Chapter B

Circling Around the VGA

Chapter

B

Understanding Hardenburgh's Algorithm for Fast Circles

Whenever I pick up one of the classic books about graphics algorithms, I come away thinking that those guys must be awfully smart, because *I* sure as heck don't understand what they're talking about. The explanations are cryptic, difficult to follow, and often leave key elements as the legendary "exercise for the reader." Worse, they're difficult to relate to the real world; the authors tell you how to derive an algorithm, when what you really want to know is how it works and how to implement it efficiently—and that final and most important step is, sadly, often omitted, or, at best, shown in marginally useful pseudocode.

I don't know why this state of affairs exists, but exist it does. One theory is that academics think in abstractions, not implementations. Maybe so. Another, more cynical, school of thought holds that academics must publish "major" works in order to get degrees, tenure, grants, and the like, so they dress up relatively simple concepts with a great deal of theory to make them seem more important or profound than they might be. Again, maybe so. At any rate, you and I are left holding the bag, with graphics code waiting to be written on the one hand and cryptic, highly theoretical explanations on the other, and a yawning gap in between.

It is against this background that I received a letter from Hal Hardenberg (who many may remember from his wild and wildly entertaining "DTACK Grounded" newsletter

and "Offramp" column in *Programmer's Journal* some years ago) regarding an article I wrote on line drawing. Hal's letter began thusly:

"*Ten pages of typeset text on drawing a LINE???* Yikes! I had in mind 5 pages on drawing three kinds of curves!" (Hal is inordinately fond of italics and exclamation points.)

This gentle missive was followed by a call from Hal asking whether I'd be interested in writing an article with him about drawing circles and ellipses. He felt strongly that there was nothing particularly complicated about any of the above, and that one article should do it for both.

In a world of people who puff themselves up by making the simple complex, I was more than slightly intrigued to meet someone who promised to make the complex simple. Hal and I got together, and while he decided to leave the article writing to me (he merely instructed me in the concepts and gave me his 68000 circle and ellipse drawing code; the man is remarkably generous with his considerable knowledge), he did indeed make the whole topic seem simple. Not quite so simple that it'll fit in one chapter—I'm going to spend two chapters on circle drawing and two on ellipses—but remarkably simpler than the graphics books would make it seem to be, and the credit for the concepts in these installments goes enthusiastically to Hal.

Hal thinks his approach may be the one used by Bresenham's circle-drawing algorithm, but he's not sure; he got fed up with the turgid nature of the standard references and derived his own approach. That's our good fortune, for Hal's approach is both fast—almost in a class with line drawing—and easy to understand.

With the credits and history out of the way, let's get started with circle drawing. We'll spend this chapter and the next on circles and then progress to the more flexible but more complicated ellipses. In the process, we'll do a good deal of the sort of low-level C and assembly optimization that I've demonstrated many times before in my articles, columns, and books like *Zen of Code Optimization* (Coriolis Group Books, 1994); we'll also learn more than a little about the sort of optimization that springs from understanding the operation of an algorithm thoroughly and then reshaping it to match the PC's capabilities. Only by applying both sorts of optimization can we test the limits of the PC's performance.

Why and When Circles and Ellipses Matter

Circles and ellipses rank no better than fourth on the list of most-used graphics elements, after bitblts, lines, and area fills. Nonetheless, they are used for many sorts of drawing, particularly in CAD and CAE applications, and their importance is magnified because most software takes so darn *long* to draw them.

True circles—that is, objects for which every point, as measured in *pixels*, not screen distance, is the same distance from the center—are useful only on screens that offer square pixels (pixels spaced equidistant along both axes). On any other sort of screen, true circles come out squashed or distended along one axis or the other. Fortunately, all

graphics modes in common use today—640×480, 800×600, and 1024×768—offer square pixels on popular displays, or close enough to square pixels as makes no difference.

Just to be sure we're all speaking the same language, let me define some terms. *Ellipses* are ovals drawn around two foci, with a constant combined distance from the two foci to each point on the ellipse. For the purposes of this book, ellipses will further be defined to have symmetry around the X and Y axes (the foci share either a common Y coordinate or a common X coordinate). That is, their major axes are either vertical or horizontal. Tilted ellipses, which I'm not going to tackle here, need not have symmetry around any axis. Basically, ellipses are specific 90-degree alignments of tilted ellipses, and circles are ellipses where the foci are both at the same point. As a final definition, I'll discuss all drawing of circles and ellipses in terms of pixels, not screen distance, except where otherwise noted, because pixels are the units with which we'll always work directly.

For this chapter and the next, we'll put ellipses aside and focus on circles.

The Basics of Drawing a Circle

Drawing a circle is actually pretty easy. The basic equation for a circle is

$$Radius^2 = X^2 + Y^2$$

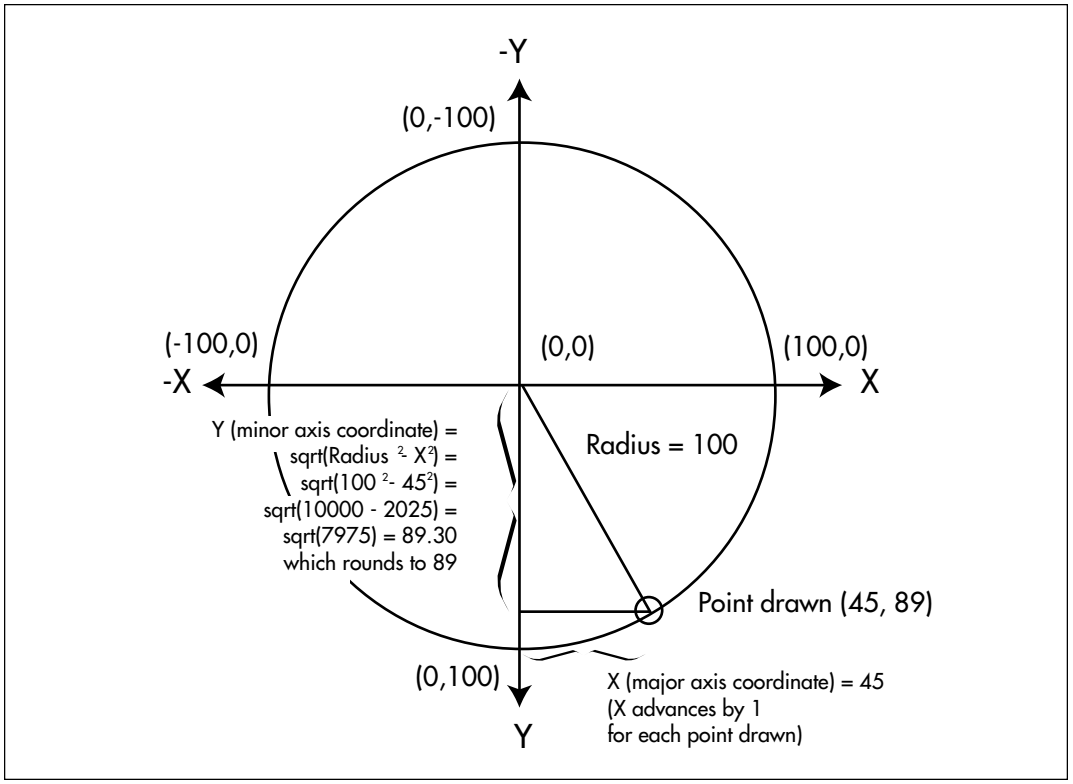
which is nothing more than a way of saying that every point on a circle is exactly **Radius** distance away from the center of the circle. (**X** and **Y** form two legs of a right triangle, with **Radius** as the hypotenuse. If you know your Pythagoras, you know how to draw circles.)

Viewing circle drawing in computer implementation terms, we can first cut our workload by realizing that we actually only need to generate 1/8th of a circle; symmetry does the rest. Next, we can generate that 1/8th-circle arc by starting at the zero setting for the major axis (the axis which advances more rapidly) and the maximum setting (**Radius**) for the minor axis, then advancing one pixel along the major axis at a time, and using the following formula to calculate the corresponding minor axis point, as shown in Figure B.1:

$$MinorAxis = sqrt(Radius^2 - MajorAxis^2)$$

Note that I'm using the terms "major axis" and "minor axis" rather than **X** and **Y** because the eight-way symmetry allows us to use the same arc calculations when either **X** or **Y** is the major axis. The arc is complete when the major axis coordinate equals or exceeds the minor axis coordinate for a point. The arc can be regenerated for each of the eight symmetries, but it's easier and less calculation-intensive to draw all eight symmetries for each point at once.

That sounds a little complicated, but is in fact less so than you might think. Listing B.1 shows a straightforward C implementation of the above approach for 16-color



Calculating the arc.
Figure B.1

modes such as modes 10H and 12H. As you can see, the code isn't very long or complicated at all, and what complication there is largely results from controlling the VGA's bit mask and set/reset features. Basically, Listing B.1 advances one pixel along the major axis, calculates the corresponding minor axis point, and then translates that result to all eight symmetries, drawing the eight points one after another through a dot-plot routine.

Listing B.2 is a demonstration program for Listing B.1. It calls the function in Listing B.1 to draw a screen full of multicolored concentric circles. If you run Listing B.2, you'll find that it does indeed draw circles, albeit none too quickly.

LISTING B.1 LB-1.C

```
/*
 * Draws a circle of the specified radius and color.
 * Compiles with either Borland or Microsoft compilers.
 * Will work on VGA or EGA in 16-color mode.
 */
```

```
#include <math.h>
#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 */

/* 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 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;
   double RadiusSquared = (double) Radius * Radius;

   /* 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 pointing to 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;
```

```
   /* Draw all points along an arc of 1/8th of the circle, drawing
      all 8 symmetries at the same time */
   do {
   /* Draw all 8 symmetries of current point */
      DrawDot(X+MajorAxis, Y-MinorAxis);
      DrawDot(X-MajorAxis, Y-MinorAxis);
      DrawDot(X+MajorAxis, Y+MinorAxis);
      DrawDot(X-MajorAxis, Y+MinorAxis);
      DrawDot(X+MinorAxis, Y-MajorAxis);
      DrawDot(X-MinorAxis, Y-MajorAxis);
      DrawDot(X+MinorAxis, Y+MajorAxis);
      DrawDot(X-MinorAxis, Y+MajorAxis);
      MajorAxis++;                          /* advance one pixel along major axis */
      MinorAxis =
         sqrt(RadiusSquared - ((double) MajorAxis * MajorAxis)) + 0.5;
      /* calculate corresponding point along minor axis */
  } while ( MajorAxis <= MinorAxis );

  /* 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 B.2 LB-2.C

```
/*
 * Draws a series of concentric circles.
 * For VGA only, because mode 12h is unique to VGA.
 * Compile and link with LB-X (1 or 3) with Borland C++:
 *   Bcc LB - X.C  LB - 2.C
 *
 */

#include <dos.h>

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

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

   /* Draw concentric circles */
   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);
}
```

Drawing a Circle More Efficiently

Now that we know how to draw a circle, let's think about how we might do it better. There are two paths to take, algorithmic refinement and implementation optimization (that is, converting to assembly, code massaging, fine-tuning, things like that). When given this choice, always take the algorithmic approach first, for it's easy to fine-tune after settling on an algorithm, but it's very difficult to change algorithms without wasting effort once you've fine-tuned. We'll concentrate on algorithmic refinement for the remainder of this chapter, continuing to work in C code with a dot-plot routine so that the operation of the algorithm is apparent. Then, in the next chapter, we'll tighten up the C code by eliminating the independent plotting of each point, and we'll finally arrive at a high-speed assembly implementation.

The optimization we'll make to Listing B.1 is this: Rather than calculating each minor axis point by taking an extremely time-consuming floating-point square root, we'll use purely integer calculations to determine when it's time to advance one pixel along the minor axis. (Granted, a floating-point coprocessor would speed up the square root calculation, but integer approaches are faster still, and even in these days where the 486 is the baseline Intel CPU, much of the installed base still doesn't have floating-point coprocessors.) In this respect, the faster circle-drawing implementation will be similar to the line-drawing code presented in Chapter 35; knowing as we do that the minor axis can only advance zero pixels or one pixel, we'll use an *integer threshold variable* (which we called an *error term variable* in line drawing) to determine when to advance along the minor axis. However, the circle drawing approach is a tad more complicated, because it involves squares.

Let's take a moment to work out the internals of our new, faster approach. This approach will also serve as a beautiful illustration of the point that you can only max out the performance of your code if you really understand what that code is doing (rather than blindly implementing an algorithm you've looked up), to the point where you can alter it into forms that are radically different and better suited to the PC, but nonetheless functionally equivalent.

An Incremental Circle Drawing Approach

Here's the trick to integer-only circle drawing: Use expansions of squared elements to keep track of when the minor axis should advance. In other words, rather than calculating each minor axis point independently through this expression

```
MinorAxis = sqrt(Radius² - MajorAxis²)
```

maintain the running states of **MinorAxis²** and **Radius² – MajorAxis²**, and decrement the minor axis by one pixel whenever **MinorAxis²** drops below **Radius² – MajorAxis²**.

Okay, I admit it's a *little* more complicated than that, but not much. Consider this: Suppose that we're drawing a circle of radius 100, centered about (0,0), and we're

drawing the arc starting at (0,100), as shown in Figure B.1. A simple multiply gives us $\mathbf{Radius}^2$ (10,000), which also happens to be $\mathbf{Radius}^2 - \mathbf{X}^2$, because **X** is zero. Now, as we advance **X** (the major axis) one pixel at a time, we want to know when it's time to advance **Y** one pixel; that occurs when **Y** (which equals $\mathbf{sqrt}(\mathbf{Radius}^2 - \mathbf{X}^2)$) drops below 99.5 (that is, moves more than halfway from the initial **Y** of 100 to the next pixel along the **Y** axis, at 99). Another way to express that is to say that we want to advance **Y** when $\mathbf{Y}^2$ (which equals $\mathbf{Radius}^2 - \mathbf{X}^2$) drops below 99.5^2—or, more conveniently for us, we want to advance **Y** when $\mathbf{Radius}^2 - \mathbf{X}^2$ drops below 9900.25.

Why is that convenient? Well, we already know the initial $\mathbf{Radius}^2 - \mathbf{X}^2$, which is 10,000. It's easy to maintain $\mathbf{Radius}^2 - \mathbf{X}^2$ as **X** advances, because $(\mathbf{X} + 1)^2$ is simply $\mathbf{X}^2 + 2\mathbf{X} + 1$, a simple integer calculation given $\mathbf{X}^2$ and **X**. Consequently, $\mathbf{Radius}^2 - (\mathbf{X} + 1)^2$ can be calculated as $\mathbf{Radius}^2 - (\mathbf{X}^2 + 2\mathbf{X} + 1)$, with no floating-point arithmetic needed.

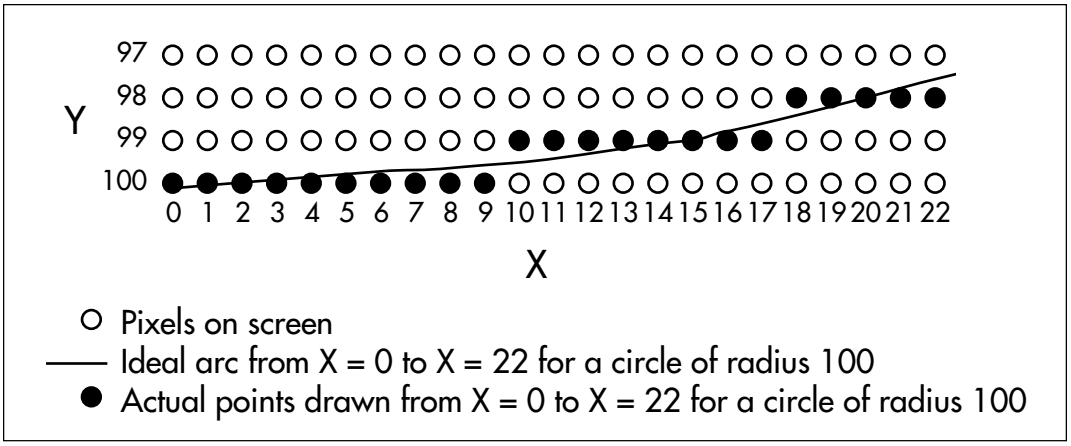
Now that we know $\mathbf{Radius}^2 - \mathbf{X}^2$ for each **X** as we move to the right, all we need to know is the threshold value for $\mathbf{Radius}^2 - \mathbf{X}^2$ below which we must advance **Y**. Initially, that threshold value is the value we calculated earlier, 9900.25, which is 99.5^2.

We seem to have a problem here, in that our threshold is not an integer. That's easily dealt with, however, as follows: First of all, that initial threshold is calculated as $(\mathbf{Y} - 0.5)^2$ (the square of the coordinate of the midpoint between **Y** and $\mathbf{Y} - 1$). That expression expands to $\mathbf{Y}^2 - \mathbf{Y} + 0.25$. Now, $\mathbf{Y}^2 - \mathbf{Y}$ is an easily-calculated integer expression; it's only the 0.25 that's a sticking point. It turns out, however, that we can just ignore the 0.25, because the $\mathbf{Radius}^2 - \mathbf{X}^2$ values we'll compare to the **Y** thresholds are always integers. If a given $\mathbf{Radius}^2 - \mathbf{X}^2$ value matches the integer portion of a $\mathbf{Y}^2 - \mathbf{Y} + 0.25$ value, we'll know that it actually is less than the threshold, because of the 0.25 we'll be carrying along.

For the example in Figure B.2, the initial point plotted is (0,100) relative to the center of the circle. (We'll always do our calculations relative to the center of the circle, and then adjust for the center of the circle and perform symmetry calculations later.) $\mathbf{Radius}^2 - \mathbf{X}^2$ is 10,000 at this point, and the initial threshold, $(\mathbf{Y} - 0.5)^2$, is 9900.25—but we'll use $\mathbf{Y}^2 - \mathbf{Y}$ to derive a working threshold of 9,900.

Next, **X** advances one pixel to coordinate 1. $\mathbf{Radius}^2 - \mathbf{X}^2$ goes to $10{,}000 - (2\mathbf{X} + 1)$, which is 9,999 (**X** refers to the last **X** drawn, which is zero at the moment.) That still exceeds the threshold of 9,900, so **Y** doesn't change. **X** then advances to coordinate 2. $\mathbf{Radius}^2 - \mathbf{X}^2$ goes to $9{,}999 - (2\mathbf{X} + 1)$, which is 9,996, so **Y** still doesn't advance. $\mathbf{Radius}^2 - \mathbf{X}^2$ then advances to 9,991, 9,984, 9,975, 9,964, 9,951, 9,936, 9,919, and finally to 9,900 as **X** advances to 10. Remember, we've only performed integer addition and subtraction as **X** has moved, and a mere two integer multiplications were required to set up the squared terms at the outset.

When **X** equals 10, $\mathbf{Radius}^2 - \mathbf{X}^2$ has finally matched our threshold for advancing **Y**; in fact, because of the implied 0.25 we've been carrying around, we know that $\mathbf{Radius}^2 - \mathbf{X}^2$ has passed the threshold. Consequently, we subtract 1 from **Y** to advance to the next



Incremental arc drawing.
Figure B.2

pixel along the **Y** axis, and draw the next point at (10,99). That's precisely what we want, since the rounded integer portion of **sqrt**($100^2 - 10^2$) (which is **Radius**^{2} – **X**^{2}) is 99.

Our only remaining task is to reset the threshold so that we know when to advance **Y** again. That's accomplished by calculating the new threshold as (**Y** – 1)^2; much as we did earlier with **X**, we can expand this to **Y**^{2} – 2**Y** + 1 in order to perform integer-only arithmetic. Given that we already know the **Y**^{2} for the threshold we just reached, it's a simple matter to calculate (**Y** – 1)^2 for the next threshold each time we advance **Y**.

There's a trick here, though: The **Y** we use in the threshold isn't the **Y** we just advanced to *or* the **Y** we just advanced from–it's the **Y** of the threshold we just reached; 99.5 is our sample case. Consequently, when **Y** advances to 99, the threshold advances to 9,900 – (99.5 * 2) + 1, which is 9,702. The 0.5 element doesn't mean we have to perform floating-point arithmetic, because 0.5 * 2 is always 1, so the above equation can also be expressed as 9,900 – (100 * 2 – 1) + 1. Consequently, the formula we'll use to advance the threshold is **Y**^{2} – 2**Y**, where **Y**^{2} is the last threshold value and **Y** is the **Y** coordinate of the last pixel drawn.

That's really all there is to it; reread the last section and work through an example or two of your own and you'll see how straightforward integer-only circle drawing is. It's remarkably easy to implement this approach, too; Listing B.3, which is such an implementation, is scarcely longer than Listing B.1. Listing B.3 is functionally equivalent to Listing B.1—except that it runs *more than five times faster than Listing B.1.*

Integer arithmetic is a wonderful thing, isn't it?

LISTING B.3 LB-3.C

```
/*
 * Draws a circle of the specified radius and color, using a fast
 * integer-only & square-root-free approach.
 * Compiles with Borland or Microsoft
 * Will work on VGA, or EGA in 16-color mode.
 */

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

/* Handle differences between Borland and Microsft compilers. 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 */

/* 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 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;
   unsigned long MinorAxisSquaredThreshold;

   /* 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 pointing to 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;

   /* Draw all points along an arc of 1/8th of the circle, drawing
      all 8 symmetries at the same time */
   do {
      /* Draw all 8 symmetries of current point */
      DrawDot(X+MajorAxis, Y-MinorAxis);
      DrawDot(X-MajorAxis, Y-MinorAxis);
      DrawDot(X+MajorAxis, Y+MinorAxis);
      DrawDot(X-MajorAxis, Y+MinorAxis);
      DrawDot(X+MinorAxis, Y-MajorAxis);
      DrawDot(X-MinorAxis, Y-MajorAxis);
      DrawDot(X+MinorAxis, Y+MajorAxis);
      DrawDot(X-MinorAxis, Y+MajorAxis);
      MajorAxis++;                      /* advance one pixel along major axis */
      if ( (RadiusSqMinusMajorAxisSq -=
         MajorAxis + MajorAxis - 1) <= MinorAxisSquaredThreshold ) {
         MinorAxis--;
         MinorAxisSquaredThreshold -= MinorAxis + MinorAxis;
      }
   } while ( MajorAxis <= MinorAxis );

   /* 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);
}
```

Notes on the Implementation

The obvious question about Listing B.3 is, Does it work? It's certainly fast, but if it doesn't draw circles properly, it's of little use.

I'm confident that Listing B.3 does indeed work. The theory is sound, and I compared all coordinates generated by Listing B.1 when linked to Listing B.2 to the coordinates generated by Listing B.3, and they were the same for all circles drawn.

Listing B.3 uses long integers, an unavoidable consequence of lugging around squared terms of values that exceed 255. Long arithmetic is relatively slow; it might be worthwhile to special-case circles with radii less than 256 and draw them with a separate routine that uses short integers.

Listing B.3 also treats the **Y** coordinates of circle centers as increasing from the top of the screen to the bottom, contrary to the normal graphing convention. If necessary,

the sense of the **Y** axis can be inverted by subtracting each **Y** coordinate from the screen height minus 1; that's the only adjustment necessary, because circles are symmetric about the **X** axis.

Continuing in Circles

That takes us halfway through our exploration of circle drawing. Coming up in the next chapter is the other portion of our optimization agenda: fine-tuning for the VGA and conversion to assembly. The end result should be an additional improvement of at least three times, bringing the total to at least *15 times* over that of Listing B.1!