Chapter D

Circles That Squish

Chapter

D

An Efficient Algorithm for Drawing Ellipses

First things first. The best circle-drawing routine I know of gets faster seemingly by the day, so much so that I'm beginning to think that the ultimate circle-drawing routine will consist of nothing more than a single **NOP** once we figure out how to trim away all the superfluous instructions. Consider this:

Hal Hardenbergh's circle-drawing approach (presented in this space over the last two chapters) drew circles faster than I would have thought was possible. Even so, in the previous chapter I went to great pains to point out that I did *not* think that my implementation of Hal's approach was the fastest possible way to draw circles, but rather just one good way among many. Good thing, too, because not long after I wrote those words, I chanced to talk once again with Hal. I mentioned that there was really no need for even the few multiplications used in his algorithm; the squared terms cancelled out right at the beginning, allowing circles to be drawn without a single multiply or divide. Hal went home, thought about that for a while, and realized that, given the elimination of the squared terms, we didn't need 32-bit integers any more; plain old 16-bit integers would do just fine. Whereupon he devised yet another circle-drawing algorithm, this one using 16-bit integers and requiring less than 10 instructions per point to generate a circle arc.

To put that in more readily understood terms, I suspect that Hal's new algorithm can be used to draw circles faster than the Bresenham's line-drawing algorithm in Chapter 35 draws lines!

I'm not going to present Hal's new circle-drawing algorithm here, for a couple of reasons: I've probably overdosed you on circles by this point, and Hal may want to publish his new findings himself. If you know what you're doing, it should be easy enough to apply the above information to my discussions in the last two chapters to derive the new approach.

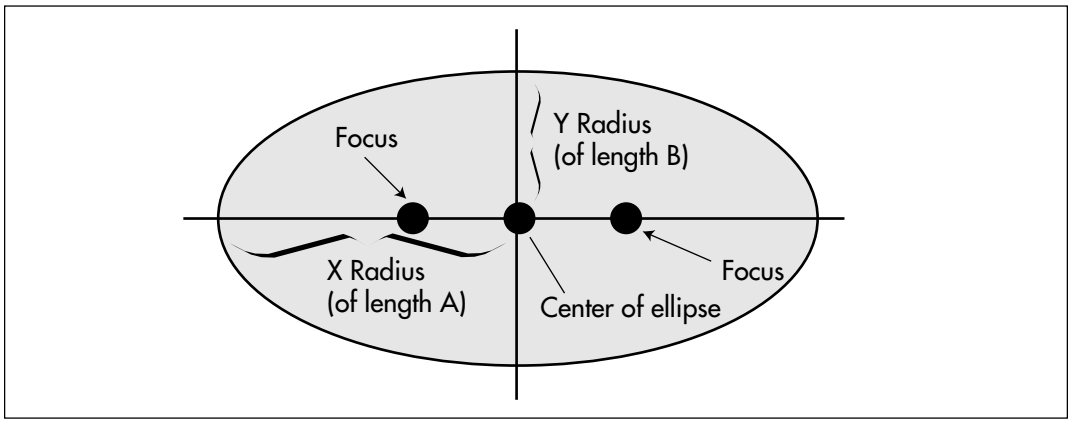
At any rate, the moral is clear. If you think your code is optimized to the limit, *maybe* it is—but don't bet your house on it.

And with that in mind, let's get on with learning how to draw ellipses pretty doggone fast.

A Quick Primer on Ellipses

An ellipse is an oval, which is to say a squished (or, possibly, non-squished) circle. Ellipses are centered around two foci; the sum of the distances from the two foci to any point on a given ellipse is a constant. A circle is actually a special case of ellipse for which every point is equidistant from the center, where both foci reside atop one another.

A circle has a single radius that is the distance from the center to every point on the circle. An ellipse has two basic radii, one that is the distance from the center of the ellipse to the edge along the X axis, and one that is the distance along the Y axis, as shown in Figure D.1. These two distances, which we'll call the X radius (of length A) and the Y radius (of length B), along with the center of the ellipse (the point at which the X and Y radii meet) are the fundamental parameters with which we'll work; we won't concern ourselves with the foci or distances from the foci from now on. You might think of ellipses as being defined by the smallest rectangle that contains them; however, for our purposes there's an additional limitation that A and B must be integers, so the encompassing rectangle must have even dimensions to allow the



The geometry of an ellipse.
Figure D.1

center to fall squarely on a pixel. (It's quite possible to support ellipses with fractionally-located centers, but it's generally not necessary and complicates matters, so we'll avoid it.)

Circles are certainly faster and easier to calculate than ellipses; whereas the equation for a circle is

$$X^2 + Y^2 = R^2$$

the equation for an ellipse is:

$$X^2/A^2 + Y^2/B^2 = 1$$

(Make A and B the same and you get a circle.) By the way, the equation above is for non-tilted ellipses—that is, ellipses with horizontal and vertical axes, where the foci share either a common X coordinate or a common Y coordinate—and that's all we'll work with in this book. Tilted ellipses are both flexible and useful, but they're also slower, more complicated, and generally quite a different kettle of fish from non-tilted ellipses.

Why Ellipses Matter

The question, as always, is how to get a PC to draw the object implied by the above equation for a non-tilted ellipse as quickly as possible, and that's what we'll spend this chapter and the next figuring out. It's well worth knowing how to draw ellipses quickly, for they're extremely useful. Of course, it's often necessary to draw ovals, and ellipses are handy for that alone, but there's more to it than that. You see, true circles, of the sort we learned to draw in the previous two chapters, are useless on displays with non-square pixels (that is, displays with aspect ratios other than 1:1). On such displays, true circles, which space pixels evenly in both directions, appear as ellipses. Examples of such displays are Hercules graphics, all CGA and EGA graphics modes, and all standard VGA graphics modes except 640×480. In other words, you can't use a true circle-drawing algorithm to draw circles in any standard IBM mode except one.

You *can* use ellipses to draw circles in all those modes, though; just adjust the ratio of the X and Y radii of each ellipse to balance the aspect ratio of the display, and bingo; you have a circle. For example, the aspect ratio of the mode 10H display is about 4:3 (4 pixels in the X direction covers the same physical distance on the screen as 3 pixels in the Y direction), so if you draw an ellipse with an eccentricity (A/B ratio) of 4/3, it will appear as a circle in mode 10H. We'll see an example of this shortly. The important point is that the capability to draw ellipses is not only useful for drawing ovals but essential for drawing circles in many PC graphics modes.

Learning to Draw Ellipses Fast: Divide and Conquer

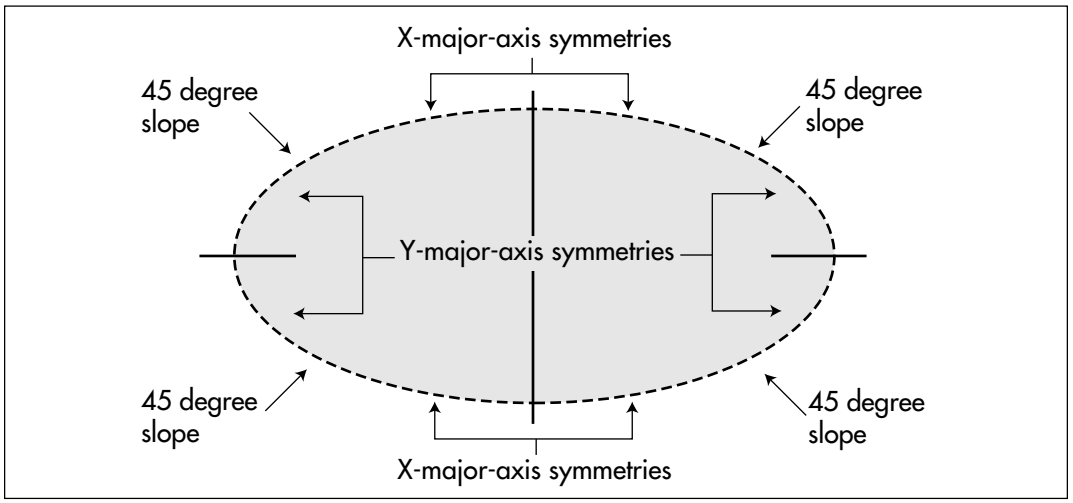
Now that we've established that ellipses are good stuff, how will we draw them fast? The path we'll take will be familiar to those of you who have followed the past two chapters on circles. First, we'll learn how to draw ellipses using the basic ellipse equation and floating-point arithmetic. Next, we'll derive a far more efficient algorithm, one which uses no non-integer arithmetic or square roots and requires no multiplies or divides in the main loops, and implement it in C. That will take us to the end of this chapter. In the next chapter, we'll tune the C implementation to the quirks of the EGA and VGA, and finally we'll convert the critical code to assembly language.

So, let's start learning how to draw ellipses. We'll start by drawing them slowly, but you can be sure that *that* will change.

Drawing an Ellipse the Easy—and Slow—Way

The straightforward way to draw an ellipse is implemented in Listing D.1. This listing takes advantage of the four-way symmetry of non-tilted ellipses, shown in Figure D.2, by generating one arc in which X is the major axis (that is, the X coordinate advances faster) and drawing all four symmetries at once via four calls to a dot-plot function, then generating one arc in which Y is the major axis and drawing all four symmetries the same way.

Arc generation is accomplished by starting at the point where the major axis is 0 relative to the center of the ellipse and the minor axis is the maximum distance from the center, then stepping the major axis by 1 pixel each time and calculating the



The symmetry of non-tilted ellipses.
Figure D.2

corresponding minor axis point via floating-point arithmetic according to the ellipse equation I stated earlier. This continues until the arc reaches the point at which the major axis changes, which is the point at which the slope of the arc reaches 45 degrees.

So, for example, when drawing an arc for which X is the major axis, the initial point drawn would be (0,B). The next point drawn would be (1,*y*), where *y* is calculated as follows:

$$x^2/A^2 + y^2/B^2 = 1$$
$$y^2/B^2 = 1 - x^2/A^2$$
$$y^2 = B^2 - B^2*x^2/A^2$$

Therefore,

$$y = \mathrm{sqrt}(B^2 - B^2*x^2/A^2)$$

is rounded to the nearest integer. The same calculation is repeated for each and every *x* as *x* is incremented along the arc, with the calculated coordinates reflected around the ellipse. (All *x* and *y* coordinates discussed are relative to the center of the ellipse.) That's easy enough, eh? The only trick is knowing when to stop, and that happens when the *y* component of

$$x^2/A^2 + y^2/B^2 = 1$$

is no longer the larger component, as detected by:

$$y^2/B^2 <= x^2/A^2$$

As you'd expect, the same thing is done for the arc where *y* advances faster, but with *x* and *y*, A and B swapped in the calculations.

LISTING D.1 LD-1.C

```
/*
 * Draws an ellipse of the specified X and Y axis radii and color,
 * using floating-point calculations.
 * Compiles with either Borland or Microsoft.
 * VGA or EGA.
 */

#include <math.h>
#include <dos.h>

/* Borland accepts outp 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 modes 10h and 12h */
#define SCREEN_SEGMENT            0xA000      /* mode 10h/12h display memory seg */
#define GC_INDEX                  0x3CE       /* Graphics Controller Index 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 */
```

```
/* Draws a pixel at screen coordinate (X,Y) */
void DrawDot(int X, int Y) {
   unsigned char far *ScreenPtr;

   /* Point to the byte the 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 bit mask within the byte for the pixel */
   outp(GC_INDEX + 1, 0x80 >> (X & 0x07));

   /* 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;
}

/* 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;
   double ASquared = (double) A * A;
   double BSquared = (double) B * B;
   double Temp;

   /* 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 initial top & bottom points */
   DrawDot(X, Y+B);
   DrawDot(X, Y-B);

   /* Draw the four arcs */
   for (WorkingX = 0; ; ) {
      /* Advance one pixel along the X axis */
      WorkingX++;

      /* Calculate the corresponding point along the Y axis. Guard
         against floating-point roundoff making the intermediate term
         less than 0 */
      Temp = BSquared - (BSquared *
         (double)WorkingX * (double)WorkingX / ASquared);
      if ( Temp >= 0 ) {
         WorkingY = sqrt(Temp) + 0.5;
      } else {
         WorkingY = 0;
      }
```

```
      /* Stop if X is no longer the major axis (the arc has passed the
         45-degree point) */
      if (((double)WorkingY/BSquared) <= ((double)WorkingX/ASquared))
         break;

      /* Draw the 4 symmetries of the current point */
      DrawDot(X+WorkingX, Y-WorkingY);
      DrawDot(X-WorkingX, Y-WorkingY);
      DrawDot(X+WorkingX, Y+WorkingY);
      DrawDot(X-WorkingX, Y+WorkingY);
   }

   /* Draw the four symmetric arcs for which Y advances faster (that is,
      for which Y is the major axis) */
   /* Draw the initial left & right points */
   DrawDot(X+A, Y);
   DrawDot(X-A, Y);

   /* Draw the four arcs */
   for (WorkingY = 0; ; ) {
      /* Advance one pixel along the Y axis */
      WorkingY++;

      /* Calculate the corresponding point along the X axis. Guard
         against floating-point roundoff making the intermediate term
         less than 0 */
      Temp = ASquared - (ASquared *
         (double)WorkingY * (double)WorkingY / BSquared);
      if ( Temp >= 0 ) {
         WorkingX = sqrt(Temp) + 0.5;
      } else {
         WorkingX = 0;           /* floating-point roundoff */
      }

      /* Stop if Y is no longer the major axis (the arc has passed the
         45-degree point) */
      if (((double)WorkingX/ASquared) < ((double)WorkingY/BSquared))
         break;

      /* Draw the 4 symmetries of the current point */
      DrawDot(X+WorkingX, Y-WorkingY);
      DrawDot(X-WorkingX, Y-WorkingY);
      DrawDot(X+WorkingX, Y+WorkingY);
      DrawDot(X-WorkingX, Y+WorkingY);
   }

   /* 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 D.2 LD-2.C

```
/*
 * Draws a series of concentric ellipses that should appear to be
 * circles in the EGA's hi-res mode, mode 10h. (They may not appear
 * to be circles on monitors that don't display mode 10h with the
 * same aspect ratio as the Enhanced Color Display.)
 * For EGA or VGA.
```

```
 * Compile and link (using Borland C++) with lD-X.c (where X is 1 or 4) with:
 *    bcc -ms -eLD-2X.EXE  LD-2.C  LD-X.C
 *
 */

#include <dos.h>

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

   /* Select EGA's hi-res 640x350 graphics mode, mode 10h */
   Regs.x.ax = 0x0010;
   int86(0x10, &Regs, &Regs);

   /* Draw concentric ellipses */
   for ( BaseRadius = 2, Color = 7; BaseRadius < 58; BaseRadius++ ) {
      DrawEllipse(640/2, 350/2, BaseRadius*4, BaseRadius*3, 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);
}
```

LISTING D.3 LD-3.C

```
/*
 * Draws nested ellipses of varying eccentricities in the VGA's
 * hi-res mode, mode 12h.
 * For VGA only.
 * Compile and link (using Borland C++) with lD-X.c (where X is 1 or 4) with:
 *    bcc -ms -eLD-3X.EXE  LD-3.C  LD-X.C
 *
 */

#include <dos.h>

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

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

   /* Draw nested ellipses */
   for ( XRadius = 100, YRadius = 2, Color = 7; YRadius < 240;
         XRadius++, 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);
}
```

Link Listing D.1 to Listing D.2 to see ellipses drawn to appear as circles in mode 10H. Some multiscanning monitors don't provide a 4:3 aspect ratio in mode 10H, so the ellipses may not look all that circular. Trust me, they do have an eccentricity of 1.33. Link Listing D.1 to Listing D.3 to see ellipses drawn with a variety of eccentricities. When you run these listings, you will see that ellipses drawn by stepping the major axis tend to look rather jagged; that's the cost of performance, and the appearance of these ellipses is nonetheless perfectly acceptable. The alternative is antialiased drawing, which can produce stunning results in 256-color and high-color modes, but would be orders of magnitude slower than the step-based ellipse drawing we'll see shortly.

As you can see, drawing an ellipse is no great trick. However, drawing an ellipse with reasonable performance requires a little more thought.

Ellipse Drawing: An Incremental Approach

Remember the incremental, integer-only algorithm we used to speed up circle drawing two chapters back? I hope so, because fast ellipse drawing, as implemented in Listing D.4, is strikingly similar, if slightly more complicated, and I'm not going to discuss the process in as much detail this time. Basically, instead of calculating the minor axis coordinate from scratch for each pixel, the incremental approach calculates it as a delta from the last pixel.

LISTING D.4 LD-4.C

```
/*
 * Draws an ellipse of the specified X and Y axis radii and color,
 * using a fast integer-only & square-root-free approach.
 * Compiles with either Borland or Microsoft.
 * VGA or EGA.
 */

#include <math.h>
#include <dos.h>

/* Borland accepts outp 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 modes 10h and 12h */
#define SCREEN_SEGMENT            0xA000      /* mode 10h/12h display memory seg */
#define GC_INDEX                  0x3CE       /* Graphics Controller Index 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 */

/* Draws a pixel at screen coordinate (X,Y) */
void DrawDot(int X, int Y) {
   unsigned char far *ScreenPtr;
```

```
   /* Point to the byte the 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 bit mask within the byte for the pixel */
   outp(GC_INDEX + 1, 0x80 >> (X & 0x07));

   /* 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;
}

/* 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;

   /* 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 initial top & bottom points */
   DrawDot(X, Y+B);
   DrawDot(X, Y-B);

   /* Draw the four arcs; set draw parameters for initial point (0,B) */
   WorkingX = 0;
   WorkingY = B;
   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 threhold
         accordingly */
      if ( Threshold >= 0 ) {
         YAdjust -= ASquared * 2;
```

```
      Threshold -= YAdjust;
      WorkingY--;
   }

   /* 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;

   /* Draw the 4 symmetries of the current point */
   DrawDot(X+WorkingX, Y-WorkingY);
   DrawDot(X-WorkingX, Y-WorkingY);
   DrawDot(X+WorkingX, Y+WorkingY);
   DrawDot(X-WorkingX, Y+WorkingY);
}

/* Draw the four symmetric arcs for which Y advances faster (that is,
   for which Y is the major axis) */
/* Draw the initial left & right points */
DrawDot(X+A, Y);
DrawDot(X-A, Y);

/* Draw the four arcs; set draw parameters for initial point (A,0) */
WorkingX = A;
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;
      WorkingX--;
   }

   /* 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;

   /* Draw the 4 symmetries of the current point */
   DrawDot(X+WorkingX, Y-WorkingY);
   DrawDot(X-WorkingX, Y-WorkingY);
```

```
      DrawDot(X+WorkingX, Y+WorkingY);
      DrawDot(X-WorkingX, Y+WorkingY);
   }

   /* 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);
}
```

The tremendous advantage of the incremental approach is that all terms used can be maintained as integers rather than floating-point values. Better yet, no multiplication or division is required to advance from one point to the next (not counting multiplication by 2, which is really an add or shift). The incremental approach isn't quite as fast for ellipses as for circles, but it nonetheless makes ellipses *nearly* as fast as the circles we've drawn in the last two chapters, and that's remarkably fast.

That said, let's take a quick trip through the math of the incremental ellipse-drawing approach for those of you with a mind to do some tinkering on your own.

A Thumbnail Derivation of the Incremental Approach

As with the floating-point approach, the incremental approach draws ellipses by generating an arc for which X advances more rapidly and then drawing the four symmetries, then doing the same for Y. Also like the floating-point approach, the coordinate which advances more rapidly—the major axis for the arc being drawn—is incremented by 1 each time, and the corresponding minor axis point is drawn. The only difference between the two approaches lies in the way that the minor axis coordinate is determined. Where the floating-point approach recalculates the minor axis coordinate, the incremental approach merely decides whether the minor axis coordinate has changed from the previous point, and advances that coordinate by 1 if that is the case.

The trick to the incremental approach, then, lies in deciding when it's time to advance the minor axis coordinate. Here, in a nutshell, is how that works. The equation for an ellipse is

$$x^2/A^2 + y^2/B^2 = 1$$

which can be expressed as:

$$B^2*x^2 + A^2*y^2 - A^2*B^2 = 0$$

When drawing an arc for which x is the major axis, all we'll do is evaluate the above equation initially for $x = 0$ and $y = B - 0.5$, which will give us a measure of how far off x is from the value at which y does equal B - 0.5 (B - 0.5 being the point at which y gets closer to the next pixel and advances). Then, we'll reevaluate the equation for x+1 each time x advances one pixel. When the result becomes positive, we'll have passed the point at which y is closer to the next pixel, so we'll decrement y and adjust the

equation for the new *y* value, then start looking in the same way for the next time *y* advances. That's really all there is to it.

So, when drawing an arc for which *x* advances faster, we'll set the initial *y* to B - 0.5 and the initial *x* to 0, which, plugged into the equation, gives us

$$B^2*0^2 + A^2*(B-0.5)^2 - A^2*B^2 = 0$$

which is to say (squaring B - 0.5)

$$0.25*A^2 - A^2*B = 0$$

so the initial threshold is:

$$A^2/4 - A^2*B$$

This is easy enough to calculate with integer arithmetic. True, we're ignoring a possible fractional term from $A^2/4$, but that's no problem; we'll simply choose to advance *y* if the threshold becomes exactly 0, thereby correctly handling the case where there's an implied fractional value.

Now that we have an initial threshold, we have to adjust it each time we advance *x* until it becomes positive, indicating a change in *y*. That's done by advancing the *x*-based component of the threshold from

$$B^2*x^2$$

to

$$B^2*(x+1)^2$$

which can be expressed as

$$B^2*(x^2+2*x+1)$$

or:

$$B^2*x^2 + B^2*2*x + B^2$$

B^2*x^2 is already in the current threshold equation, so

$$B^2*2*x + B^2$$

is added to the *x*-based term each time *x* advances, and that quantity can be maintained with integer arithmetic on an ongoing basis.

When the threshold is reached or exceeded, the *y* coordinate is decremented by 1, and the threshold must be adjusted back down in preparation for the next advance. This is done by adjusting the *y* component of the ellipse equation to

$$B^2*(y-1)^2$$

which is

$B^2*(y^2-2*y+1)$

or:

$B^2*y^2 - B^2*2*y + B^2$

Since B^2*y^2 is the value we're adjusting from, the incremental portion of this equation is simply $-B^2*2*y + B^2$, and, like the *x* component above, that value is easily maintained with integer arithmetic as *y* advances. (Note, however, that the *y* in the above equation has a fractional component of 0.5 that ends up cancelling the B^2 added at the end of the equation. See Chapter B for a more detailed discussion of this phenomenon in the context of circles.)

Drawing stops when the 45 degree point is reached. That condition is detected when the minor axis adjustment equals or exceeds the major axis adjustment, thereby becoming the dominant component in calculating the threshold and causing the arc to advance more rapidly along the minor axis.

I've gone fast here, because we covered this approach in detail when we discussed circles, but everything you really need to know to understand how the incremental approach works for ellipses is laid out above. If you had trouble following along, you might refer back to the lengthier explanation of the incremental approach for circles in Chapter B.

Or you might not. After all, what you really need to know is how to draw ellipses fast, and Listing D.4 does that, with far better results yet to come in the next chapter.

Notes and Caveats on the Code

Unlike our circle-drawing code, the ellipse-drawing code in Listing D.4 won't handle radii as large as 32K; the limit varies depending on the radii combination, but is never less than 1K. Given that 800 is the largest usable non-clipped radius on the highest-resolution SuperVGA available, a limit of 1K shouldn't pose any problem. If a greater range is needed, integers larger than 32 bits could be used, although that's more easily done in assembly language than in C. Along the same lines, calculations for smaller ellipses could potentially be performed using smaller integers. In general, no particular attempt was made to optimize the code presented in this chapter. In the next chapter we'll worry about fine-tuning, which is pointless without the foundation of a good algorithm such as the one we've developed here.

The incremental approach used in Listing D.4 is not Hal Hardenbergh's ellipse-drawing approach. Hal has come up with a fixed-point technique that is slightly less precise due to fractional roundoff but looks to be faster than the approach I've presented. I derived the approach I've presented from the circle-drawing approach we've already covered, because I prefer exact plotting when I can get it at little cost and

because I thought it would be easier for readers to understand an extension of what we've already covered than something new.

Notwithstanding that this is not Hal's approach, he patiently let me bounce it off him and kept me from getting wildly off track, for which I am most grateful.

How Fast Is It?

The big question, of course, is: How much faster is the incremental approach? *Plenty.* Listing D.2 runs about 20 times faster when linked to the version of **DrawEllipse** in Listing D.4 than to the version in Listing D.1, and Listing D.3 produces similar results. A numeric coprocessor would help Listing D.1, but not *that* much; anyway, you can't count on every system having a coprocessor. And we've only begun to kick ellipse drawing into high gear. In the next chapter we'll tackle the two remaining legs of the optimization sequence by tailoring the code to the EGA/VGA and converting to assembler. Based on our experience with circle drawing, I'd expect a performance improvement of an additional two to three times, with the tally for our final code running at around 40 to 60 times faster than Listing D.1.