axis radius B in
 * color Color centered at screen coordinate (X,Y). Radii must
 * both be non-zero. */
void DrawEllipse(int X, int Y, int A, int B, int Color) {
   int WorkingX, WorkingY;
   long Threshold;
   long ASquared = (long) A * A;
   long BSquared = (long) B * B;
   long XAdjust, YAdjust;
   unsigned char *PixListPtr;

   /* Set drawing color via set/reset */
   outpw(GC_INDEX, (0x0F << 8) | SET_RESET_ENABLE_INDEX);
                               /* enable set/reset for all planes */
   outpw(GC_INDEX, (Color << 8) | SET_RESET_INDEX);
                               /* set set/reset (drawing) color */
   outp(GC_INDEX, BIT_MASK_INDEX); /* leave the GC Index reg pointing
                                      to the Bit Mask reg */

   /* Draw the four symmetric arcs for which X advances faster (that is,
      for which X is the major axis) */

   /* Draw the four arcs; set draw parameters for initial point (0,B) */
   /* Calculate all points along an arc of 1/8th of the ellipse and
      store that info in PixList for later drawing */
   PixListPtr = PixList;
   WorkingX = 0;
   XAdjust = 0;
   YAdjust = ASquared * 2 * B;
   Threshold = ASquared / 4 - ASquared * B;

   for (;;) {
      /* Advance the threshold to the value for the next X point
         to be drawn */
      Threshold += XAdjust + BSquared;

      /* If the threshold has passed 0, then the Y coordinate has
         advanced more than halfway to the next pixel and it's time
         to advance the Y coordinate by 1 and set the next threshold
         accordingly */
      if ( Threshold >= 0 ) {
         YAdjust -= ASquared * 2;
         Threshold -= YAdjust;
         *PixListPtr++ = 1;   /* advance along both axes */
      } else {
         *PixListPtr++ = 0;   /* advance only along the X axis */
      }

      /* Advance the X coordinate by 1 */
      XAdjust += BSquared * 2;
      WorkingX++;
```

```
   /* Stop if X is no longer the major axis (the arc has passed the
      45-degree point) */
   if ( XAdjust >= YAdjust )
      break;
}

/* Now draw each of 4 symmetries of the octant in turn, the
   octants for which X is the major axis. Adjust every other arc
   so that there's no overlap. */
DrawHOctant(X,Y-B,WorkingX,SCREEN_WIDTH_IN_BYTES,0,PixList);
DrawHOctant(X+1,Y-B+(*PixList),WorkingX-1,SCREEN_WIDTH_IN_BYTES,1,
   PixList+1);
DrawHOctant(X,Y+B,WorkingX,-SCREEN_WIDTH_IN_BYTES,0,PixList);
DrawHOctant(X+1,Y+B-(*PixList),WorkingX-1,-SCREEN_WIDTH_IN_BYTES,1,
   PixList+1);

/* Draw the four symmetric arcs for which X advances faster (that is,
   for which Y is the major axis) */

/* Draw the four arcs; set draw parameters for initial point (A,0) */
/* Calculate all points along an arc of 1/8th of the ellipse and
   store that info in PixList for later drawing */
PixListPtr = PixList;
WorkingY = 0;
XAdjust = BSquared * 2 * A;
YAdjust = 0;
Threshold = BSquared / 4 - BSquared * A;

for (;;) {
   /* Advance the threshold to the value for the next Y point
      to be drawn */
   Threshold += YAdjust + ASquared;

   /* If the threshold has passed 0, then the X coordinate has
      advanced more than halfway to the next pixel and it's time
      to advance the X coordinate by 1 and set the next threhold
      accordingly */
   if ( Threshold >= 0 ) {
      XAdjust -= BSquared * 2;
      Threshold = Threshold - XAdjust;
      *PixListPtr++ = 1;   /* advance along both axes */
   } else {
      *PixListPtr++ = 0;   /* advance only along the X axis */
   }

   /* Advance the Y coordinate by 1 */
   YAdjust += ASquared * 2;
   WorkingY++;

   /* Stop if Y is no longer the major axis (the arc has passed the
      45-degree point) */
   if ( YAdjust > XAdjust )
      break;
}

/* Now draw each of 4 symmetries of the octant in turn, the
   octants for which Y is the major axis. Adjust every other arc
   so that there's no overlap. */
DrawVOctant(X-A,Y,WorkingY,-SCREEN_WIDTH_IN_BYTES,1,PixList);
DrawVOctant(X-A+(*PixList),Y+1,WorkingY-1,SCREEN_WIDTH_IN_BYTES,1,
   PixList+1);
```

```
   DrawVOctant(X+A,Y,WorkingY,-SCREEN_WIDTH_IN_BYTES,0,PixList);
   DrawVOctant(X+A-(*PixList),Y+1,WorkingY-1,SCREEN_WIDTH_IN_BYTES,0,
      PixList+1);

   /* Reset the Bit Mask register to normal */
   outp(GC_INDEX + 1, 0xFF);
   /* Turn off set/reset enable */
   outpw(GC_INDEX, (0x00 << 8) | SET_RESET_ENABLE_INDEX);
}
```

LISTING E.2 LE-2.C

```
/*
 * Draws ellipses of varying eccentricities in the VGA's hi-res mode,
 * mode 12h.
 * For VGA only.
 * Compile and link (using Borland C++) with listing LE-X.C (where X is 1 or 4)
 * with:
 *    bcc -ms LE-X.C  LE-2.C
 *
 */

#include <dos.h>
#include <stdio.h>

main() {
   int XRadius, YRadius, Temp, Color, i;
   union REGS Regs;

   /* Select VGA's hi-res 640x480 graphics mode, mode 12h */
   Regs.x.ax = 0x0012;
   int86(0x10, &Regs, &Regs);

   /* Repeat 10 times */
   for ( i = 0; i < 10; i++ ) {
      /* Draw nested ellipses */
      for ( XRadius = 319, YRadius = 1, Color = 7; YRadius < 240;
            XRadius -= 1, YRadius += 2 ) {
         DrawEllipse(640/2, 480/2, XRadius, YRadius, Color);
         Color = (Color + 1) & 0x0F;   /* cycle through 16 colors */
      }
   }

   /* Wait for a key, restore text mode, and done */
   scanf("%c", &Temp);
   Regs.x.ax = 0x0003;
   int86(0x10, &Regs, &Regs);
}
```

I am well aware that Listing E.1 is not fully optimized; for one thing, it doesn't take advantage of the write mode 3 and pixel-accumulation techniques used by the assembly code in Listing E.4. Listing E.1 is intended as an illustrative bridge between the standard ellipse-drawing code of the previous chapter and the fast but hard to understand code in Listing E.4, so I've leaned toward comprehension rather than maximum speed in Listing E.1. To my mind, it's wasted effort to spend time squeezing cycles out of Listing E.1 when Listings E.3 and E.4 will always be faster no matter how well-optimized Listing E.1 is.

Listing E.3 is the C portion of the final and fastest ellipse-drawing routine. Listing E.3 calls functions in the assembly language module shown in Listing E.4 in order to perform the two critical aspects of ellipse drawing: draw list generation and the actual octant drawing.

Listings E.3 and E.4 together are about twice as fast as Listing E.1. They are in the range of 2.5 to 3.5 times faster than Listing 19.4. Lastly, they are 60 or so times faster than the floating-point based implementation in Listing 19.1.

See what a little forethought and attention to detail can do?

LISTING E.3 LE-3.C

```
/*
 * Draws an ellipse of the specified X and Y axis radii and color,
 * using a fast integer-only & square-root-free approach, and
 * generating the arc for one octant into a buffer, drawing four
 * symmetries from that buffer, then doing the same for the other
 * axis. Uses assembly language for inner loops of octant generation
 * & drawing. Link to Listing E.4.
 *
 * Compiles with either Borland or Microsoft.
 * VGA or EGA.
 */
#define ISVGA  0              /* set to 1 to use VGA write mode 3*/
                              /* keep synchronized with Listing 4 */
#include <dos.h>

/* Handle differences between Borland and Microsift. Note that Borland accepts
   outp as a synonym for outportb, but not outpw for outport */
#ifdef __TURBOC__
#define outpw  outport
#endif

#define SCREEN_WIDTH_IN_BYTES     80          /* # of bytes across one scan
                                                 line in mode 12h */
#define SCREEN_SEGMENT            0xA000      /* mode 12h display memory seg */
#define GC_INDEX                  0x3CE       /* Graphics Controller port */
#define SET_RESET_INDEX           0           /* Set/Reset reg index in GC */
#define SET_RESET_ENABLE_INDEX    1           /* Set/Reset Enable reg index
                                                 in GC */
#define GC_MODE_INDEX             5           /* Graphics Mode reg index in GC */
#define COLOR_DONT_CARE           7           /* Color Don't Care reg index in GC */
#define BIT_MASK_INDEX            8           /* Bit Mask reg index in GC */

unsigned char PixList[SCREEN_WIDTH_IN_BYTES*8/2];
                                              /* maximum major axis length is
                                                 1/2 screen width, because we're
                                                 assuming no clipping is needed */

/* Draws an ellipse of X axis radius A and Y axis radius B in
 * color Color centered at screen coordinate (X,Y). Radii must
 * both be non-zero. */
void DrawEllipse(int X, int Y, int A, int B, int Color) {
   int Length;
   long Threshold;
   long ASquared = (long) A * A;
   long BSquared = (long) B * B;
   long XAdjust, YAdjust;
   unsigned char *PixListPtr, OriginalGCMode;
```

```
   /* Set drawing color via set/reset */
   outpw(GC_INDEX, (0x0F << 8) | SET_RESET_ENABLE_INDEX);
                                 /* enable set/reset for all planes */
   outpw(GC_INDEX, (Color << 8) | SET_RESET_INDEX);
                                 /* set set/reset (drawing) color */
#if ISVGA
   /* Remember original read/write mode & select
      read mode 1/write mode 3, with Color Don't Care
      set to ignore all planes and therefore always return 0xFF */
   outp(GC_INDEX, GC_MODE_INDEX);
   OriginalGCMode = inp(GC_INDEX + 1);
   outp(GC_INDEX+1, OriginalGCMode | 0x0B);
   outpw(GC_INDEX, (0x00 << 8) | COLOR_DONT_CARE);
   outpw(GC_INDEX, (0xFF << 8) | BIT_MASK_INDEX);
#else
   outp(GC_INDEX, BIT_MASK_INDEX); /* leave the GC Index reg pointing
                                      to the Bit Mask reg */
#endif

   /* Draw the four symmetric arcs for which X advances faster (that is,
      for which X is the major axis) */
   /* Generate the draw list for 1 octant */
   Length = GenerateEOctant(PixList, (long) ASquared * 2 * B,
      (long) ASquared / 4 - ASquared * B, ASquared, BSquared);

   /* Now draw each of 4 symmetries of the octant in turn, the
      octants for which X is the major axis. Adjust every other arc
      so that there's no overlap. */
   DrawHOctant(X,Y-B,Length,SCREEN_WIDTH_IN_BYTES,0,PixList);
   DrawHOctant(X+1,Y-B+(*PixList),Length-1,SCREEN_WIDTH_IN_BYTES,1,
      PixList+1);
   DrawHOctant(X,Y+B,Length,-SCREEN_WIDTH_IN_BYTES,0,PixList);
   DrawHOctant(X+1,Y+B-(*PixList),Length-1,-SCREEN_WIDTH_IN_BYTES,1,
      PixList+1);

   /* Draw the four symmetric arcs for which Y advances faster (that is,
      for which Y is the major axis) */
   /* Generate the draw list for 1 octant */
   Length = GenerateEOctant(PixList, (long) BSquared * 2 * A,
      (long) BSquared / 4 - BSquared * A, BSquared, ASquared);

   /* Now draw each of 4 symmetries of the octant in turn, the
      octants for which X is the major axis. Adjust every other arc
      so that there's no overlap. */
   DrawVOctant(X-A,Y,Length,-SCREEN_WIDTH_IN_BYTES,1,PixList);
   DrawVOctant(X-A+(*PixList),Y+1,Length-1,SCREEN_WIDTH_IN_BYTES,1,
      PixList+1);
   DrawVOctant(X+A,Y,Length,-SCREEN_WIDTH_IN_BYTES,0,PixList);
   DrawVOctant(X+A-(*PixList),Y+1,Length-1,SCREEN_WIDTH_IN_BYTES,0,
      PixList+1);

#if ISVGA
   /* Restore original write mode */
   outpw(GC_INDEX, (OriginalGCMode << 8) | GC_MODE_INDEX);
   /* Restore normal Color Don't Care setting */
   outpw(GC_INDEX, (0x0F << 8) | COLOR_DONT_CARE);
#else
   /* Reset the Bit Mask register to normal */
   outp(GC_INDEX + 1, 0xFF);
#endif
```

```
   /* Turn off set/reset enable */
   outpw(GC_INDEX, (0x00 << 8) | SET_RESET_ENABLE_INDEX);
}
```

LISTING E.4 LE-4.ASM

```
; Contains 3 C-callable routines: GenerateEOctant, DrawVOctant, and
;       DrawHOctant. See individual routines for comments. Link to
;       Listing E.3.
;
; Works with TASM or MASM
;
ISVGA   equ     0       ;set to 1 to use VGA write mode 3
                        ; keep synchronized with Listing 3
        .model small
        .code
;*************************************************************************
; Generates an octant of the specified ellipse, placing the results in
; PixList, with a 0 in PixList meaning draw pixel & move only along
; major axis, and a 1 in PixList meaning draw pixel & move along both
; axes.
; C near-callable as:
;  int GenerateEOctant(unsigned char *PixList, long MinorAdjust,
;       long Threshold, long MajorSquared, long MinorSquared);
;
; Return value = PixelCount (# of points)
;
; Passed parameters:
;
GenerateOctantParms     struc
        dw      ?       ;pushed BP
        dw      ?       ;return address pushed by call
PixList dw      ?       ;pointer to list to store draw control data in
MinorAdjust dd  ?       ;initially MajorAxis**2 * 2 * MinorAxis, used
                        ; to adjust threshold after minor axis move
Threshold dd    ?       ;initially MajorAxis**2 / 4 + MajorAxis**2 *
                        ; MinorAxis, used to determine when to advance
                        ; along the minor axis
MajorSquared dd ?       ;MajorAxis**2
MinorSquared dd ?       ;MinorAxis**2
GenerateOctantParms     ends
;
; Local variables (offsets relative to BP in stack frame):
;
PixelCount      equ     -2      ;running major axis coordinate
                                ; relative to center
MajorAdjust     equ     -6      ;used to adjust threshold after major
                                ; axis move
MajorSquaredTimes2 equ  -10     ;MajorSquared * 2
MinorSquaredTimes2 equ  -14     ;MinorSquared * 2
;
        public _GenerateEOctant
_GenerateEOctant proc    near
        push    bp              ;preserve caller's stack frame
        mov     bp,sp           ;point to our stack frame
        add     sp,MinorSquaredTimes2
                                ;allocate room for local vars
        push    si              ;preserve C register variables
        push    di
;Initialize local variables.
        mov     word ptr [bp+PixelCount],0 ;initialize count of pixels
                                           ; to zero
```

```
        mov     ax,word ptr [bp+MajorSquared] ;set MajorSquaredTimes2
        shl     ax,1                    ;lower word times 2
        mov     word ptr [bp+MajorSquaredTimes2],ax
        mov     ax,word ptr [bp+MajorSquared+2]
        rcl     ax,1                    ;upper word times 2
        mov     word ptr [bp+MajorSquaredTimes2+2],ax

        mov     ax,word ptr [bp+MinorSquared] ;set MinorSquaredTimes2
        shl     ax,1                    ;lower word times 2
        mov     word ptr [bp+MinorSquaredTimes2],ax
        mov     ax,word ptr [bp+MinorSquared+2]
        rcl     ax,1                    ;upper word times 2
        mov     word ptr [bp+MinorSquaredTimes2+2],ax
;Set up registers for loop.
        mov     di,[PixList+bp]         ;point DI to PixList
; Set MajorAdjust to 0.
        sub     cx,cx
        mov     si,cx                   ;SI:CX = MajorAdjust

        mov     bx,word ptr [bp+Threshold]      ;DX:BX = threshold
        mov     dx,word ptr [bp+Threshold+2]
; At this point:
;  DX:BX = threshold
;  SI:CX = MajorAdjust
;  DI = PixList pointer
GenLoop:
; Advance the threshold by MajorAdjust + MinorAxis**2.
        add     bx,cx
        adc     dx,si
        add     bx,word ptr [bp+MinorSquared]
        adc     dx,word ptr [bp+MinorSquared+2]
; If the threshold has passed 0, then the minor coordinate has
; advanced more than halfway to the next pixel and it's time to
; advance the minor coordinate by 1 and set the next threshold
; accordingly.
        mov     byte ptr [di],0 ;assume we won't move along the
                                ; minor axis
        js      MoveMajor       ;and, in fact, we won't move minor
; Minor coordinate has advanced.
; Adjust the minor axis adjust value.
        mov     ax,word ptr [bp+MajorSquaredTimes2]
        sub     word ptr [bp+MinorAdjust],ax
        mov     ax,word ptr [bp+MajorSquaredTimes2+2]
        sbb     word ptr [bp+MinorAdjust+2],ax
; Adjust the threshold for the minor axis move
        sub     bx,word ptr [bp+MinorAdjust]
        sbb     dx,word ptr [bp+MinorAdjust+2]
        mov     byte ptr [di],1
MoveMajor:
        inc     di
; Count this point.
        inc     word ptr [bp+PixelCount]
; Adjust the major adjust for the new point.
        add     cx,word ptr [bp+MinorSquaredTimes2]
        adc     si,word ptr [bp+MinorSquaredTimes2+2]
; Stop if the major axis has switched (the arc has passed the
; 45-degree point).
        cmp     si,word ptr [bp+MinorAdjust+2]
        ja      Done
        jb      GenLoop
```

```
        cmp     cx,word ptr [bp+MinorAdjust]
        jb      GenLoop
Done:
        mov     ax,[bp+PixelCount]      ;return # of points
        pop     di              ;restore C register variables
        pop     si
        mov     sp,bp           ;deallocate local vars
        pop     bp              ;restore caller's stack frame
        ret
_GenerateEOctant endp
;*************************************************************************
; Draws the arc for an octant in which Y is the major axis. (X,Y) is the
; starting point of the arc. HorizontalMoveDirection selects whether the
; arc advances to the left or right horizontally (0=left, 1=right).
; RowOffset contains the offset in bytes from one scan line to the next,
; controlling whether the arc is drawn up or down. DrawLength is the
; vertical length in pixels of the arc, and DrawList is a list
; containing 0 for each point if the next point is vertically aligned,
; and 1 if the next point is 1 pixel diagonally to the left or right.
;
; The Graphics Controller Index register must already point to the Bit
; Mask register.
;
; C near-callable as:
;  void DrawVOctant(int X, int Y, int DrawLength, int RowOffset,
;       int HorizontalMoveDirection, unsigned char *DrawList);
;
DrawParms       struc
        dw      ?       ;pushed BP
        dw      ?       ;return address
X       dw      ?       ;initial coordinates
Y       dw      ?
DrawLength dw   ?       ;vertical length
RowOffset dw    ?       ;distance from one scan line to the next
HorizontalMoveDirection dw ? ;1 to move right, 0 to move left
DrawList dw     ?       ;pointer to list containing 1 to draw
DrawParms       ends    ; diagonally, 0 to draw vertically for
                        ; each point
SCREEN_SEGMENT  equ     0a000h  ;display memory segment in mode 12h
SCREEN_WIDTH_IN_BYTES equ 80    ;distance from one scan line to next
GC_INDEX        equ     3ceh    ;GC Index register address
;
        public _DrawVOctant
_DrawVOctant    proc    near
        push    bp              ;preserve caller's stack frame
        mov     bp,sp           ;point to our stack frame
        push    si              ;preserve C register variables
        push    di
;Point ES:DI to the byte the initial pixel is in.
        mov     ax,SCREEN_SEGMENT
        mov     es,ax
        mov     ax,SCREEN_WIDTH_IN_BYTES
        mul     [bp+Y]  ;Y*SCREEN_WIDTH_IN_BYTES
        mov     di,[bp+X] ;X
        mov     cx,di   ;set X aside in CX
        shr     di,1
        shr     di,1
        shr     di,1    ;X/8
        add     di,ax   ;screen offset = Y*SCREEN_WIDTH_IN_BYTES+X/8
        and     cl,07h  ;X modulo 8
```

```
if ISVGA                ;--VGA--
        mov     ah,80h  ;keep VGA bit mask in AH
        shr     ah,cl   ;initial bit mask = 80h shr (X modulo 8);
        cld             ;for LODSB, used below
else                    ;--EGA--
        mov     al,80h  ;keep EGA bit mask in AL
        shr     al,cl   ;initial bit mask = 80h shr (X modulo 8);
        mov     dx,GC_INDEX+1 ;point DX to GC Data reg/bit mask
endif                   ;--------
        mov     si,[bp+DrawList] ;SI points to list to draw from
        sub     bx,bx           ;so we have the constant 0 in a reg
        mov     cx,[bp+DrawLength] ;CX=# of pixels to draw
        jcxz    VDrawDone       ;skip this if no pixels to draw
        cmp     [bp+HorizontalMoveDirection],0 ;draw right or left
        mov     bp,[bp+RowOffset] ;BP=offset to next row
        jz      VGoLeft         ;draw from right to left
VDrawRightLoop:                 ;draw from left to right
if ISVGA                        ;--VGA--
        and     es:[di],ah      ;AH becomes bit mask in write mode 3,
                                ; set/reset provides color
        lodsb                   ;get next draw control byte
        and     al,al           ;move right?
        jz      VAdvanceOneLineRight ;no move right
        ror     ah,1            ;move right
else                            ;--EGA--
        out     dx,al           ;set the desired bit mask
        and     es:[di],al      ;data doesn't matter (set/reset provides
                                ; color); just force read then write
        cmp     [si],bl         ;check draw control byte; move right?
        jz      VAdvanceOneLineRight ;no move right
        ror     al,1            ;move right
endif                           ;--------
        adc     di,bx           ;move one byte to the right if mask wrapped
VAdvanceOneLineRight:
ife ISVGA                       ;--EGA--
        inc     si              ;advance draw control list pointer
endif                           ;--------
        add     di,bp           ;move to the next scan line up or down
        loop    VDrawRightLoop  ;do next pixel, if any
        jmp     short VDrawDone ;done
VGoLeft:                        ;draw from right to left
VDrawLeftLoop:
if ISVGA                        ;--VGA--
        and     es:[di],ah      ;AH becomes bit mask in write mode 3
        lodsb                   ;get next draw control byte
        and     al,al           ;move left?
        jz      VAdvanceOneLineLeft ;no move left
        rol     ah,1            ;move left
else                            ;--EGA--
        out     dx,al           ;set the desired bit mask
        and     es:[di],al      ;data doesn't matter; force read/write
        cmp     [si],bl         ;check draw control byte; move left?
        jz      VAdvanceOneLineLeft ;no move left
        rol     al,1            ;move left
endif                           ;--------
        sbb     di,bx           ;move one byte to the left if mask wrapped
VAdvanceOneLineLeft:
ife ISVGA                       ;--EGA--
        inc     si              ;advance draw control list pointer
endif                           ;--------
```

```
        add     di,bp           ;move to the next scan line up or down
        loop    VDrawLeftLoop   ;do next pixel, if any
VDrawDone:
        pop     di              ;restore C register variables
        pop     si
        pop     bp
        ret
_DrawVOctant    endp
;*********************************************************************
; Draws the arc for an octant in which X is the major axis. (X,Y) is the
; starting point of the arc. HorizontalMoveDirection selects whether the
; arc advances to the left or right horizontally (0=left, 1=right).
; RowOffset contains the offset in bytes from one scan line to the next,
; controlling whether the arc is drawn up or down. DrawLength is the
; horizontal length in pixels of the arc, and DrawList is a list
; containing 0 for each point if the next point is horizontally aligned,
; and 1 if the next point is 1 pixel above or below diagonally.
;
; Graphics Controller Index register must already point to the Bit Mask
; register.
;
; C near-callable as:
;  void DrawHOctant(int X, int Y, int DrawLength, int RowOffset,
;       int HorizontalMoveDirection, unsigned char *DrawList)
;
; Uses same parameter structure as DrawVOctant().
;
        public _DrawHOctant
_DrawHOctant    proc    near
        push    bp              ;preserve caller's stack frame
        mov     bp,sp           ;point to our stack frame
        push    si              ;preserve C register variables
        push    di
;Point ES:DI to the byte the initial pixel is in.
        mov     ax,SCREEN_SEGMENT
        mov     es,ax
        mov     ax,SCREEN_WIDTH_IN_BYTES
        mul     [bp+Y]  ;Y*SCREEN_WIDTH_IN_BYTES
        mov     di,[bp+X] ;X
        mov     cx,di   ;set X aside in CX
        shr     di,1
        shr     di,1
        shr     di,1    ;X/8
        add     di,ax   ;screen offset = Y*SCREEN_WIDTH_IN_BYTES+X/8
        and     cl,07h  ;X modulo 8
        mov     bh,80h
        shr     bh,cl   ;initial bit mask = 80h shr (X modulo 8);
if ISVGA                ;--VGA--
        cld             ;for LODSB, used below
else                    ;--EGA--
        mov     dx,GC_INDEX+1 ;point DX to GC Data reg/bit mask
endif                    ;--------
        mov     si,[bp+DrawList] ;SI points to list to draw from
        sub     bl,bl           ;so we have the constant 0 in a reg
        mov     cx,[bp+DrawLength] ;CX=# of pixels to draw
        jcxz    HDrawDone       ;skip this if no pixels to draw
if ISVGA                        ;--VGA--
        sub     ah,ah           ;clear bit mask accumulator
else                            ;--EGA--
        sub     al,al           ;clear bit mask accumulator
endif                           ;--------
```

```
        cmp     [bp+HorizontalMoveDirection],0 ;draw right or left
        mov     bp,[bp+RowOffset] ;BP=offset to next row
        jz      HGoLeft         ;draw from right to left
HDrawRightLoop:                 ;draw from left to right
if ISVGA                        ;--VGA--
        or      ah,bh           ;put this pixel in bit mask accumulator
        lodsb                   ;get next draw control byte
        and     al,al           ;move up/down?
else                            ;--EGA--
        or      al,bh           ;put this pixel in bit mask accumulator
        cmp     [si],bl         ;check draw control byte; move up/down?
endif                           ;--------
        jz      HAdvanceOneLineRight ;no move up/down
                                ;move up/down; first draw accumulated pixels
if ISVGA                        ;--VGA--
        and     es:[di],ah      ;AH becomes bit mask in write mode 3
        sub     ah,ah           ;clear bit mask accumulator
else                            ;--EGA--
        out     dx,al           ;set the desired bit mask
        and     es:[di],al      ;data doesn't matter; force read/write
        sub     al,al           ;clear bit mask accumulator
endif                           ;--------
        add     di,bp           ;move to the next scan line up or down
HAdvanceOneLineRight:
ife ISVGA                       ;--EGA--
        inc     si              ;advance draw control list pointer
endif                           ;--------
        ror     bh,1            ;move to right; shift mask
        jnc     HDrawLoopRightBottom ;didn't wrap to the next byte
                                ;move to next byte; 1st draw accumulated pixels
if ISVGA                        ;--VGA--
        and     es:[di],ah      ;AH becomes bit mask in write mode 3
        sub     ah,ah           ;clear bit mask accumulator
else
        out     dx,al           ;set the desired bit mask
        and     es:[di],al      ;data doesn't matter; force read/write
        sub     al,al           ;clear bit mask accumulator
endif                           ;--------
        inc     di              ;move 1 byte to the right
HDrawLoopRightBottom:
        loop    HDrawRightLoop  ;draw next pixel, if any
        jmp     short HDrawDone ;done
HGoLeft:                        ;draw from right to left
HDrawLeftLoop:
if ISVGA                        ;--VGA--
        or      ah,bh           ;put this pixel in bit mask accumulator
        lodsb                   ;get next draw control byte
        and     al,al           ;move up/down?
else                            ;--EGA--
        or      al,bh           ;put this pixel in bit mask accumulator
        cmp     [si],bl         ;check draw control byte; move up/down?
endif                           ;--------
        jz      HAdvanceOneLineLeft ;no move up/down
                                ;move up/down; first draw accumulated pixels
if ISVGA                        ;--VGA--
        and     es:[di],ah      ;AH becomes bit mask in write mode 3
        sub     ah,ah           ;clear bit mask accumulator
else                            ;--EGA--
        out     dx,al           ;set the desired bit mask
        and     es:[di],al      ;data doesn't matter; force read/write
```

```
        sub     al,al           ;clear bit mask accumulator
endif                           ;--------
        add     di,bp           ;move to the next scan line up or down
HAdvanceOneLineLeft:
ife ISVGA                       ;--EGA--
        inc     si              ;advance draw control list pointer
endif                           ;--------
        rol     bh,1            ;move to left; shift mask
        jnc     HDrawLoopLeftBottom ;didn't wrap to next byte
                                ;move to next byte; 1st draw accumulated pixels
if ISVGA                        ;--VGA--
        and     es:[di],ah      ;AH becomes bit mask in write mode 3
        sub     ah,ah           ;clear bit mask accumulator
else                            ;--EGA--
        out     dx,al           ;set the desired bit mask
        and     es:[di],al      ;data doesn't matter; force read/write
        sub     al,al           ;clear bit mask accumulator
endif                           ;--------
        dec     di              ;move 1 byte to the left
HDrawLoopLeftBottom:
        loop    HDrawLeftLoop   ;draw next pixel, if any
HDrawDone:
                                ;draw any remaining accumulated pixels
if ISVGA                        ;--VGA--
        and     es:[di],ah      ;AH becomes bit mask in write mode 3
else                            ;--EGA--
        out     dx,al           ;set the desired bit mask
        and     es:[di],al      ;data doesn't matter; force read/write
endif                           ;--------
        pop     di              ;restore C register variables
        pop     si
        pop     bp
        ret
_DrawHOctant    endp
        end
```

Notes on the Ellipse-Drawing Implementations

This chapter is truly a synthesis of what has come before. We spent the previous chapter developing the incremental ellipse-drawing approach, and the chapter before that developing the draw-list approach for circles. In fact, not only is there nothing particularly new, but there's a bit of optimization overkill in Listing E.4.

I must in all honesty point out that it's scarcely worth bothering with converting the draw list generation code to assembly language at all. First, drawing from the draw list is likely to take much longer than generating the draw list, because four octants are drawn for each draw list generated and because drawing is usually slowed considerably by video wait states. That means that draw list generation doesn't represent a very large fraction of total execution time, and therefore isn't a particularly fruitful place to expend optimization effort.

Second, draw list generation involves many 32-bit variables, too many to be able to keep them all in the registers; when that's the case, particularly in fairly straightforward add-subtract-compare code like that used in draw list generation, I've found that

there's not likely to be much difference between good assembly code and the code produced by a good C compiler. Sure, the assembly-language code is better, but again the difference isn't likely to translate into a sizable improvement in overall performance.

When you optimize code, it's important to understand where in your code the effort will pay off best. In Listings E.3 and E.4, optimization effort is better spent in trying to draw octants from draw lists faster than in trying to generate the draw lists themselves faster.

Why Optimizing Isn't a Science

There is one slightly tricky element to Listing E.4. If you look closely, you'll note that each draw list element is *always* set to 0. Later, that same element may be set again, this time to 1, meaning that a single element may be set twice, incurring an extra instruction and an extra memory access. Is this wise?

In this case, it probably is, although the truth of the matter is far from clear; the question illustrates the hazards of optimizing in today's multi-platform (286, 386, 486, and Pentium, with a variety of memory architectures) world. Presetting each element to 0 allows us to branch and be done with it if there is no minor move, although it also requires us to perform a second set for each element for which there *is* a minor move. The alternative would be something like this:

```
     js    NoMoveMinor
     <advance minor coordinate>
     mov   byte ptr [di],1
     jmp   short MoveMajor
NoMoveMinor:
     mov   byte ptr [di],0
MoveMajor:
     inc   di
```

This code only sets each draw list element once, but also requires a branch in the case where the minor coordinate advances. That means that the latter approach saves one **MOV BYTE PTR [DI],0** when the minor coordinate advances, but adds one **JMP SHORT MoveMajor** at the same time—not a good trade on *any* 80x86 processor. Although in actual use, instruction fetching can alter those cycle counts somewhat; any variation is usually to the relative detriment of **JMP**, which empties the prefetch queue.

Well, then, why not load 1 or 0 into a register and then store the register, eliminating a memory access? That code would look like this:

```
     sub   al,al       ;assume there's no minor move
     js    NoMoveMinor
     <advance minor coordinate>
     inc   ax          ;upper byte doesn't matter, but word INC
                       ; is more efficient than byte INC
NoMoveMinor:
     stosb
```

This last approach might be faster—or it might not. It all depends on the processor and the memory architecture. On an 8088, there's no question that the last approach is faster; the 8088 instruction set favors both string instructions like **STOSB** and keeping values in registers. On a 386, however, **MOV [DI],0** takes just 2 cycles, and **INC DI** takes another 2; **STOSB** takes 4, so there's no advantage to **STOSB** there. (On a 486 and Pentium, **STOSB** is almost always a loser to **MOV**/**INC**.) However, the latter approach requires not only **STOSB** but also **SUB AL,AL** and possibly **INC AX**, so it's actually 2 cycles slower on a 386 in both cases.

At this point we get into issues such as whether the system has a cache, and if so whether the code is in the cache and whether it's a write-back or write-through cache, and if not whether we've hit a ready-to-go interleaved memory bank or the current column in static-column RAM. In short, there's no clear answer as to which code is fastest here unless you know not only your target processor but also your target memory architecture. The only thing that's constant across all 80x86-family processors is that branching is slow, so all of the alternative approaches we've discussed are faster than the obvious approach that we first discussed, the approach of not preloading or presetting at all and branching in both cases.

I'd like to be able to pull a clear, simple lesson out of all this, but the life of an optimizing PC programmer is neither simple nor clear. If there's a moral here, it would be: aim down the middle. That is, try to write code that provides good performance (by assembly language standards) on every common 80x86 processor, and terrible performance on none. In this particular example, that translates into not branching any more than you absolutely must.