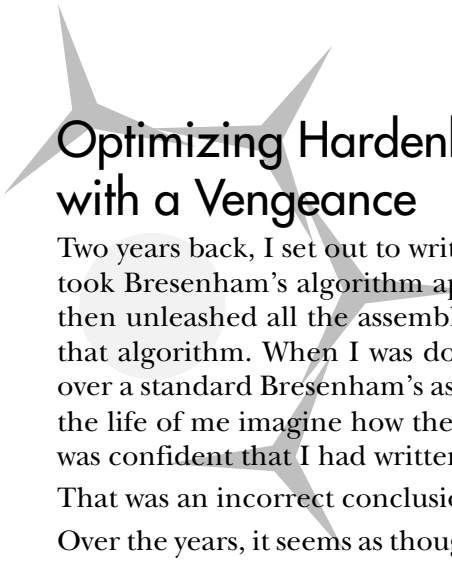
Chapter C

Circling in for the Kill

Chapter

C



Optimizing Hardenburgh's Circle Algorithm with a Vengeance

Two years back, I set out to write the fastest possible line-drawing code for the PC. I took Bresenham's algorithm apart piece by piece until I understood it completely, then unleashed all the assembly-language skills I had on the task of implementing that algorithm. When I was done, I had boosted performance by about two times over a standard Bresenham's assembly-language implementation, and I couldn't for the life of me imagine how the code could be improved one iota. In other words, I was confident that I had written just about the fastest possible code.

That was an incorrect conclusion—to put it mildly.

Over the years, it seems as though every graphics programmer I know has decided to clue me in on his or her favorite way to draw lines. I've heard about state-machine line drawing, run-based line drawing, fixed-point line drawing, and even a technique that lets the floating-point processor perform a single division while the line-drawing routine sets up, then uses the bits of the floating-point remainder to control line drawing. Wildly different as these approaches are, they have one thing in common—they're all faster than my original "optimized" code.

Which brings us to this chapter. In the previous chapter, we learned how to draw circles, then we vastly improved performance by switching from floating-point code to Hal Hardenbergh's integer-only algorithm that requires multiplication and division

only during the initial setup. That was step 1 of graphics optimization: selection of algorithm. Steps 2 (matching the algorithm to the hardware) and 3 (conversion of critical code to assembly language) are coming up next. When we're done, we'll have circle-drawing code that's more than 20 times as fast as the original implementation. Our final circle-drawing code will be more than 3 times as fast as the faster routine we developed last time*, even though both routines use exactly the same algorithm.* Clearly, there is more to performance than selecting the right algorithm, although that is an essential starting point.

Fast as our final implementation will be, however, I wouldn't dream of calling it optimized. As I said in the opening to this chapter, it's presumptuous indeed to think you've come up with the best possible graphics code, and circle drawing is no exception. In fact, the code could easily be speeded up by using unrolled loops to draw multiple pixels without looping, and could possibly be made faster yet with fixed-point arithmetic or by encoding the pixel list in a more readily usable format. There are also many small improvements—simplified calculations, eliminated redundancies, and the like—to be made throughout the code, although optimizing outside the inner loops tends to be a waste of time and effort.

My point is this: Don't take the code in this chapter as gospel. It's fast, but it could be at least a little faster—and maybe a *lot* faster. Understand what I've done, then try to come up with a way to do it better; one of the great things about PC graphics programming is that it's almost always possible to do exactly that.

Slimming Down the Main Loop

In order to be able to focus on specific optimizations in this chapter, I'm going to assume that you're familiar with circle drawing in general. If not, I suggest you read the previous chapter, which discussed the fundamentals of circle drawing and described in detail Hal Hardenburgh's integer-only algorithm that we'll use here.

The question at hand is this: How can we speed up the circle-drawing code we saw at the end of the previous chapter, which used integer-only calculations and required no multiplication or division inside the main loop? If you refer back to that code, you'll see that while the main loop is quite compact, it calls a separate routine to draw each pixel, and it's there that we can save a whole slew of processor cycles.

The pixel-drawing routine in the previous chapter was short, but it performed two time-consuming and avoidable tasks. First, it calculated each pixel's display memory offset from scratch, requiring both a multiply and the construction of a far pointer. Similarly, it generated each pixel's bit mask from scratch with a little arithmetic and a shift. As it turns out, neither of those actions is actually necessary, because with the circle-drawing approach we were using (and will continue to use) the offset and mask for each pixel can be calculated incrementally from the offset and mask for the *previous* pixel.

In other words, once we've drawn pixel *n*, we can generate the bit mask for pixel *n*+1 with at most a single-bit rotation. Likewise, we can calculate the offset for pixel *n*+1 with at most an increment or decrement followed by an addition. Those simpler calculations can save many cycles relative to calculating each pixel from scratch. We can also save some cycles by moving the code that draws each pixel into the drawing loop and eliminating the call to and return from the drawing function.

Reflecting Octants

There's another, less obvious optimization we can make. Consider this: The process of calculating adjacent points along one-eighth of a circle is nothing more than a matter of deciding whether to move along both axes after each pixel or only along the major axis; making these moves and drawing the corresponding pixels illuminates precisely those pixels closest to the desired arc. Given that, it should be clear that it's possible to calculate the set of moves from one pixel to the next and store that set in an array, then play back the array with varying major and minor axis directions eight times in order to draw the circle. Put another way, we could calculate and store the information needed to determine when to advance along the minor axis just once for a given circle, then play back that information eight times to draw the eight symmetric arcs that make up the circle, interpreting the information slightly differently each time.

What does that buy us? It lets us separate the arc calculation code from the arc drawing code so that each can be better optimized. Truth to tell, that's not such a big deal in C; we could just calculate each point and draw all eight symmetries on the spot, as we did in the last chapter, although that would require maintaining eight display memory pointers and eight masks. When we get to assembly language, however, separation of calculation and drawing will stand us in good stead; by reducing complexity it will allow us to keep all variables in registers for the duration of both the calculation and drawing loops—and that will translate directly into better performance.

I don't know whether the separation of calculation from drawing has a similarly large advantage in C, or indeed any advantage at all. However, given that the major purpose of the C code in this chapter is to prototype and illuminate the assembler code, I'll use the same approach of separating calculation and drawing in the C code that I'll use later in the assembly code.

Faster Circles in C

Listing C.1 shows a C circle-drawing function that uses the separate calculation and drawing approach, along with the incremental offset and mask calculation technique described earlier. Arcs with horizontal major axes are handled separately from arcs with vertical major axes because the process of advancing across a scan line of EGA/VGA memory is fundamentally different from that of advancing from one scan line

Listing	Compiler/Processor Microsoft C 5.0 286	 386	 Turbo C 2.0 286	 386
B.1 (C/floating point)	468 sec	175 sec	339 sec	127 sec
B.3 (C/integer)	44 sec	16 sec	47 sec	18 sec
C.1 (C/incremental pixel addressing)	31 sec	12 sec	32 sec	13 sec
C.3/4 (ISVGA=0) (ASM)	14 sec	7 sec	14 sec	7 sec
C.3/4 (ISVGA=1) (ASM/write mode 3)	13 sec	5.5 sec	13 sec	5.5 sec

Notes: The execution times shown are for the various circle-drawing implementations in this chapter and the last when linked to Listing C.2. Maximum optimization (/Ox for Microsoft C, -G -O -Z -r for Turbo C) was used to compile all C code. Times in the columns labelled "286" were recorded on a Video Seven VRAM VGA running on a 10-MHz 1-wait-state AT clone (a monochrome adapter was also installed, making the VGA an 8-bit device); times in the columns labelled "386" were recorded on a built-in Paradise VGA in a 20-MHz, 32K-0-wait-state-cache Toshiba 5200 (a monochrome adapter was also installed). No floating-point processor was installed in either computer. Results could vary considerably on different hardware.

Table C.1 Hardenburg Circles versus Optimized Hardenburgh Circles

to the next. When compiled and linked to Listing C.2, which repeats the drawing of a set of circles 20 times for timing purposes, Listing C.1 is about 40 percent faster than Listing B.3 in the previous chapter, which was our speed champ until now, as shown in Table C.1. That's nowhere near the improvement that Listing B.3 produced over Listing B.1, but it's certainly significant.

LISTING C.1 LC-1.C

```
/*
 * Draws a circle of the specified radius and color, using a fast
 * integer-only & square-root-free approach, and generating the
 * arc for one octant into a buffer, then drawing all 8 symmetries
 * from that buffer.
 * Compiles with either Borland or Microsoft.
 * Will work on VGA or EGA, but will draw what appears to be an
 * ellipse in non-square-pixel modes.
 */

#include <dos.h>
```

```
/* Handle differences between Borland and Micrsoft. 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 a circle of radius Radius in color Color centered at
   screen coordinate (X,Y) */

void DrawCircle(int X, int Y, int Radius, int Color) {
   int MajorAxis, MinorAxis;
   unsigned long RadiusSqMinusMajorAxisSq, MinorAxisSquaredThreshold;
   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 */

   /* Set up to draw the circle by setting the initial point to one
      end of the 1/8th of a circle arc we'll draw */
   MajorAxis = 0;
   MinorAxis = Radius;
   /* Set initial Radius**2 - MajorAxis**2 (MajorAxis is initially 0) */
   RadiusSqMinusMajorAxisSq = (unsigned long) Radius * Radius;
   /* Set threshold for minor axis movement at (MinorAxis - 0.5)**2 */
   MinorAxisSquaredThreshold = (unsigned long) MinorAxis * MinorAxis -
      MinorAxis;

   /* Calculate all points along an arc of 1/8th of the circle and
      store that info in PixList for later drawing */
   PixListPtr = PixList;
   do {
      /* Advance (Radius**2 - MajorAxis**2); if it equals or passes
         the MinorAxis**2 threshold, advance one pixel along both the
         major and minor axes and set the next MinorAxis**2 threshold;
         otherwise, advance one pixel only along the major axis. */
      RadiusSqMinusMajorAxisSq -=
         MajorAxis + MajorAxis + 1;
      if ( RadiusSqMinusMajorAxisSq <= MinorAxisSquaredThreshold ) {
         /* Advance 1 pixel along both the major and minor axes */
         MinorAxis--;
         MinorAxisSquaredThreshold -= MinorAxis + MinorAxis;
         *PixListPtr++ = 1;   /* advance along both axes */
      } else {
         *PixListPtr++ = 0;   /* advance only along the major axis */
      }
      MajorAxis++;   /* always advance one pixel along the major axis */
   } while ( MajorAxis <= MinorAxis );

   /* Now draw each of the 8 symmetries of the octant in turn */
   /* Draw the octants for which Y is the major axis */
   DrawVOctant(X-Radius,Y,MajorAxis,-SCREEN_WIDTH_IN_BYTES,1,PixList);
```

```
   DrawVOctant(X-Radius,Y,MajorAxis,SCREEN_WIDTH_IN_BYTES,1,PixList);
   DrawVOctant(X+Radius,Y,MajorAxis,-SCREEN_WIDTH_IN_BYTES,0,PixList);
   DrawVOctant(X+Radius,Y,MajorAxis,SCREEN_WIDTH_IN_BYTES,0,PixList);

   /* Draw the octants for which X is the major axis */
   DrawHOctant(X,Y-Radius,MajorAxis,SCREEN_WIDTH_IN_BYTES,0,PixList);
   DrawHOctant(X,Y-Radius,MajorAxis,SCREEN_WIDTH_IN_BYTES,1,PixList);
   DrawHOctant(X,Y+Radius,MajorAxis,-SCREEN_WIDTH_IN_BYTES,0,PixList);
   DrawHOctant(X,Y+Radius,MajorAxis,-SCREEN_WIDTH_IN_BYTES,1,PixList);

   /* 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);
}
```

It's worth noting that Listings C.1 and B.3 both use the same integer-only algorithm presented in the previous chapter. The performance difference between the two listings comes from understanding the code the compiler generates for each and the performance implications of that code. In particular, Listing C.1 virtually eliminates multiplication, multi-bit shifts, and call and return instructions, all of which are relatively slow on x86 processors.

LISTING C.2 LC-2.C

```
/*
 * Draws a series of concentric circles.
 * For VGA only, because mode 12h is unique to VGA.
 * Compile and link using Borland C++ with accompanying listings as follows:
 *    bcc -ms lC-1 lC-2            (or)
 *    bcc -ms lC-3 lC-2 lC-4.asm
 *
 *
 *
 */
#include <dos.h>

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

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

   /* Draw 20 sets of concentric circles for timing purposes. */
   for (i = 0; i < 20; i++) {
      for ( Radius = 10, Color = 7; Radius < 240; Radius += 2 ) {
         DrawCircle(640/2, 480/2, Radius, 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);
}
```

Notes on the C Implementation

Listing C.1, like Listing B.3, uses long integers because the squared values used can grow too large for short integers. The performance cost of long integers can be avoided by special-casing circles with radii less than 256. In order for circles with radii less than 256 to work with short integers, however, short *unsigned* integers must be used, or comparisons may not work correctly when squared quantities exceed 32,767.

Listing C.1 (and, indeed, all the listings in this chapter and the previous one) assumes that no clipping is needed. If that's not the case, the approach of generating a pixel list and then drawing the eight symmetries from the list would lend itself well to clipping, because the drawing routine could use the list to skip quickly over any initial portion of an octant that's outside a clip region, and could terminate processing immediately when the far edge of the clip region is reached. At the very least, clipping from a pixel list is surely more manageable than attempting to draw and clip all eight symmetries simultaneously, and is undoubtedly more efficient than calculating and drawing the clipped arcs for each of the eight octants separately.

Circles to the Metal: Assembly Language

I believe that the critical code for graphics primitives should be written in assembly language, and circle drawing is no exception. Listings C.3 and C.4 show a C/assembly language hybrid circle-drawing routine that is a good deal faster yet than Listing C.1; as Table C.1 shows, the code in Listings C.3 and C.4 is about twice as fast as Listing C.1.

LISTING C.3 LC-3.C

```
/*
 * Draws a circle of the specified radius and color, using a fast
 * integer-only & square-root-free approach, and generating the
 * arc for one octant into a buffer, then drawing all 8 symmetries
 * from that buffer. Uses assembly language for inner loops of octant
 * generation & drawing.
 * Compiles with either Borland or Microsoft.
 * Will work on VGA or EGA, but will draw what appears to be an
 * ellipse in non-square-pixel modes.
 */
#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 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 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 a circle of radius Radius in color Color centered at
 * screen coordinate (X,Y) */
void DrawCircle(int X, int Y, int Radius, int Color) {
   int MajorAxis, MinorAxis;
   unsigned long RadiusSqMinusMajorAxisSq, MinorAxisSquaredThreshold;
   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

   /* Set up to draw the circle by setting the initial point to one
      end of the 1/8th of a circle arc we'll draw */
   MajorAxis = 0;
   MinorAxis = Radius;
   /* Set initial Radius**2 - MajorAxis**2 (MajorAxis is initially 0) */
   RadiusSqMinusMajorAxisSq = (unsigned long) Radius * Radius;
   /* Set threshold for minor axis movement at (MinorAxis - 0.5)**2 */
   MinorAxisSquaredThreshold = (unsigned long) MinorAxis * MinorAxis -
      MinorAxis;

   /* Calculate all points along an arc of 1/8th of the circle.
      Results are placed in PixList */
   MajorAxis = GenerateOctant(PixList, MajorAxis, MinorAxis,
      RadiusSqMinusMajorAxisSq, MinorAxisSquaredThreshold);

   /* Now draw each of the 8 symmetries of the octant in turn */
   /* Draw the octants for which Y is the major axis */
   DrawVOctant(X-Radius,Y,MajorAxis,-SCREEN_WIDTH_IN_BYTES,1,PixList);
   DrawVOctant(X-Radius,Y,MajorAxis,SCREEN_WIDTH_IN_BYTES,1,PixList);
   DrawVOctant(X+Radius,Y,MajorAxis,-SCREEN_WIDTH_IN_BYTES,0,PixList);
   DrawVOctant(X+Radius,Y,MajorAxis,SCREEN_WIDTH_IN_BYTES,0,PixList);

   /* Draw the octants for which X is the major axis */
   DrawHOctant(X,Y-Radius,MajorAxis,SCREEN_WIDTH_IN_BYTES,0,PixList);
```

```
   DrawHOctant(X,Y-Radius,MajorAxis,SCREEN_WIDTH_IN_BYTES,1,PixList);
   DrawHOctant(X,Y+Radius,MajorAxis,-SCREEN_WIDTH_IN_BYTES,0, PixList);
   DrawHOctant(X,Y+Radius,MajorAxis,-SCREEN_WIDTH_IN_BYTES,1, PixList);

#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 C.4 LC-4.ASM

```
; Contains 3 C-callable routines: GenerateOctant, DrawVOctant, and
;       DrawHOctant. See individual routines for comments.
;
; 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 circle, 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 GenerateOctant(unsigned char *PixList, int MajorAxis,
;       int MinorAxis, unsigned long RadiusSqMinusMajorAxisSq,
;       unsigned long MinorAxisSquaredThreshold);
;
; Return value = MajorAxis
;
GenerateOctantParms       struc
          dw      ?       ;pushed BP
          dw      ?       ;return address
PixList   dw      ?       ;pointer to list to store draw control data in
MajorAxis dw      ?       ;initial major/minor axis coords relative to
MinorAxis dw      ?       ; to the center of the circle
RadiusSqMinusMajorAxisSq  dd ?   ;initial Radius**2 - MajorAxis**2
MinorAxisSquaredThreshold dd ?   ;initial threshhold for minor axis
GenerateOctantParms       ends   ; movement is MinorAxis**2 - MinorAxis
;
        public _GenerateOctant
_GenerateOctant 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
;                                       ;get all parms into registers
        mov     di,[PixList+bp]         ;point DI to PixList
        mov     ax,[MajorAxis+bp]       ;AX=MajorAxis
        mov     bx,[MinorAxis+bp]       ;BX=MinorAxis
        mov     cx,word ptr [RadiusSqMinusMajorAxisSq+bp]
```

```
        mov     dx,word ptr [RadiusSqMinusMajorAxisSq+bp+2]
                                ;DX:CX=RadiusSqMinusMajorAxisSq
        mov     si,word ptr [MinorAxisSquaredThreshold+bp]
        mov     bp,word ptr [MinorAxisSquaredThreshold+bp+2]
                                ;BP:SI=MinorAxisSquaredThreshold
GenLoop:
        sub     cx,1            ;subtract MajorAxis + MajorAxis + 1 from
        sbb     dx,0            ; RadiusSqMinusMajorAxisSq
        sub     cx,ax
        sbb     dx,0
        sub     cx,ax
        sbb     dx,0
        cmp     dx,bp           ;if RadiusSqMinusMajorAxisSq <=
        jb      IsMinorMove     ; MinorAxisSquaredThreshold, move along
        ja      NoMinorMove     ; minor as well as major, otherwise move
        cmp     cx,si           ; only along major
        ja      NoMinorMove
IsMinorMove:                    ;move along minor as well as major
        dec     bx              ;decrement MinorAxis
        sub     si,bx           ;subtract MinorAxis + MinorAxis from
        sbb     bp,0            ; MinorAxisSquaredThreshold
        sub     si,bx
        sbb     bp,0
        mov     byte ptr [di],1 ;enter 1 (move both axes) in PixList
        inc     di              ;advance PixList pointer
        inc     ax              ;increment MajorAxis
        cmp     ax,bx           ;done if MajorAxis > MinorAxis, else
        jbe     GenLoop         ; continue generating PixList entries
        jmp     short Done
NoMinorMove:
        mov     byte ptr [di],0 ;enter 0 (move only major) in PixList
        inc     di              ;advance PixList pointer
        inc     ax              ;increment MajorAxis
        cmp     ax,bx           ;done if MajorAxis > MinorAxis, else
        jbe     GenLoop         ; continue generating PixList entries
Done:
        pop     di              ;restore C register variables
        pop     si
        pop     bp
        ret
_GenerateOctant 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
```

Why didn't I use pure assembly language? Primarily because it's rarely worth using assembly outside of heavily-used loops. The overall performance improvement resulting from assembly language used anywhere else is generally imperceptible and hard to justify; assembly is difficult to write and harder to read and change. Listings C.3 and C.4 strike a good balance between performance, ease of coding and comprehension, and maintainability.

Listings C.3 and C.4 generally correspond directly to Listing C.1, although that may be obscured by the considerable optimization performed in Listing C.4. Two aspects of Listings C.3 and C.4 do not correspond to Listing C.1, however, and bear further discussion.

When the major axis is horizontal, multiple horizontally adjacent pixels are often drawn. Whenever multiple adjacent pixels are controlled by the same display memory byte, it is possible to draw all the pixels that reside in that byte with a single display memory read/write operation. This is highly desirable because display memory is extremely slow relative to normal system memory, especially on 386 and later computers. Consequently, Listing C.4 accumulates pixels that reside in the same byte when drawing horizontal-major-axis arcs, and accesses display memory only when all pixels that could possibly belong in a given byte have been processed. (See Chapter 35 for the application of this technique to straight-line drawing.) Pixel accumulation is a good example of matching an algorithm to the hardware; the basic operation of the algorithm is unchanged, but the code is fine-tuned so that the implementation suffers less from the poor performance of display memory.

Supporting VGA Write Mode 3

Listings C.3 and C.4 contain another example of matching the circle-drawing algorithm to the hardware: Optional support for write mode 3, which only the VGA supports. When the **ISVGA** symbols in both Listing C.3 and Listing C.4 are set to 1 (make sure both symbols are the same at all times), write mode 3 is used to draw pixels, improving overall performance by as much as 25 percent. **ISVGA** makes Listing C.4 in particular a little hard to follow—but I hope that you'll agree that the performance improvement and the exposure to a useful VGA-specific optimization are worth the trouble.

The virtue of write mode 3 is that it ANDs the Bit Mask register with the byte written by the CPU to form the working bit mask; that in turn means that there's no need to do an **OUT** when write mode 3 is used, thereby eliminating both an instruction and the many wait states that occur during I/O to most VGAs. What's more, by eliminating the need to perform **OUT**s we free up AL and DX for other purposes, in this case making it possible to use **LODSB**. (Note, though, that **LODSB** is not as fast as

```
MOV AL,[SI]
INC SI
```

on 486 and Pentium computers.) See Chapter 26 for more information about write mode 3.

In order for write mode 3 to work properly, the desired bit mask must be written to memory. (The set/reset circuitry provides the pixel color.) First, however, display memory must be read to latch the surrounding pixels, so that the bit mask can work its magic. An efficient way to both read from and write to display memory is to do so in a single instruction with **AND**, **OR**, or **XCHG**. (On 486s and Pentiums, using one **MOV** to read display memory followed by another **MOV** to write to display memory is a little faster than the single-instruction solutions, but it makes for more instruction bytes and destroys the contents of a register.) However, **XCHG** wipes out the register containing the working copy of the bit mask, and **AND** with any value other than 0FFH or **OR** with any value other than 0 normally alters the value written to memory so that it no longer sets the desired bit mask. (We just want to write the desired bit mask straight from the register to display memory.) This is solved by selecting read mode 1 (color compare mode) and setting the Color Don't Care register to 0, with the result that 0FFH is always read from display memory. Once that is done, we can **AND** the desired bit mask with display memory without altering either the value written to memory or the register containing the working copy of the bit mask. (See Chapter 28 for a discussion of this application of read mode 1.)

By the way, Listings C.3 and C.4 run just as well on VGAs as on EGAs if both **ISVGA** symbols are set to zero—they just run more slowly than if compiled/assembled specifically for the VGA.

There's no reason that the C code in Listing C.1 couldn't be made faster by using both the write mode 3/read mode 1 and pixel accumulation techniques we've applied in the assembler code. However, such approaches, which save an approximately fixed number of cycles, have a greater percentage impact on overall performance after all other aspects of the code have been streamlined, and so are most worth using in high-performance assembler code.

When graphics code is truly streamlined across the board, as in Listing C.4, the characteristics of the display adapter can affect performance significantly. For example, the 286 timings in Table 1 were all performed in a 10-MHz AT with both a Video Seven VRAM VGA and a monochrome adapter installed. It's a little-known fact that when a monochrome adapter is present, all VGAs must revert to being 8-bit memory devices. When the monochrome adapter was removed from the test-unit AT, allowing the VRAM VGA to become a 16-bit device, the time for the non-VGA specific version of Listings C.3 and C.4 dropped from 14 seconds to 12.5 seconds—an improvement of more than 10 percent. This means that Listing C.4 has reached the point where it is starting to bump up against the hardware's inherent limits—a sure sign of high-performance code, and an indication that perceptible performance variations from one make of VGA to the next are likely to occur.

Circles? Done

In the last chapter and Chapter 37, we've seen performance improvements of as much as 36 times from our initial circle-drawing implementation. Viewing this in terms of the three optimization steps I described at the outset, that very roughly breaks down as 5 to 10 times improvement from algorithm selection; 1.5 to 2.5 times improvement from matching the algorithm to the hardware; and 1.5 to 2 times improvement from conversion to assembly.

The conclusions are two: 1) algorithm selection is indeed important, but processor- and hardware-specific optimizations are important too, and 2) circles, not normally considered to be one of the speedier graphics primitives, can be drawn surprisingly rapidly, in the ballpark with if not quite so fast as lines.

Circles? Done. Now it's on to ellipses.