Chapter 42

Wu'ed in Haste; Fried, Stewed at Leisure

Chapter 42

Fast Antialiased Lines Using Wu's Algorithm

The thought first popped into my head as I unenthusiastically picked through the salad bar at a local "family" restaurant, trying to decide whether the meatballs, the fried clams, or the lasagna was likely to shorten my life the least. I decided on the chicken in mystery sauce.

The thought recurred when my daughter asked, "Dad, is that fried chicken?"

"I don't think so," I said. "I think it's stewed chicken."

"It looks like fried chicken."

"Maybe it's fried, stewed chicken," my wife volunteered hopefully. I took a bite. It was, indeed, fried, stewed chicken. I can now, unhesitatingly and without reservation, recommend that you avoid fried, stewed chicken at all costs.

The thought I had was as follows: *This is not good food.* Not a profound thought, but it raises an interesting question: Why was I eating in this restaurant? The answer, to borrow a phrase from E.F. Schumacher, is *appropriate technology.* For a family on a budget, with a small child, tired of staring at each other over the kitchen table, this was a perfect place to eat. It was cheap, it had greasy food and ice cream, no one cared if children dropped things or talked loudly or walked around, and, most important of all, it wasn't home. So what if the food was lousy? Good food was a luxury, a bonus; everything on the above list was necessary. A family restaurant was the appropriate dining-out technology, given the parameters within which we had to work.

When I read through SIGGRAPH proceedings and other state-of-the-art computer-graphics material, all too often I feel like I'm dining at a four-star restaurant with two-year-old triplets and an empty wallet. We're talking incredibly inappropriate technology for PC graphics here. Sure, I say to myself as I read about an antialiasing technique, that sounds wonderful—if I had 24-bpp color, and dedicated hardware to do the processing, and all day to wait to generate one image. Yes, I think, that is a good way to do hidden surface removal—in a system with hardware z-buffering. Most of the stuff in the journal *Computer Graphics* is riveting, but, alas, pretty much useless on PCs. When an x86 has to do all the work, speed becomes the overriding parameter, especially for real-time graphics.

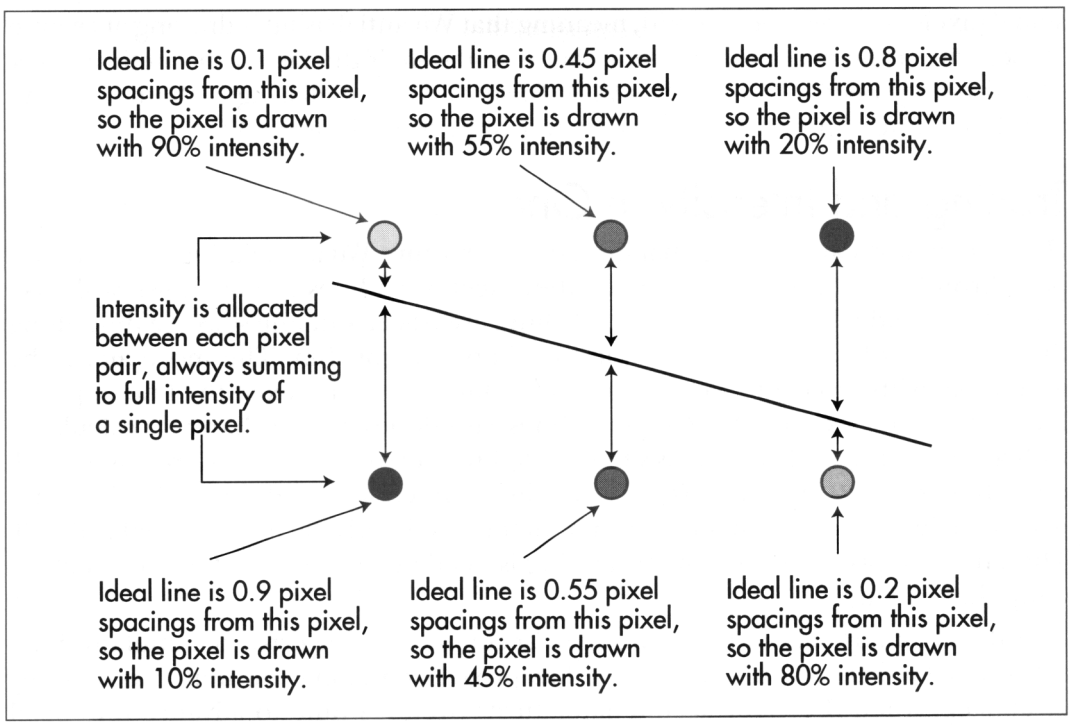
Literature that's applicable to fast PC graphics is hard enough to find, but what we'd really like is above-average image quality combined with terrific speed, and there's almost no literature of that sort around. There is some, however, and you folks are right on top of it. For example, alert reader Michael Chaplin, of San Diego, wrote to suggest that I might enjoy the line-antialiasing algorithm presented in Xiaolin Wu's article, "An Efficient Antialiasing Technique," in the July 1991 issue of *Computer Graphics*. Michael was dead-on right. This is a great algorithm, combining excellent antialiased line quality with speed that's close to that of non-antialiased Bresenham's line drawing. This is the sort of algorithm that makes you want to go out and write a wire-frame animation program, just so you can see how good those smooth lines look in motion. Wu antialiasing is a wonderful example of what can be accomplished on inexpensive, mass-market hardware with the proper programming perspective. In short, it's a splendid example of appropriate technology for PCs.

Wu Antialiasing

Antialiasing, as we've been discussing for the past few chapters, is the process of smoothing lines and edges so that they appear less jagged. Antialiasing is partly an aesthetic issue, because it makes images more attractive. It's also partly an accuracy issue, because it makes it possible to position and draw images with effectively more precision than the resolution of the display. Finally, it's partly a flat-out necessity, to avoid the horrible, crawling, jagged edges of temporal aliasing when performing animation.

The basic premise of Wu antialiasing is almost ridiculously simple: As the algorithm steps one pixel unit at a time along the major (longer) axis of a line, it draws the two pixels bracketing the line along the minor axis at each point. Each of the two bracketing pixels is drawn with a weighted fraction of the full intensity of the drawing color, with the weighting for each pixel equal to one minus the pixel's distance along the minor axis from the ideal line. Yes, it's a mouthful, but Figure 42.1 illustrates the concept.

The intensities of the two pixels that bracket the line are selected so that they always sum to exactly 1; that is, to the intensity of one fully illuminated pixel of the drawing color. The presence of aggregate full-pixel intensity means that at each step, the line



The basic concept of Wu antialiasing.
Figure 42.1

has the same brightness it would have if a single pixel were drawn at precisely the correct location. Moreover, thanks to the distribution of the intensity weighting, that brightness is centered at the ideal line. Not coincidentally, a line drawn with pixel pairs of aggregate single-pixel intensity, centered on the ideal line, is perceived by the eye not as a jagged collection of pixel pairs, but as a smooth line centered on the ideal line. Thus, by weighting the bracketing pixels properly at each step, we can readily produce what looks like a smooth line at precisely the right location, rather than the jagged pattern of line segments that non-antialiased line-drawing algorithms such as Bresenham's (see Chapters 35, 36, and 37) trace out.

You might expect that the implementation of Wu antialiasing would fall into two distinct areas: tracing out the line (that is, finding the appropriate pixel pairs to draw) and calculating the appropriate weightings for each pixel pair. Not so, however. The weighting calculations involve only a few shifts, XORs, and adds; for all practical purposes, tracing and weighting are rolled into one step—and a very fast step it is. How fast is it? On a 33-MHz 486 with a fast VGA, a good but not maxed-out assembly implementation of Wu antialiasing draws a more than respectable 5,000 150-pixel-long vectors per second. That's especially impressive considering that about 1,500,000

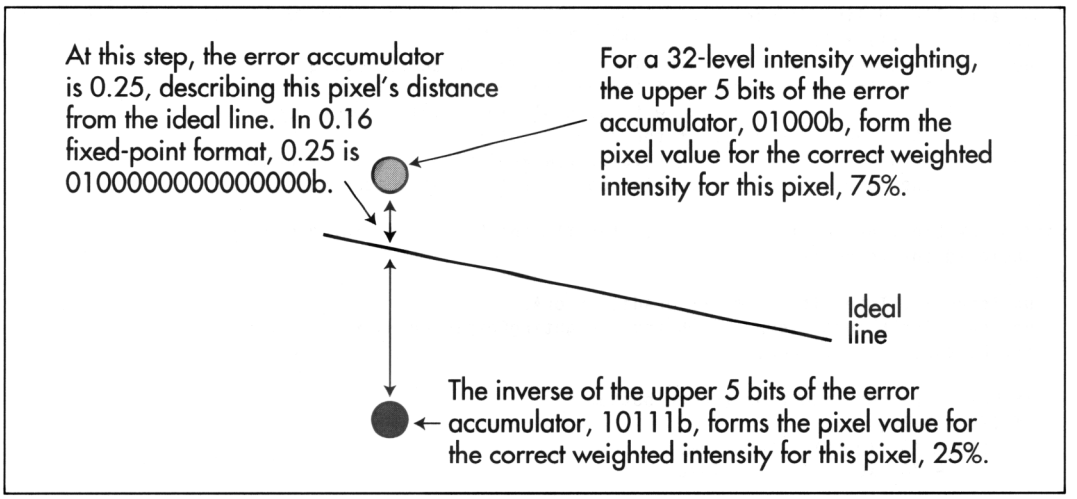
actual pixels are drawn per second, meaning that Wu antialiasing is drawing at around 50 percent of the maximum memory bandwidth—half the fastest theoretically possible drawing speed—of an AT-bus VGA. In short, Wu antialiasing is about as fast an antialiased line approach as you could ever hope to find for the VGA.

Tracing and Intensity in One

Horizontal, vertical, and diagonal lines do not require Wu antialiasing because they pass through the center of every pixel they meet; such lines can be drawn with fast, special-case code. For all other cases, Wu lines are traced out one step at a time along the major axis by means of a simple, fixed-point algorithm. The move along the minor axis with respect to a one-pixel move along the major axis (the line slope for lines with slopes less than 1, 1/slope for lines with slopes greater than 1) is calculated with a single integer divide. This value, called the "error adjust," is stored as a fixed-point fraction, in 0.16 format (that is, all bits are fractional, and the decimal point is just to the left of bit 15). An error accumulator, also in 0.16 format, is initialized to 0. Then the first pixel is drawn; no weighting is needed, because the line intersects its endpoints exactly.

Now the error adjust is added to the error accumulator. The error accumulator indicates how far between pixels the line has progressed along the minor axis at any given step; when the error accumulator turns over, it's time to advance one pixel along the minor axis. At each step along the line, the major-axis coordinate advances by one pixel. The two bracketing pixels to draw are simply the two pixels nearest the line along the minor axis. For instance, if X is the current major-axis coordinate and Y is the current minor-axis coordinate, the two pixels to be drawn are (X,Y) and (X,Y+1). In short, the derivation of the pixels at which to draw involves nothing more complicated than advancing one pixel along the major axis, adding the error adjust to the error accumulator, and advancing one pixel along the minor axis when the error accumulator turns over.

So far, nothing special; but now we come to the true wonder of Wu antialiasing. We know which pair of pixels to draw at each step along the line, but we also need to generate the two proper intensities, which must be inversely proportional to distance from the ideal line and sum to 1, and that's a potentially time-consuming operation. Let's assume, however, that the number of possible intensity levels to be used for weighting is the value NumLevels = 2^n for some integer n, with the minimum weighting (0 percent intensity) being the value 2^n-1, and the maximum weighting (100 percent intensity) being the value 0. Given that, lo and behold, the most significant n bits of the error accumulator select the proper intensity value for one element of the pixel pair, as shown in Figure 42.2. Better yet, 2^n-1 minus the intensity of the first pixel selects the intensity of the other pixel in the pair, because the intensities of the two pixels must sum to 1; as it happens, this result can be obtained simply by flipping the n least-significant bits of the first pixel's value. All this



Wu intensity calculations.
Figure 42.2

works because what the error accumulator accumulates is precisely the ideal line's current distance between the two bracketing pixels.

The intensity calculations take longer to describe than they do to perform. All that's involved is a shift of the error accumulator to right-justify the desired intensity weighting bits, and then an XOR to flip the least-significant n bits of the first pixel's value in order to generate the second pixel's value. Listing 42.1 illustrates just how efficient Wu antialiasing is; the intensity calculations take only three statements, and the entire Wu line-drawing loop is only nine statements long. Of course, a single C statement can hide a great deal of complexity, but Listing 42.6, an assembly implementation, shows that only 15 instructions are required per step along the major axis—and the number of instructions could be reduced to ten by special-casing and loop unrolling. Make no mistake about it, Wu antialiasing is fast.

LISTING 42.1 L42-1.C

```
/* Function to draw an antialiased line from (X0,Y0) to (X1,Y1), using an
 * antialiasing approach published by Xiaolin Wu in the July 1991 issue of
 * Computer Graphics. Requires that the palette be set up so that there
 * are NumLevels intensity levels of the desired drawing color, starting at
 * color BaseColor (100% intensity) and followed by (NumLevels-1) levels of
 * evenly decreasing intensity, with color (BaseColor+NumLevels-1) being 0%
 * intensity of the desired drawing color (black). This code is suitable for
 * use at screen resolutions, with lines typically no more than 1K long; for
 * longer lines, 32-bit error arithmetic must be used to avoid problems with
 * fixed-point inaccuracy. No clipping is performed in DrawWuLine; it must be
 * performed either at a higher level or in the DrawPixel function.
 * Tested with Borland C++ in C compilation mode and the small model.
 */
extern void DrawPixel(int, int, int);
```

```
/* Wu antialiased line drawer.
 * (X0,Y0),(X1,Y1) = line to draw
 * BaseColor = color # of first color in block used for antialiasing, the
 *          100% intensity version of the drawing color
 * NumLevels = size of color block, with BaseColor+NumLevels-1 being the
 *          0% intensity version of the drawing color
 * IntensityBits = log base 2 of NumLevels; the # of bits used to describe
 *          the intensity of the drawing color. 2**IntensityBits==NumLevels
 */
void DrawWuLine(int X0, int Y0, int X1, int Y1, int BaseColor, int NumLevels,
   unsigned int IntensityBits)
{
   unsigned int IntensityShift, ErrorAdj, ErrorAcc;
   unsigned int ErrorAccTemp, Weighting, WeightingComplementMask;
   int DeltaX, DeltaY, Temp, XDir;

   /* Make sure the line runs top to bottom */
   if (Y0 > Y1) {
      Temp = Y0; Y0 = Y1; Y1 = Temp;
      Temp = X0; X0 = X1; X1 = Temp;
   }
   /* Draw the initial pixel, which is always exactly intersected by
      the line and so needs no weighting */
   DrawPixel(X0, Y0, BaseColor);

   if ((DeltaX = X1 - X0) >= 0) {
      XDir = 1;
   } else {
      XDir = -1;
      DeltaX = -DeltaX; /* make DeltaX positive */
   }
   /* Special-case horizontal, vertical, and diagonal lines, which
      require no weighting because they go right through the center of
      every pixel */
   if ((DeltaY = Y1 - Y0) == 0) {
      /* Horizontal line */
      while (DeltaX-- != 0) {
         X0 += XDir;
         DrawPixel(X0, Y0, BaseColor);
      }
      return;
   }
   if (DeltaX == 0) {
      /* Vertical line */
      do {
         Y0++;
         DrawPixel(X0, Y0, BaseColor);
      } while (--DeltaY != 0);
      return;
   }
   if (DeltaX == DeltaY) {
      /* Diagonal line */
      do {
         X0 += XDir;
         Y0++;
         DrawPixel(X0, Y0, BaseColor);
      } while (--DeltaY != 0);
      return;
   }
   /* line is not horizontal, diagonal, or vertical */
   ErrorAcc = 0;  /* initialize the line error accumulator to 0 */
```

```
   /* # of bits by which to shift ErrorAcc to get intensity level */
   IntensityShift = 16 - IntensityBits;
   /* Mask used to flip all bits in an intensity weighting, producing the
      result (1 - intensity weighting) */
   WeightingComplementMask = NumLevels - 1;
   /* Is this an X-major or Y-major line? */
   if (DeltaY > DeltaX) {
      /* Y-major line; calculate 16-bit fixed-point fractional part of a
         pixel that X advances each time Y advances 1 pixel, truncating the
         result so that we won't overrun the endpoint along the X axis */
      ErrorAdj = ((unsigned long) DeltaX << 16) / (unsigned long) DeltaY;
      /* Draw all pixels other than the first and last */
      while (--DeltaY) {
         ErrorAccTemp = ErrorAcc;   /* remember currrent accumulated error */
         ErrorAcc += ErrorAdj;      /* calculate error for next pixel */
         if (ErrorAcc <= ErrorAccTemp) {
            /* The error accumulator turned over, so advance the X coord */
            X0 += XDir;
         }
         Y0++; /* Y-major, so always advance Y */
         /* The IntensityBits most significant bits of ErrorAcc give us the
            intensity weighting for this pixel, and the complement of the
            weighting for the paired pixel */
         Weighting = ErrorAcc >> IntensityShift;
         DrawPixel(X0, Y0, BaseColor + Weighting);
         DrawPixel(X0 + XDir, Y0,
               BaseColor + (Weighting ^ WeightingComplementMask));
      }
      /* Draw the final pixel, which is always exactly intersected by the line
         and so needs no weighting */
      DrawPixel(X1, Y1, BaseColor);
      return;
   }
   /* It's an X-major line; calculate 16-bit fixed-point fractional part of a
      pixel that Y advances each time X advances 1 pixel, truncating the
      result to avoid overrunning the endpoint along the X axis */
   ErrorAdj = ((unsigned long) DeltaY << 16) / (unsigned long) DeltaX;
   /* Draw all pixels other than the first and last */
   while (--DeltaX) {
      ErrorAccTemp = ErrorAcc;   /* remember currrent accumulated error */
      ErrorAcc += ErrorAdj;      /* calculate error for next pixel */
      if (ErrorAcc <= ErrorAccTemp) {
         /* The error accumulator turned over, so advance the Y coord */
         Y0++;
      }
      X0 += XDir; /* X-major, so always advance X */
      /* The IntensityBits most significant bits of ErrorAcc give us the
         intensity weighting for this pixel, and the complement of the
         weighting for the paired pixel */
      Weighting = ErrorAcc >> IntensityShift;
      DrawPixel(X0, Y0, BaseColor + Weighting);
      DrawPixel(X0, Y0 + 1,
            BaseColor + (Weighting ^ WeightingComplementMask));
   }
   /* Draw the final pixel, which is always exactly intersected by the line
      and so needs no weighting */
   DrawPixel(X1, Y1, BaseColor);
}
```

Sample Wu Antialiasing

The true test of any antialiasing technique is how good it looks, so let's have a look at Wu antialiasing in action. Listing 42.1 is a C implementation of Wu antialiasing. Listing 42.2 is a sample program that draws a variety of Wu-antialiased lines, followed by non-antialiased lines, for comparison. Listing 42.3 contains **DrawPixel()** and **SetMode()** functions for mode 13H, the VGA's 320×200 256-color mode. Finally, Listing 42.4 is a simple, non-antialiased line-drawing routine. Link these four listings together and run the resulting program to see both Wu-antialiased and non-antialiased lines.

LISTING 42.2 L42-2.C

```
/* Sample line-drawing program to demonstrate Wu antialiasing. Also draws
 * non-antialiased lines for comparison.
 * Tested with Borland C++ in C compilation mode and the small model.
 */
#include <dos.h>
#include <conio.h>

void SetPalette(struct WuColor *);
extern void DrawWuLine(int, int, int, int, int, int, unsigned int);
extern void DrawLine(int, int, int, int, int);
extern void SetMode(void);
extern int ScreenWidthInPixels;  /* screen dimension globals */
extern int ScreenHeightInPixels;

#define NUM_WU_COLORS 2 /* # of colors we'll do antialiased drawing with */
struct WuColor {        /* describes one color used for antialiasing */
   int BaseColor;       /* # of start of palette intensity block in DAC */
   int NumLevels;       /* # of intensity levels */
   int IntensityBits;   /* IntensityBits == log2 NumLevels */
   int MaxRed;          /* red component of color at full intensity */
   int MaxGreen;        /* green component of color at full intensity */
   int MaxBlue;         /* blue component of color at full intensity */
};
enum {WU_BLUE=0, WU_WHITE=1};             /* drawing colors */
struct WuColor WuColors[NUM_WU_COLORS] =  /* blue and white */
    {{192, 32, 5, 0, 0, 0x3F}, {224, 32, 5, 0x3F, 0x3F, 0x3F}};

void main()
{
   int CurrentColor, i;
   union REGS regset;

   /* Draw Wu-antialiased lines in all directions */
   SetMode();
   SetPalette(WuColors);
   for (i=5; i<ScreenWidthInPixels; i += 10) {
      DrawWuLine(ScreenWidthInPixels/2-ScreenWidthInPixels/10+i/5,
            ScreenHeightInPixels/5, i, ScreenHeightInPixels-1,
            WuColors[WU_BLUE].BaseColor, WuColors[WU_BLUE].NumLevels,
            WuColors[WU_BLUE].IntensityBits);
   }
   for (i=0; i<ScreenHeightInPixels; i += 10) {
      DrawWuLine(ScreenWidthInPixels/2-ScreenWidthInPixels/10, i/5, 0, i,
            WuColors[WU_BLUE].BaseColor, WuColors[WU_BLUE].NumLevels,
            WuColors[WU_BLUE].IntensityBits);
   }
```

```
   for (i=0; i<ScreenHeightInPixels; i += 10) {
      DrawWuLine(ScreenWidthInPixels/2+ScreenWidthInPixels/10, i/5,
            ScreenWidthInPixels-1, i, WuColors[WU_BLUE].BaseColor,
            WuColors[WU_BLUE].NumLevels, WuColors[WU_BLUE].IntensityBits);
   }
   for (i=0; i<ScreenWidthInPixels; i += 10) {
      DrawWuLine(ScreenWidthInPixels/2-ScreenWidthInPixels/10+i/5,
            ScreenHeightInPixels, i, 0, WuColors[WU_WHITE].BaseColor,
            WuColors[WU_WHITE].NumLevels,
            WuColors[WU_WHITE].IntensityBits);
   }
   getch();                /* wait for a key press */

   /* Now clear the screen and draw non-antialiased lines */
   SetMode();
   SetPalette(WuColors);
   for (i=0; i<ScreenWidthInPixels; i += 10) {
      DrawLine(ScreenWidthInPixels/2-ScreenWidthInPixels/10+i/5,
            ScreenHeightInPixels/5, i, ScreenHeightInPixels-1,
            WuColors[WU_BLUE].BaseColor);
   }
   for (i=0; i<ScreenHeightInPixels; i += 10) {
      DrawLine(ScreenWidthInPixels/2-ScreenWidthInPixels/10, i/5, 0, i,
            WuColors[WU_BLUE].BaseColor);
   }
   for (i=0; i<ScreenHeightInPixels; i += 10) {
      DrawLine(ScreenWidthInPixels/2+ScreenWidthInPixels/10, i/5,
            ScreenWidthInPixels-1, i, WuColors[WU_BLUE].BaseColor);
   }
   for (i=0; i<ScreenWidthInPixels; i += 10) {
      DrawLine(ScreenWidthInPixels/2-ScreenWidthInPixels/10+i/5,
            ScreenHeightInPixels, i, 0, WuColors[WU_WHITE].BaseColor);
   }
   getch();                /* wait for a key press */

   regset.x.ax = 0x0003;   /* AL = 3 selects 80x25 text mode */
   int86(0x10, &regset, &regset);   /* return to text mode */
}

/* Sets up the palette for antialiasing with the specified colors.
 * Intensity steps for each color are scaled from the full desired intensity
 * of the red, green, and blue components for that color down to 0%
 * intensity; each step is rounded to the nearest integer. Colors are
 * corrected for a gamma of 2.3. The values that the palette is programmed
 * with are hardwired for the VGA's 6 bit per color DAC.
 */
void SetPalette(struct WuColor * WColors)
{
   int i, j;
   union REGS regset;
   struct SREGS sregset;
   static unsigned char PaletteBlock[256][3];   /* 256 RGB entries */
   /* Gamma-corrected DAC color components for 64 linear levels from 0% to
      100% intensity */
   static unsigned char GammaTable[] = {
      0, 10, 14, 17, 19, 21, 23, 24, 26, 27, 28, 29, 31, 32, 33, 34,
      35, 36, 37, 37, 38, 39, 40, 41, 41, 42, 43, 44, 44, 45, 46, 46,
      47, 48, 48, 49, 49, 50, 51, 51, 52, 52, 53, 53, 54, 54, 55, 55,
      56, 56, 57, 57, 58, 58, 59, 59, 60, 60, 61, 61, 62, 62, 63, 63};
```

```
   for (i=0; i<NUM_WU_COLORS; i++) {
      for (j=0; j<WColors[i].NumLevels; j++) {
         PaletteBlock[j][0] = GammaTable[((double)WColors[i].MaxRed * (1.0 -
               (double)j / (double)(WColors[i].NumLevels - 1))) + 0.5];
         PaletteBlock[j][1] = GammaTable[((double)WColors[i].MaxGreen * (1.0 -
               (double)j / (double)(WColors[i].NumLevels - 1))) + 0.5];
         PaletteBlock[j][2] = GammaTable[((double)WColors[i].MaxBlue * (1.0 -
               (double)j / (double)(WColors[i].NumLevels - 1))) + 0.5];
      }
      /* Now set up the palette to do Wu antialiasing for this color */
      regset.x.ax = 0x1012;   /* set block of DAC registers function */
      regset.x.bx = WColors[i].BaseColor;   /* first DAC location to load */
      regset.x.cx = WColors[i].NumLevels;   /* # of DAC locations to load */
      regset.x.dx = (unsigned int)PaletteBlock; /* offset of array from which
                                                   to load RGB settings */
      sregset.es = _DS; /* segment of array from which to load settings */
      int86x(0x10, &regset, &regset, &sregset); /* load the palette block */
   }
}
```

LISTING 42.3 L42-3.C

```
/* VGA mode 13h pixel-drawing and mode set functions.
 * Tested with Borland C++ in C compilation mode and the small model.
 */
#include <dos.h>

/* Screen dimension globals, used in main program to scale. */
int ScreenWidthInPixels = 320;
int ScreenHeightInPixels = 200;

/* Mode 13h draw pixel function. */
void DrawPixel(int X, int Y, int Color)
{
#define SCREEN_SEGMENT  0xA000
   unsigned char far *ScreenPtr;

   FP_SEG(ScreenPtr) = SCREEN_SEGMENT;
   FP_OFF(ScreenPtr) = (unsigned int) Y * ScreenWidthInPixels + X;
   *ScreenPtr = Color;
}

/* Mode 13h mode-set function. */
void SetMode()
{
   union REGS regset;

   /* Set to 320x200 256-color graphics mode */
   regset.x.ax = 0x0013;
   int86(0x10, &regset, &regset);
}
```

LISTING 42.4 L42-4.C

```
/* Function to draw a non-antialiased line from (X0,Y0) to (X1,Y1), using a
 * simple fixed-point error accumulation approach.
 * Tested with Borland C++ in C compilation mode and the small model.
 */
extern void DrawPixel(int, int, int);
```

```
/* Non-antialiased line drawer.
 * (X0,Y0),(X1,Y1) = line to draw, Color = color in which to draw
 */
void DrawLine(int X0, int Y0, int X1, int Y1, int Color)
{
   unsigned long ErrorAcc, ErrorAdj;
   int DeltaX, DeltaY, XDir, Temp;

   /* Make sure the line runs top to bottom */
   if (Y0 > Y1) {
      Temp = Y0; Y0 = Y1; Y1 = Temp;
      Temp = X0; X0 = X1; X1 = Temp;
   }
   DrawPixel(X0, Y0, Color);  /* draw the initial pixel */
   if ((DeltaX = X1 - X0) >= 0) {
      XDir = 1;
   } else {
      XDir = -1;
      DeltaX = -DeltaX; /* make DeltaX positive */
   }
   if ((DeltaY = Y1 - Y0) == 0)  /* done if only one point in the line */
      if (DeltaX == 0) return;

   ErrorAcc = 0x8000;   /* initialize line error accumulator to .5, so we can
                           advance when we get halfway to the next pixel */
   /* Is this an X-major or Y-major line? */
   if (DeltaY > DeltaX) {
      /* Y-major line; calculate 16-bit fixed-point fractional part of a
         pixel that X advances each time Y advances 1 pixel */
      ErrorAdj = ((((unsigned long)DeltaX << 17) / (unsigned long)DeltaY) +
            1) >> 1;
      /* Draw all pixels between the first and last */
      do {
         ErrorAcc += ErrorAdj;      /* calculate error for this pixel */
         if (ErrorAcc & ~0xFFFFL) {
            /* The error accumulator turned over, so advance the X coord */
            X0 += XDir;
            ErrorAcc &= 0xFFFFL;    /* clear integer part of result */
         }
         Y0++;                      /* Y-major, so always advance Y */
         DrawPixel(X0, Y0, Color);
      } while (--DeltaY);
      return;
   }
   /* It's an X-major line; calculate 16-bit fixed-point fractional part of a
      pixel that Y advances each time X advances 1 pixel */
   ErrorAdj = ((((unsigned long)DeltaY << 17) / (unsigned long)DeltaX) +
         1) >> 1;
   /* Draw all remaining pixels */
   do {
      ErrorAcc += ErrorAdj;      /* calculate error for this pixel */
      if (ErrorAcc & ~0xFFFFL) {
         /* The error accumulator turned over, so advance the Y coord */
         Y0++;
         ErrorAcc &= 0xFFFFL;    /* clear integer part of result */
      }
      X0 += XDir;                /* X-major, so always advance X */
      DrawPixel(X0, Y0, Color);
   } while (--DeltaX);
}
```

Listing 42.1 isn't particularly fast, because it calls **DrawPixel()** for each pixel. On the other hand, **DrawPixel()** makes it easy to try out Wu antialiasing in a variety of modes; just adapt the code in Listing 42.3 for the 256-color mode you want to support. For example, Listing 42.5 shows code to draw Wu-antialiased lines in 640×480 256-color mode on SuperVGAs built around the Tseng Labs ET4000 chip with at least 512K of display memory installed. It's well worth checking out Wu antialiasing at 640×480. Although antialiased lines look much smoother than normal lines at 320×200 resolution, they're far from perfect, because the pixels are so big that the eye can't blend them properly. At 640×480, however, Wu-antialiased lines look fabulous; from a couple of feet away, they look as straight and smooth as if they were drawn with a ruler.

LISTING 42.5 L42-5.C

```
/* Mode set and pixel-drawing functions for the 640x480 256-color mode of
 * Tseng Labs ET4000-based SuperVGAs.
 * Tested with Borland C++ in C compilation mode and the small model.
 */
#include <dos.h>

/* Screen dimension globals, used in main program to scale */
int ScreenWidthInPixels = 640;
int ScreenHeightInPixels = 480;

/* ET4000 640x480 256-color draw pixel function. */
void DrawPixel(int X, int Y, int Color)
{
#define SCREEN_SEGMENT        0xA000
#define GC_SEGMENT_SELECT     0x3CD /* ET4000 segment (bank) select reg */
   unsigned char far *ScreenPtr;
   unsigned int Bank;
   unsigned long BitmapAddress;

   /* full bitmap address of pixel, as measured from address 0 to 0xFFFFF */
   BitmapAddress = (unsigned long) Y * ScreenWidthInPixels + X;
   /* Bank # is upper word of bitmap addr */
   Bank = BitmapAddress >> 16;
   /* Upper nibble is read bank #, lower nibble is write bank # */
   outp(GC_SEGMENT_SELECT, (Bank << 4) | Bank);
   /* Draw into the bank */
   FP_SEG(ScreenPtr) = SCREEN_SEGMENT;
   FP_OFF(ScreenPtr) = (unsigned int) BitmapAddress;
   *ScreenPtr = Color;
}

/* ET4000 640x480 256-color mode-set function. */
void SetMode()
{
   union REGS regset;

   /* Set to 640x480 256-color graphics mode */
   regset.x.ax = 0x002E;
   int86(0x10, &regset, &regset);
}
```

Listing 42.1 requires that the DAC palette be set up so that a **NumLevel**-long block of palette entries contains linearly decreasing intensities of the drawing color. The size

of the block is programmable, but must be a power of two. The more intensity levels, the better. Wu says that 32 intensities are enough; on my system, eight and even four levels looked pretty good. I found that gamma correction, which gives linearly spaced intensity steps, improved antialiasing quality significantly. Fortunately, we can program the palette with gamma-corrected values, so our drawing code doesn't have to do any extra work.

Listing 42.1 isn't very fast, so I implemented Wu antialiasing in assembly, hard-coded for mode 13H. The implementation is shown in full in Listing 42.6. High-speed graphics code and fast VGAs go together like peanut butter and jelly, which is to say very well indeed; the assembly implementation ran more than twice as fast as the C code on my 486. Enough said!

LISTING 42.6 L42-6.ASM

```
; C near-callable function to draw an antialiased line from
; (X0,Y0) to (X1,Y1), in mode 13h, the VGA's standard 320x200 256-color
; mode. Uses an antialiasing approach published by Xiaolin Wu in the July
; 1991 issue of Computer Graphics. Requires that the palette be set up so
; that there are NumLevels intensity levels of the desired drawing color,
; starting at color BaseColor (100% intensity) and followed by (NumLevels-1)
; levels of evenly decreasing intensity, with color (BaseColor+NumLevels-1)
; being 0% intensity of the desired drawing color (black). No clipping is
; performed in DrawWuLine. Handles a maximum of 256 intensity levels per
; antialiased color. This code is suitable for use at screen resolutions,
; with lines typically no more than 1K long; for longer lines, 32-bit error
; arithmetic must be used to avoid problems with fixed-point inaccuracy.
; Tested with TASM.
;
; C near-callable as:
;   void DrawWuLine(int X0, int Y0, int X1, int Y1, int BaseColor,
;      int NumLevels, unsigned int IntensityBits);

SCREEN_WIDTH_IN_BYTES equ 320      ;# of bytes from the start of one scan line
                                   ; to the start of the next
SCREEN_SEGMENT   equ   0a000h      ;segment in which screen memory resides

; Parameters passed in stack frame.
parms   struc
      dw    2 dup (?)   ;pushed BP and return address
X0    dw    ?           ;X coordinate of line start point
Y0    dw    ?           ;Y coordinate of line start point
X1    dw    ?           ;X coordinate of line end point
Y1    dw    ?           ;Y coordinate of line end point
BaseColor dw     ?           ;color # of first color in block used for
                       ;antialiasing, the 100% intensity version of the
                       ;drawing color
NumLevels dw     ?           ;size of color block, with BaseColor+NumLevels-1
                       ; being the 0% intensity version of the drawing color
                       ; (maximum NumLevels = 256)
IntensityBits dw ?           ;log base 2 of NumLevels; the # of bits used to
                       ; describe the intensity of the drawing color.
                       ; 2**IntensityBits=NumLevels
                       ; (maximum IntensityBits = 8)
parms ends
```

```
.model      small
.code
; Screen dimension globals, used in main program to scale.
_ScreenWidthInPixels    dw      320
_ScreenHeightInPixels   dw      200

      .code
      public   _DrawWuLine
_DrawWuLine proc near
      push  bp          ;preserve caller's stack frame
      mov   bp,sp       ;point to local stack frame
      push  si          ;preserve C's register variables
      push  di
      push  ds          ;preserve C's default data segment
      cld               ;make string instructions increment their pointers

; Make sure the line runs top to bottom.
      mov   si,[bp].X0
      mov   ax,[bp].Y0
      cmp   ax,[bp].Y1  ;swap endpoints if necessary to ensure that
      jna   NoSwap      ; Y0 <= Y1
      xchg  [bp].Y1,ax
      mov   [bp].Y0,ax
      xchg  [bp].X1,si
      mov   [bp].X0,si
NoSwap:

; Draw the initial pixel, which is always exactly intersected by the line
; and so needs no weighting.
      mov   dx,SCREEN_SEGMENT
      mov   ds,dx             ;point DS to the screen segment
      mov   dx,SCREEN_WIDTH_IN_BYTES
      mul   dx                ;Y0 * SCREEN_WIDTH_IN_BYTES yields the offset
                              ; of the start of the row start the initial
                              ; pixel is on
      add   si,ax             ;point DS:SI to the initial pixel
      mov   al,byte ptr [bp].BaseColor   ;color with which to draw
      mov   [si],al           ;draw the initial pixel

      mov   bx,1              ;XDir = 1; assume DeltaX >= 0
      mov   cx,[bp].X1
      sub   cx,[bp].X0        ;DeltaX; is it >= 1?
      jns   DeltaXSet         ;yes, move left->right, all set
                              ;no, move right->left
      neg   cx                ;make DeltaX positive
      neg   bx                ;XDir = -1
DeltaXSet:

; Special-case horizontal, vertical, and diagonal lines, which require no
; weighting because they go right through the center of every pixel.
      mov   dx,[bp].Y1
      sub   dx,[bp].Y0        ;DeltaY; is it 0?
      jnz   NotHorz           ;no, not horizontal
                              ;yes, is horizontal, special case
      and   bx,bx             ;draw from left->right?
      jns   DoHorz            ;yes, all set
      std                     ;no, draw right->left
DoHorz:
      lea   di,[bx+si]        ;point DI to next pixel to draw
      mov   ax,ds
```

```
      mov   es,ax             ;point ES:DI to next pixel to draw
      mov   al,byte ptr [bp].BaseColor ;color with which to draw
                              ;CX = DeltaX at this point
      rep   stosb             ;draw the rest of the horizontal line
      cld                                     ;restore default direction flag
      jmp   Done                              ;and we're done

      align 2
NotHorz:
      and   cx,cx                             ;is DeltaX 0?
      jnz   NotVert                           ;no, not a vertical line
                                              ;yes, is vertical, special case
      mov   al,byte ptr [bp].BaseColor        ;color with which to draw
VertLoop:
      add   si,SCREEN_WIDTH_IN_BYTES          ;point to next pixel to draw
      mov   [si],al                           ;draw the next pixel
      dec   dx                                ;--DeltaY
      jnz   VertLoop
      jmp   Done                              ;and we're done

      align 2
NotVert:
      cmp   cx,dx                             ;DeltaX == DeltaY?
      jnz   NotDiag                           ;no, not diagonal
                                              ;yes, is diagonal, special case
      mov   al,byte ptr [bp].BaseColor   ;color with which to draw
DiagLoop:
      lea   si,[si+SCREEN_WIDTH_IN_BYTES+bx]
                                              ;advance to next pixel to draw by
                                              ; incrementing Y and adding XDir to X
      mov   [si],al                           ;draw the next pixel
      dec   dx                                ;--DeltaY
      jnz   DiagLoop
      jmp   Done                              ;and we're done

; Line is not horizontal, diagonal, or vertical.
      align 2
NotDiag:
; Is this an X-major or Y-major line?
      cmp   dx,cx
      jb    XMajor                            ;it's X-major

; It's a Y-major line. Calculate the 16-bit fixed-point fractional part of a
; pixel that X advances each time Y advances 1 pixel, truncating the result
; to avoid overrunning the endpoint along the X axis.
      xchg  dx,cx             ;DX = DeltaX, CX = DeltaY
      sub   ax,ax             ;make DeltaX 16.16 fixed-point value in DX:AX
      div   cx                ;AX = (DeltaX << 16) / DeltaY. Won't overflow
                              ; because DeltaX < DeltaY
      mov   di,cx             ;DI = DeltaY (loop count)
      sub   si,bx             ;back up the start X by 1, as explained below
      mov   dx,-1             ;initialize the line error accumulator to -1,
                              ; so that it will turn over immediately and
                              ; advance X to the start X. This is necessary
                              ; properly to bias error sums of 0 to mean
                              ; "advance next time" rather than "advance
                              ; this time," so that the final error sum can
                              ; never cause drawing to overrun the final X
                              ; coordinate (works in conjunction with
                              ; truncating ErrorAdj, to make sure X can't
                              ; overrun)
```

```
        mov   cx,8                          ;CL = # of bits by which to shift
        sub   cx,[bp].IntensityBits         ; ErrorAcc to get intensity level (8
                                            ; instead of 16 because we work only
                                            ; with the high byte of ErrorAcc)
        mov   ch,byte ptr [bp].NumLevels    ;mask used to flip all bits in an
        dec   ch                            ; intensity weighting, producing
                                            ; result (1 - intensity weighting)
        mov   bp,BaseColor[bp]              ;***stack frame not available***
                                            ;***from now on              ***
        xchg  bp,ax                         ;BP = ErrorAdj, AL = BaseColor,
                                            ; AH = scratch register

; Draw all remaining pixels.
YMajorLoop:
        add   dx,bp                         ;calculate error for next pixel
        jnc   NoXAdvance                    ;not time to step in X yet
                                            ;the error accumulator turned over,
                                            ;so advance the X coord
        add   si,bx                         ;add XDir to the pixel pointer
NoXAdvance:
        add   si,SCREEN_WIDTH_IN_BYTES      ;Y-major, so always advance Y

; The IntensityBits most significant bits of ErrorAcc give us the intensity
; weighting for this pixel, and the complement of the weighting for the
; paired pixel.
        mov   ah,dh                 ;msb of ErrorAcc
        shr   ah,cl                 ;Weighting = ErrorAcc >> IntensityShift;
        add   ah,al                 ;BaseColor + Weighting
        mov   [si],ah               ;DrawPixel(X, Y, BaseColor + Weighting);
        mov   ah,dh                 ;msb of ErrorAcc
        shr   ah,cl                 ;Weighting = ErrorAcc >> IntensityShift;
        xor   ah,ch                 ;Weighting ^ WeightingComplementMask
        add   ah,al                 ;BaseColor + (Weighting ^ WeightingComplementMask)
        mov   [si+bx],ah            ;DrawPixel(X+XDir, Y,
; BaseColor + (Weighting ^ WeightingComplementMask));
        dec   di                    ;--DeltaY
        jnz   YMajorLoop
        jmp   Done                  ;we're done with this line

; It's an X-major line.
        align 2
XMajor:
; Calculate the 16-bit fixed-point fractional part of a pixel that Y advances
; each time X advances 1 pixel, truncating the result to avoid overrunning
; the endpoint along the X axis.
        sub   ax,ax                 ;make DeltaY 16.16 fixed-point value in DX:AX
        div   cx                    ;AX = (DeltaY << 16) / Deltax. Won't overflow
                                    ; because DeltaY < DeltaX
        mov   di,cx                 ;DI = DeltaX (loop count)
        sub   si,SCREEN_WIDTH_IN_BYTES    ;back up the start X by 1, as
                                    ; explained below
        mov   dx,-1                 ;initialize the line error accumulator to -1,
                                    ; so that it will turn over immediately and
                                    ; advance Y to the start Y. This is necessary
                                    ; properly to bias error sums of 0 to mean
                                    ; "advance next time" rather than "advance
                                    ; this time," so that the final error sum can
                                    ; never cause drawing to overrun the final Y
                                    ; coordinate (works in conjunction with
                                    ; truncating ErrorAdj, to make sure Y can't
                                    ; overrun)
```

```
      mov   cx,8                        ;CL = # of bits by which to shift
      sub   cx,[bp].IntensityBits       ; ErrorAcc to get intensity level (8
                                        ; instead of 16 because we work only
                                        ; with the high byte of ErrorAcc)
      mov   ch,byte ptr [bp].NumLevels  ;mask used to flip all bits in an
      dec   ch                          ; intensity weighting, producing
                                        ; result (1 - intensity weighting)
      mov   bp,BaseColor[bp]            ;***stack frame not available***
                                        ;***from now on              ***
      xchg  bp,ax                       ;BP = ErrorAdj, AL = BaseColor,
                                        ; AH = scratch register
; Draw all remaining pixels.
XMajorLoop:
      add   dx,bp                       ;calculate error for next pixel
      jnc   NoYAdvance                  ;not time to step in Y yet
                                        ;the error accumulator turned over,
                                        ; so advance the Y coord
      add   si,SCREEN_WIDTH_IN_BYTES    ;advance Y
NoYAdvance:
      add   si,bx                       ;X-major, so add XDir to the pixel pointer

; The IntensityBits most significant bits of ErrorAcc give us the intensity
; weighting for this pixel, and the complement of the weighting for the
; paired pixel.
      mov   ah,dh                       ;msb of ErrorAcc
      shr   ah,cl                       ;Weighting = ErrorAcc >> IntensityShift;
      add   ah,al                       ;BaseColor + Weighting
      mov   [si],ah                     ;DrawPixel(X, Y, BaseColor + Weighting);
      mov   ah,dh                       ;msb of ErrorAcc
      shr   ah,cl                       ;Weighting = ErrorAcc >> IntensityShift;
      xor   ah,ch                       ;Weighting ^ WeightingComplementMask
      add   ah,al                       ;BaseColor + (Weighting ^ WeightingComplementMask)
      mov   [si+SCREEN_WIDTH_IN_BYTES],ah
 ;DrawPixel(X, Y+SCREEN_WIDTH_IN_BYTES,
 ; BaseColor + (Weighting ^ WeightingComplementMask));
      dec   di                          ;--DeltaX
      jnz   XMajorLoop

Done:                                   ;we're done with this line
      pop   ds                          ;restore C's default data segment
      pop   di                          ;restore C's register variables
      pop   si
      pop   bp                          ;restore caller's stack frame
      ret                               ;done
_DrawWuLine endp
      end
```

Notes on Wu Antialiasing

Wu antialiasing can be applied to any curve for which it's possible to calculate at each step the positions and intensities of two bracketing pixels, although the implementation will generally be nowhere near as efficient as it is for lines. However, Wu's article in *Computer Graphics* does describe an efficient algorithm for drawing antialiased circles. Wu also describes a technique for antialiasing solids, such as filled circles and polygons. Wu's approach biases the edges of filled objects outward. Although this is no good for adjacent polygons of the sort used in rendering, it's certainly possible to

design a more accurate polygon-antialiasing approach around Wu's basic weighting technique. The results would not be quite so good as more sophisticated antialiasing techniques, but they would be much faster.

In general, the results obtained by Wu antialiasing are only so-so, by theoretical measures. Wu antialiasing amounts to a simple box filter placed over a fixed-point step approximation of a line, and that process introduces a good deal of deviation from the ideal. On the other hand, Wu notes that even a 10 percent error in intensity doesn't lead to noticeable loss of image quality, and for Wu-antialiased lines up to 1K pixels in length, the error is under 10 percent. If it looks good, it is good—and it looks good.

With a 16-bit error accumulator, fixed-point inaccuracy becomes a problem for Wu-antialiased lines longer than 1K. For such lines, you should switch to using 32-bit error values, which would let you handle lines of any practical length.

In the listings, I have chosen to truncate, rather than round, the error-adjust value. This increases the intensity error of the line but guarantees that fixed-point inaccuracy won't cause the minor axis to advance past the endpoint. Overrunning the endpoint would result in the drawing of pixels outside the line's bounding box, and potentially even in an attempt to access pixels off the edge of the bitmap.

Finally, I should mention that, as published, Wu's algorithm draws lines symmetrically, from both ends at once. I haven't done this for a number of reasons, not least of which is that symmetric drawing is an inefficient way to draw lines that span banks on banked Super-VGAs. Banking aside, however, symmetric drawing is potentially faster, because it eliminates half of all calculations; in so doing, it cuts cumulative error in half, as well.

With or without symmetrical processing, Wu antialiasing beats fried, stewed chicken hands-down. Trust me on this one.