# Chapter F

# The VGA versus the Jaggies

# Chapter F

## The Sierra Hicolor DAC as the Means to Antialiased Lines

Does anything evolve faster than PC hardware? Not so long ago, we thought that juicing up the crystal in an AT from 6 to 8MHz was awesome. Now we take it for granted when Intel triples the internal speed of a 33MHz CPU, and with the Pentium's superscalar execution and 90MHz clock speeds, the trend is, if anything, accelerating. An awful lot of the PC's evolution has involved graphics adapters, and the next chapter and Chapter 42 are about the first successful step away from the 256-color SuperVGAs of the late 1980s, toward the Holy Grail of 24 bits per pixel.

In computer graphics, two display characteristics are paramount: color and resolution. In 1991, SuperVGAs had taken resolution up to 1024×768, but were still stuck at the 256-color resolution that IBM had designed into the original VGA. The rule for graphics is simple—the more colors the better—and the PC world was primed for a color breakthrough. The only questions were how and when it would happen.

The first shot was fired by Edsun Laboratories, in the form of the Edsun Continuous Edge Graphics (CEG) digital to analog converter (DAC, the chip that converts pixel values from the VGA into analog signals for input to the monitor). The CEG DAC was an ingenious bridge between SuperVGAs and higher color that required no modification to VGA chips and no additional memory, yet achieved (with considerable software effort) stunning results. If you're curious about ancient history, have a look at my "Graphics Programming" column in *Dr. Dobb's Journal* for April and May 1991, where I discussed the CEG DAC in detail. If you do look up these columns, you'll also find that my record as a prophet has its blemishes—I thought the CEG DAC was going to take off, but, alas, the CEG DAC's programming model was more than difficult and (far worse) delivery schedules slipped. Only a few Edsun-equippped VGAs ever made it into the field, and the unique Edsun technology became a historical oddity, its potential unrealized.

Time and technology marched on, and later in 1991 the spotlight settled on another new device, the Sierra Semiconductor Hicolor DAC, whose mission was much the same as the CEG DAC, but with a more promising implementation. The Hicolor DAC was and is easy to program, a simple extension of the 256-color programming model; it's also affordable and a simple drop-in replacement for the standard VGA DAC. Best of all, it shipped in good order and worked as advertised right off the bat, and, consequently, met with good success.

A fairly typical high-end SuperVGA card in late 1991 and early 1992 was built around the Tseng Labs ET4000 VGA chip, 1MB of RAM, and the Hicolor DAC, a combination that enabled support of the then-unprecedented 800×600 and 640×480, 32,768-color modes. Other technologies have appeared to crowd the Hicolor DAC off the cutting edge, but the chip is still used in low-end VGAs, and almost all new video adapters these days support high-color modes much like the ones that the Hicolor DAC pioneered. Besides, there are still lots of Hicolor DACs around in the installed base, and the Hicolor programming model is still used by most new adapters, so the Hicolor DAC is well worth understanding as one of the keys to the generation of graphics hardware that in the early 1990s swept far beyond the limits set by the original VGA.

## Unreal Color

To those of us who remembered buying IBM EGAs for $1000, it was kind of unreal to see an 800×600 32K-color VGA for less than $200 back in 1991. Understand, now, that I'm not talking about clever bitmap encoding or color look-up tables or other tricky ways of boosting color here. This is the real, 15-bpp (bits per pixel), almost true-color McCoy, beautifully suited to imaging, antialiasing, and virtually any sort of high-color graphics you might imagine. The Hicolor DAC supports normal bitmaps that are just like 256-color bitmaps, except that each pixel is composed of 15 bits spread across 2 bytes. If you know how to program 800×600 256-color mode, you should have no trouble at all programming the 800×600 32K-color mode; for the most part, just double the horizontal byte counts. (Lower-resolution 32K-color modes, such as 640×480, are also available.) The 32K-color banking schemes are the same as in 256-color modes, except that there are half as many pixels in each bank. Even the complexities of the DAC's programmable palette go away in 32K-color mode, because there is no programmable palette.

And therein lies the strength of the Hicolor DAC: It's easy to program. Theoretically, the Edsun CEG DAC could produce more precise images, with higher color content, using less display memory than the Hicolor DAC, because with the CEG DAC color resolutions of 24-bpp and even higher were possible. Practically speaking, however, it was hard to write software—especially real-time software—that took full advantage of the Edsun CEG DAC's capabilities. On the other hand, it's very easy to extend existing 256-color SuperVGA code to support the Hicolor DAC, and although 32K colors (15-bpp) is not the same as true color (24-bpp), it's close enough for most purposes, and *astonishingly* better than 256 colors. Digitized and rendered images look terrific on the Hicolor DAC, just as they did on the Edsun CEG DAC—and it's a lot easier and much faster to generate such images for the Hicolor DAC.

## The Gamma Correction Disadvantage

The Hicolor DAC has three disadvantages. First, it requires twice as much memory at a given resolution as does an equivalent 256-color mode. This is no longer a significant problem; memory is cheap, with 1MB essentially standard on SuperVGAs and 2MB becoming common. (The 1024×768 32K color mode that was impossible on a 1 MB SuperVGA is easy on a board with 2MB.) Second, graphics operations can take considerably longer, simply because there are twice as many bytes of display memory to be dealt with; however, the latest generation of SuperVGAs provides for such fast memory access that 32K-color software runs faster than 256-color software did on the first generation of SuperVGAs. Finally, the Hicolor DAC neither performs gamma correction in hardware nor provides a built-in look-up table to allow programmable gamma correction.

To refresh your memory, gamma correction is the process of compensating for the nonlinear response of pixel brightness to input voltage. A pixel with a green value of 60 is much more than twice as bright as a pixel of value 30. The Hicolor DAC's lack of built-in gamma correction puts the burden on software to perform the correction so that antialiasing will work properly, and so that images such as digitized photographs will display with the proper brightness. Software gamma correction is possible, but it's a time-consuming nuisance; it also decreases the effective color resolution of the Hicolor DAC for bright colors because the bright colors supported by the Hicolor DAC are spaced relatively farther apart than the dim colors.

The lack of gamma correction is, however, a manageable annoyance. On balance, the Hicolor DAC is true to its heritage; a logical, inexpensive, and painless extension of SuperVGA. The obvious next steps are 1024×768 in 32K colors (now common enough, but very exotic in 1991), and 800×600 with 24 bpp; heck, 4MB of display memory (eight 4-Megabit RAMS) would be enough for 1024×768 24-bpp with room to spare, and, as I write this, that's right around the corner. In short, the Hicolor DAC is squarely in the mainstream of VGA evolution. (Note that although most of the first generation of Hicolor boards were built around the Tseng ET4000, which quietly and for good reason became the preeminent SuperVGA chip of 1991 and 1992, the Hicolor DAC works with other VGA chips and can be found in SuperVGAs of all sorts.)

## Polygon Antialiasing

To my mind, the best thing about the Hicolor DAC is that it makes fast, general antialiasing possible. You see, what I've been working toward in this book is real-time 3-D perspective drawing on a standard PC, without the assistance of any expensive hardware. The object model I'll be using is polygon-based; hence the fast polygon fill code I presented in Chapters 38 and 39. With Mode X (320×240, 256 colors, undocumented by IBM but covered in this book in Chapters 47–49), we now have a fast, square-pixel, page-flipped, 256-color mode, the best that standard VGA has to offer. In this mode, it's possible to do not only real-time, polygon-based perspective drawing and animation, but also relatively sophisticated effects such as lighting sources, smooth shading, and hidden surface removal. That's everything we need for real-time 3-D—but things could still be better.

Pixels are so large in Mode X that polygons have *very* visibly jagged edges. These jaggies are the result of *aliasing*; that is, distortion of the true image that results from undersampling at the low pixel rate of the screen. Jaggies are a serious problem; the whole point of real-time 3-D is to create the illusion of reality, but jaggies quickly destroy that illusion, particularly when they're crawling along the edges of moving objects. More frequent sampling (higher resolution) helps, but not as much as you'd think. What's really needed is the ability to blend colors arbitrarily within a single pixel, the better to reflect the nature of the true image in the neighborhood of that pixel—that is, *antialiasing*. The pixels are still as large as ever, but with the colors blended properly, the eye processes the screen as a continuous image, rather than as a collection of discrete pixels, and perceives the image at much higher resolution than the display actually supports.

tip

*There are many ways to antialias, some of them fast enough for real-time processing, and they can work wonders in improving image appearance—but they all require a high degree of freedom in choosing colors. For many sorts of graphics, 256 simultaneous colors is fine, but it's not enough for generally useful antialiasing (although we will shortly see an interesting sort of special-case antialiasing with 256 colors). Therefore, the one element lacking in standard SuperVGA for affordable real-time 3-D has been good antialiasing.*

Sierra filled that gap. The Hicolor DAC provides plenty of colors (although I sure do wish the software didn't have to do gamma correction!) and makes them available in a way that allows for efficient programming. In the next chapter, I'll present antialiasing code for the Hicolor DAC.

## 256-Color Antialiasing

In the next chapter, I'll explain how the Hicolor DAC actually works—how to detect it, how to initialize it, the pixel format, banking, and so on—and then I'll demonstrate Hicolor antialiasing. For now, I'm going to demonstrate antialiasing on a standard VGA, partly to introduce the uncomplicated but effective antialiasing technique that I'll use in the next chapter, partly so you can see the improvement that even quick and dirty antialiasing produces, and partly to show the sorts of interesting things that can be done with the palette in 256-color mode.

I'm going to draw a cube in perspective. For reference, Listing F.1 draws the cube in mode 13H (320×200, 256 colors) using the standard polygon fill routine that I developed in Chapters 38 and 39. No, the perspective calculations aren't performed in Listing F.1; I just got the polygon vertices out of some 3-D software that I'm developing and hardwired them into Listing F.1. Never fear, though; we'll get to true 3-D soon enough.

Some listings from previous chapters are required to build the executable files for this chapter (including the edge-scanning code in Listing 39.2 and POLYGON.H), but to lessen the confusion all of these are present in the Chapter F subdirectory on the listings diskette.

Listing F.1 draws a serviceable cube, but the edges of the cube are very jagged. Imagine the cube spinning, and the jaggies rippling along its edges, and you'll see the full dimensions of the problem.

### LISTING F.1 LF-1.C

```
/* Demonstrates non-antialiased drawing in 256 color mode. Tested with
   Borland C++ in C mode in the small model. */

#include <conio.h>
#include <dos.h>
#include "polygon.h"

/* Draws the polygon described by the point list PointList in color
   Color with all vertices offset by (X,Y) */
#define DRAW_POLYGON(PointList,Color,X,Y)                  \
   Polygon.Length = sizeof(PointList)/sizeof(struct Point); \
   Polygon.PointPtr = PointList;                            \
   FillConvexPolygon(&Polygon, Color, X, Y);

void main(void);
extern int FillConvexPolygon(struct PointListHeader *, int, int, int);

/* Palette RGB settings to load the first four palette locations with
   black, pure blue, pure green, and pure red */
static char Palette[4*3] = {0, 0, 0,  0, 0, 63,  0, 63, 0,  63, 0, 0};

void main()
{
   struct PointListHeader Polygon;
   static struct Point Face0[] =
         {{198,138},{211,89},{169,44},{144,89}};
   static struct Point Face1[] =
         {{153,150},{198,138},{144,89},{105,113}};
   static struct Point Face2[] =
         {{169,44},{133,73},{105,113},{144,89}};
   union REGS regset;
   struct SREGS sregs;

   /* Set the display to VGA mode 13h, 320x200 256-color mode */
   regset.x.ax = 0x0013; int86(0x10, &regset, &regset);

   /* Set color 0 to black, color 1 to pure blue, color 2 to pure
      green, and color 3 to pure red */
   regset.x.ax = 0x1012;   /* load palette block BIOS function */
   regset.x.bx = 0;        /* start with palette register 0 */
   regset.x.cx = 4;        /* set four palette registers */
   regset.x.dx = (unsigned int) Palette;
   segread(&sregs);
   sregs.es = sregs.ds;       /* point ES:DX to Palette */
   int86x(0x10, &regset, &regset, &sregs);

   /* Draw the cube */
   DRAW_POLYGON(Face0, 3, 0, 0);
   DRAW_POLYGON(Face1, 2, 0, 0);
   DRAW_POLYGON(Face2, 1, 0, 0);
   getch();    /* wait for a keypress */

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

Listings F.2 and F.3 together draw the same cube, but with simple, unweighted antialiasing. The results are much better than Listing F.1; there's no question in my mind as to which cube I'd rather see in my graphics software.

### LISTING F.2 LF-2.C

```
/* Demonstrates unweighted antialiased drawing in 256 color mode.
   Tested with Borland C++ in C mode in the small model. */

#include <conio.h>
#include <dos.h>
#include <stdlib.h>
#include <string.h>
#include "polygon.h"

/* Draws the polygon described by the point list PointList in color
   Color, with all vertices offset by (X,Y), to ScanLineBuffer, at
   double horizontal and vertical resolution */
#define DRAW_POLYGON_DOUBLE_RES(PointList,Color,x,y)        \
   Polygon.Length = sizeof(PointList)/sizeof(struct Point); \
   Polygon.PointPtr = PointTemp;                            \
   /* Double all vertical & horizontal coordinates */       \
   for (k=0; k<sizeof(PointList)/sizeof(struct Point); k++) { \
      PointTemp[k].X = PointList[k].X * 2;                  \
      PointTemp[k].Y = PointList[k].Y * 2;                  \
   }                                                        \
   FillCnvxPolyDrvr(&Polygon, Color, x, y, DrawBandedList);
```

```
#define SCREEN_WIDTH 320
#define SCREEN_HEIGHT 200
#define SCREEN_SEGMENT 0xA000
#define SCAN_BAND_WIDTH (SCREEN_WIDTH*2)  /* # of double-res pixels
                                             across scan band */
#define BUFFER_SIZE  (SCREEN_WIDTH*2*2)   /* enough space for one scan
                                           line scanned out at double
                                           resolution horz and vert */
void main(void);
void DrawPixel(int, int, char);
int ColorComponent(int, int);
extern int FillCnvxPolyDrvr(struct PointListHeader *, int, int, int,
   void (*)());
extern void DrawBandedList(struct HLineList *, int);

/* Pointer to buffer in which double-res scanned data will reside */
unsigned char *ScanLineBuffer;
int ScanBandStart, ScanBandEnd;  /* top & bottom of each double-res
                                  band we'll draw to ScanLineBuffer */
int ScanBandWidth = SCAN_BAND_WIDTH;  /* # pixels across scan band */
static char Palette[256*3];

void main()
{
   int i, j, k;
   struct PointListHeader Polygon;
   struct Point PointTemp[4];
   static struct Point Face0[] =
         {{198,138},{211,89},{169,44},{144,89}};
   static struct Point Face1[] =
         {{153,150},{198,138},{144,89},{105,113}};
   static struct Point Face2[] =
         {{169,44},{133,73},{105,113},{144,89}};
   unsigned char Megapixel;
   union REGS regset;
   struct SREGS sregs;

   if ((ScanLineBuffer = malloc(BUFFER_SIZE)) == NULL) {
      printf("Couldn't get memory\n");
      exit(0);
   }

   /* Set the display to VGA mode 13h, 320x200 256-color mode */
   regset.x.ax = 0x0013; int86(0x10, &regset, &regset);

   /* Stack the palette for the desired megapixel effect, with each
      2-bit field representing 1 of 4 double-res pixels in one of four
      colors */
   for (i=0; i<256; i++) {
      Palette[i*3] = ColorComponent(i, 3);   /* red component */
      Palette[i*3+1] = ColorComponent(i, 2); /* green component */
      Palette[i*3+2] = ColorComponent(i, 1); /* blue component */
   }
   regset.x.ax = 0x1012;   /* load palette block BIOS function */
   regset.x.bx = 0;        /* start with palette register 0 */
   regset.x.cx = 256;      /* set all 256 palette registers */
   regset.x.dx = (unsigned int) Palette;
   segread(&sregs);
   sregs.es = sregs.ds;       /* point ES:DX to Palette */
   int86x(0x10, &regset, &regset, &sregs);
   /* Scan out the polygons at double resolution one screen scan line
      at a time (two double-res scan lines at a time) */
   for (i=0; i<SCREEN_HEIGHT; i++) {
      /* Set the band dimensions for this pass */
      ScanBandEnd = (ScanBandStart = i*2) + 1;
      /* Clear the drawing buffer */
      memset(ScanLineBuffer, 0, BUFFER_SIZE);
      /* Draw the current band of the cube to the scan line buffer */
      DRAW_POLYGON_DOUBLE_RES(Face0, 3, 0, 0);
      DRAW_POLYGON_DOUBLE_RES(Face1, 2, 0, 0);
      DRAW_POLYGON_DOUBLE_RES(Face2, 1, 0, 0);

      /* Coalesce the double-res pixels into normal screen pixels
         and draw them */
      for (j=0; j<SCREEN_WIDTH; j++) {
         Megapixel = (ScanLineBuffer[j*2] << 6) +
                     (ScanLineBuffer[j*2+1] << 4) +
                     (ScanLineBuffer[j*2+SCAN_BAND_WIDTH] << 2) +
                     (ScanLineBuffer[j*2+SCAN_BAND_WIDTH+1]);
         DrawPixel(j, i, Megapixel);
      }
   }

   getch();    /* wait for a keypress */

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

/* Draws a pixel of color Color at (X,Y) in mode 13h */
void DrawPixel(int X, int Y, char Color)
{
   char far *ScreenPtr;

   ScreenPtr = (char far *)MK_FP(SCREEN_SEGMENT, Y*SCREEN_WIDTH+X);
   *ScreenPtr = Color;
}

/* Returns the gamma-corrected value representing the number of
   double-res pixels containing the specified color component in a
   megapixel with the specified value */
int ColorComponent(int Value, int Component)
{
   /* Palette settings for 0%, 25%, 50%, 75%, and 100% brightness,
      assuming a gamma value of 2.3 */
   static int GammaTable[] = {0, 34, 47, 56, 63};
   int i;

   /* Add up the number of double-res pixels of the specified color
      in a megapixel of this value */
   i = (((Value & 0x03) == Component) ? 1 : 0) +
       ((((Value >> 2) & 0x03) == Component) ? 1 : 0) +
       ((((Value >> 4) & 0x03) == Component) ? 1 : 0) +
       ((((Value >> 6) & 0x03) == Component) ? 1 : 0);
   /* Look up brightness of the specified color component in a
      megapixel of this value */
   return GammaTable[i];
}
```

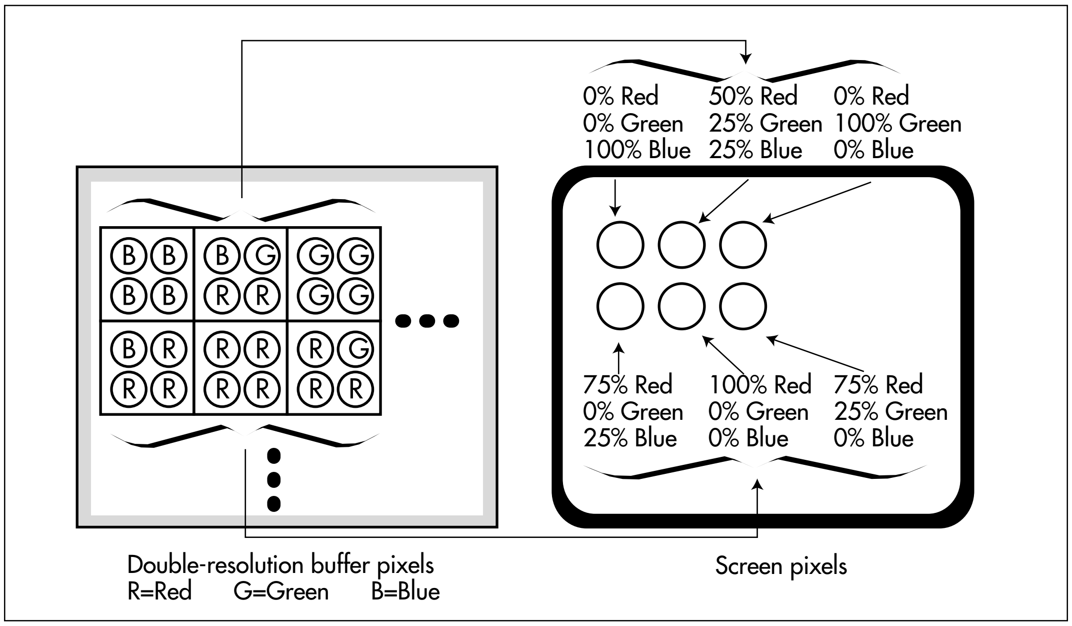
**LISTING F.3 LF-3.C**

```
/* Draws pixels from the list of horizontal lines passed in; drawing
   takes place only for scan lines between ScanBandStart and
   ScanBandEnd, inclusive; drawing goes to ScanLineBuffer, with
   the scan line at ScanBandStart mapping to the first scan line in
   ScanLineBuffer. Intended for use in unweighted antialiasing,
   whereby a polygon is scanned out into a buffer at a multiple of the
   screen's resolution, and then the scanned-out info in the buffer is
   grouped into megapixels that are mapped to the closest
   approximation the screen supports and drawn. Tested with Borland
   C++ in C mode in the small model */

#include <string.h>
#include <dos.h>
#include "polygon.h"

extern unsigned char *ScanLineBuffer;  /* drawing goes here */
extern int ScanBandStart, ScanBandEnd; /* limits of band to draw */
extern int ScanBandWidth;  /* # of pixels across scan band */

void DrawBandedList(struct HLineList * HLineListPtr, int Color)
{
   struct HLine *HLinePtr;
   int Length, Width, YStart = HLineListPtr->YStart;
   unsigned char *BufferPtr;

   /* Done if fully off the bottom or top of the band */
   if (YStart > ScanBandEnd) return;
   Length = HLineListPtr->Length;
   if ((YStart + Length) <= ScanBandStart) return;

   /* Point to the XStart/XEnd descriptor for the first (top)
      horizontal line */
   HLinePtr = HLineListPtr->HLinePtr;

   /* Confine drawing to the specified band */
   if (YStart < ScanBandStart) {
      /* Skip ahead to the start of the band */
      Length -= ScanBandStart - YStart;
      HLinePtr += ScanBandStart - YStart;
      YStart = ScanBandStart;
   }
   if (Length > (ScanBandEnd - YStart + 1))
      Length = ScanBandEnd - YStart + 1;

   /* Point to the start of the first scan line on which to draw */
   BufferPtr = ScanLineBuffer + (YStart - ScanBandStart) *
         ScanBandWidth;

   /* Draw each horizontal line within the band in turn, starting with
      the top one and advancing one line each time */
   while (Length-- > 0) {
      /* Draw the whole horizontal line if it has a positive width */
      if ((Width = HLinePtr->XEnd - HLinePtr->XStart + 1) > 0)
         memset(BufferPtr + HLinePtr->XStart, Color, Width);
      HLinePtr++;                /* point to next scan line X info */
      BufferPtr += ScanBandWidth; /* point to next scan line start */
   }
}
```

The antialiasing technique used in Listing F.2 is straightforward. Each polygon is scanned out in the usual way, but at twice the screen's resolution both horizontally and vertically (which I'll call "double-resolution," although it produces four times as many pixels), with the double-resolution pixels drawn to a memory buffer, rather than directly to the screen.

Then, after all the polygons have been drawn to the memory buffer, a second pass is performed. This pass looks at the colors stored in each set of four double-resolution pixels, and draws to the screen a single pixel that reflects the colors and intensities of the four double-resolution pixels that make it up, as shown in Figure F.1. In other words, Listing F.2 temporarily draws the polygons at double resolution, then uses the extra information from the double-resolution bitmap to generate an image with an effective resolution considerably higher than the screen's actual 320×200 capabilities.

Two interesting tricks are employed in Listing F.2. First, it would be best from the standpoint of speed if the entire screen could be drawn to the double-resolution intermediate buffer in a single pass. Unfortunately, a buffer capable of holding one full 640×400 screen would be 256K bytes in size—too much memory for most programs to spare. Consequently, Listing F.2 instead scans out the image just two double-resolution scan lines (corresponding to one screen scan line) at a time. That is, the entire image is scanned once for every two double-resolution scan lines, and



*Mapping double-resolution pixels to single screen pixels.*
**Figure F.1**

all information not concerning the two lines of current interest is thrown away. This banding is implemented in Listing F.3, which accepts a full list of scan lines to draw, but actually draws only those lines within the current scan line band. Listing F.3 also draws to the intermediate buffer, rather than to the screen.

The polygon-scanning code from Chapter 38 was hard-wired to call the function **DrawHorizontalLineList**, which drew to the display; this is the polygon-drawing code called by Listing F.1. That was fine so long as there was only one possible drawing target, but now we have two possible targets—the display (for nonantialiased drawing), and the intermediate buffer (for antialiased drawing). It's desirable to be able to mix the two, even within a single screen, because antialiased drawing looks better but nonantialiased is faster. Consequently, I have modified Listing 38.1 from Chapter 38—the function **FillConvexPolygon**—to create **FillCnvxPolyDrvr**, which is the same as **FillConvexPolygon**, except that it accepts as a parameter the name of the function to be used to draw the scanned-out polygon. The modified file is shown in its entirety in Listing F.4.

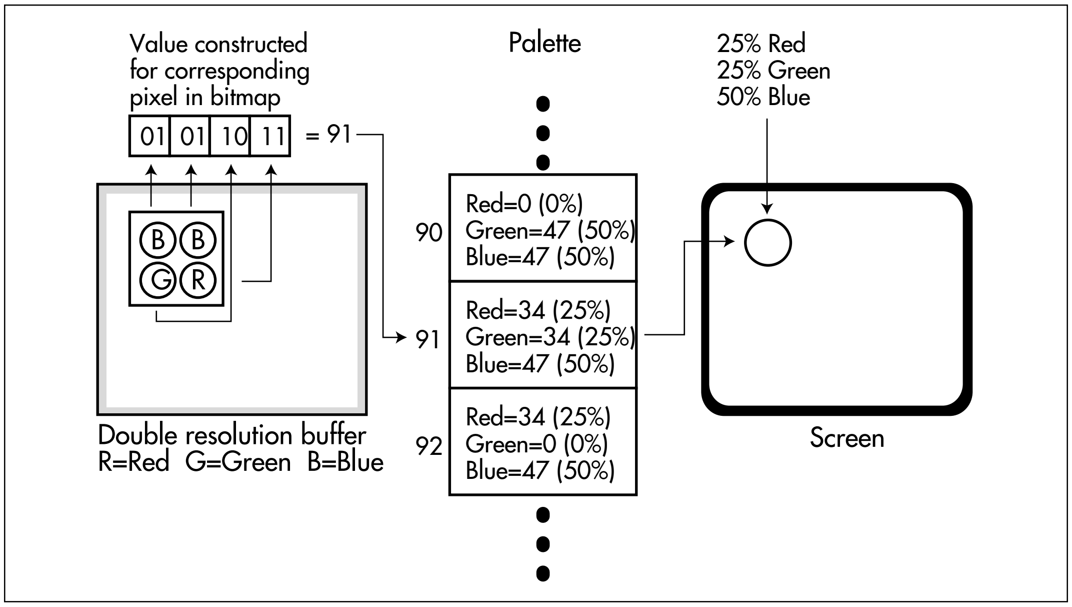
**LISTING F.4 FILCNVX.C**

```
/* Color-fills a convex polygon, using the passed-in driver to perform
   all drawing. All vertices are offset by (XOffset, YOffset).
   "Convex" means that every horizontal line drawn through the polygon
   at any point would cross exactly two active edges (neither
   horizontal lines nor zero-length edges count as active edges; both
   are acceptable anywhere in the polygon), and that the right & left
   edges never cross. (It's OK for them to touch, though, so long as
   the right edge never crosses over to the left of the left edge.)
   Nonconvex polygons won't be drawn properly. Returns 1 for success,
   0 if memory allocation failed. */

#include <stdio.h>
#include <math.h>
#ifdef __TURBOC__
#include <alloc.h>
#else    /* MSC */
#include <malloc.h>
#endif
#include "polygon.h"

/* Advances the index by one vertex forward through the vertex list,
   wrapping at the end of the list */
#define INDEX_FORWARD(Index) \
   Index = (Index + 1) % VertexList->Length;

/* Advances the index by one vertex backward through the vertex list,
   wrapping at the start of the list */
#define INDEX_BACKWARD(Index) \
   Index = (Index - 1 + VertexList->Length) % VertexList->Length;

/* Advances the index by one vertex either forward or backward through
   the vertex list, wrapping at either end of the list */
#define INDEX_MOVE(Index,Direction)                              \
   if (Direction > 0)                                            \
      Index = (Index + 1) % VertexList->Length;                  \
   else                                                          \
      Index = (Index - 1 + VertexList->Length) % VertexList->Length;
```

```
void ScanEdge(int, int, int, int, int, int, struct HLine **);

int FillCnvxPolyDrvr(struct PointListHeader * VertexList, int Color,
      int XOffset, int YOffset, void (*DrawListFunc)())
{
   int i, MinIndexL, MaxIndex, MinIndexR, SkipFirst, Temp;
   int MinPoint_Y, MaxPoint_Y, TopIsFlat, LeftEdgeDir;
   int NextIndex, CurrentIndex, PreviousIndex;
   int DeltaXN, DeltaYN, DeltaXP, DeltaYP;
   struct HLineList WorkingHLineList;
   struct HLine *EdgePointPtr;
   struct Point *VertexPtr;

   /* Point to the vertex list */
   VertexPtr = VertexList->PointPtr;

   /* Scan the list to find the top and bottom of the polygon */
   if (VertexList->Length == 0)
      return(1);  /* reject null polygons */
   MaxPoint_Y = MinPoint_Y = VertexPtr[MinIndexL = MaxIndex = 0].Y;
   for (i = 1; i < VertexList->Length; i++) {
      if (VertexPtr[i].Y < MinPoint_Y)
         MinPoint_Y = VertexPtr[MinIndexL = i].Y; /* new top */
      else if (VertexPtr[i].Y > MaxPoint_Y)
         MaxPoint_Y = VertexPtr[MaxIndex = i].Y; /* new bottom */
   }
   if (MinPoint_Y == MaxPoint_Y)
      return(1);  /* polygon is 0-height; avoid infinite loop below */

   /* Scan in ascending order to find the last top-edge point */
   MinIndexR = MinIndexL;
   while (VertexPtr[MinIndexR].Y == MinPoint_Y)
      INDEX_FORWARD(MinIndexR);
   INDEX_BACKWARD(MinIndexR); /* back up to last top-edge point */

   /* Now scan in descending order to find the first top-edge point. */
   while (VertexPtr[MinIndexL].Y == MinPoint_Y)
      INDEX_BACKWARD(MinIndexL);
   INDEX_FORWARD(MinIndexL); /* back up to first top-edge point */

   /* Figure out which direction through the vertex list from the top
      vertex is the left edge and which is the right */
   LeftEdgeDir = -1; /* assume left edge runs down thru vertex list */
   if ((TopIsFlat = (VertexPtr[MinIndexL].X !=
         VertexPtr[MinIndexR].X) ? 1 : 0) == 1) {
      /* If the top is flat, just see which of the ends is leftmost */
      if (VertexPtr[MinIndexL].X > VertexPtr[MinIndexR].X) {
         LeftEdgeDir = 1;  /* left edge runs up through vertex list */
         Temp = MinIndexL;       /* swap the indices so MinIndexL   */
         MinIndexL = MinIndexR;  /* points to the start of the left */
         MinIndexR = Temp;       /* edge, similarly for MinIndexR   */
      }
   } else {
      /* Point to the downward end of the first line of each of the
         two edges down from the top */
      NextIndex = MinIndexR;
      INDEX_FORWARD(NextIndex);
      PreviousIndex = MinIndexL;
      INDEX_BACKWARD(PreviousIndex);
   /* Calculate X and Y lengths from the top vertex to the end of
      the first line down each edge; use those to compare slopes
      and see which line is leftmost */
   DeltaXN = VertexPtr[NextIndex].X - VertexPtr[MinIndexL].X;
   DeltaYN = VertexPtr[NextIndex].Y - VertexPtr[MinIndexL].Y;
   DeltaXP = VertexPtr[PreviousIndex].X - VertexPtr[MinIndexL].X;
   DeltaYP = VertexPtr[PreviousIndex].Y - VertexPtr[MinIndexL].Y;
   if (((long)DeltaXN * DeltaYP - (long)DeltaYN * DeltaXP) < 0L) {
      LeftEdgeDir = 1;  /* left edge runs up through vertex list */
      Temp = MinIndexL;       /* swap the indices so MinIndexL   */
      MinIndexL = MinIndexR;  /* points to the start of the left */
      MinIndexR = Temp;       /* edge, similarly for MinIndexR   */
   }
}

/* Set the # of scan lines in the polygon, skipping the bottom edge
   and also skipping the top vertex if the top isn't flat because
   in that case the top vertex has a right edge component, and set
   the top scan line to draw, which is likewise the second line of
   the polygon unless the top is flat. */
if ((WorkingHLineList.Length =
      MaxPoint_Y - MinPoint_Y - 1 + TopIsFlat) <= 0)
   return(1);  /* there's nothing to draw, so we're done */
WorkingHLineList.YStart = YOffset + MinPoint_Y + 1 - TopIsFlat;

/* Get memory in which to store the line list we generate */
if ((WorkingHLineList.HLinePtr =
      (struct HLine *) (malloc(sizeof(struct HLine) *
      WorkingHLineList.Length))) == NULL)
   return(0);  /* couldn't get memory for the line list */

/* Scan the left edge and store the boundary points in the list */
/* Initial pointer for storing scan converted left-edge coords */
EdgePointPtr = WorkingHLineList.HLinePtr;
/* Start from the top of the left edge */
PreviousIndex = CurrentIndex = MinIndexL;
/* Skip the first point of the first line unless the top is flat;
   if the top isn't flat, the top vertex is exactly on a right
   edge and isn't drawn */
SkipFirst = TopIsFlat ? 0 : 1;
/* Scan convert each line in the left edge from top to bottom */
do {
   INDEX_MOVE(CurrentIndex,LeftEdgeDir);
   ScanEdge(VertexPtr[PreviousIndex].X + XOffset,
         VertexPtr[PreviousIndex].Y,
         VertexPtr[CurrentIndex].X + XOffset,
         VertexPtr[CurrentIndex].Y, 1, SkipFirst, &EdgePointPtr);
   PreviousIndex = CurrentIndex;
   SkipFirst = 0; /* scan convert the first point from now on */
} while (CurrentIndex != MaxIndex);

/* Scan the right edge and store the boundary points in the list */
EdgePointPtr = WorkingHLineList.HLinePtr;
PreviousIndex = CurrentIndex = MinIndexR;
SkipFirst = TopIsFlat ? 0 : 1;
/* Scan convert the right edge, top to bottom. X coordinates are
   adjusted 1 to the left, effectively causing scan conversion of
   the nearest points to the left of but not exactly on the edge */
do {
   INDEX_MOVE(CurrentIndex,-LeftEdgeDir);
```

```
      ScanEdge(VertexPtr[PreviousIndex].X + XOffset - 1,
            VertexPtr[PreviousIndex].Y,
            VertexPtr[CurrentIndex].X + XOffset - 1,
            VertexPtr[CurrentIndex].Y, 0, SkipFirst, &EdgePointPtr);
      PreviousIndex = CurrentIndex;
      SkipFirst = 0; /* scan convert the first point from now on */
   } while (CurrentIndex != MaxIndex);

   /* Draw the line list representing the scan converted polygon */
   (*DrawListFunc)(&WorkingHLineList, Color);

   /* Release the line list's memory and we're successfully done */
   free(WorkingHLineList.HLinePtr);
   return(1);
}
```

The second interesting trick in Listing F.2 is the way in which the palette is stacked to allow unweighted antialiasing. Listing F.2 arranges the palette so that rather than 256 independent colors, we'll work with four-way combinations within each pixel of three independent colors (red, green, and blue), with each pixel accurately reflecting the intensities of each of the four color components (double-resolution pixels) that it contains. This allows fast and easy mapping from four double-resolution pixels to the single screen pixel to which they correspond. Figure F.2 illustrates the mapping of subpixels (double-resolution pixels) through the palette to screen pixels. This palette organization converts mode 13H from a 256-color mode to a three-color antialiasing mode.



*From the double-resolution buffer to the screen.*
**Figure F.2**

It's worth noting that many palette registers are set to identical values by Listing F.2, because whereas the values of subpixels matter, arrangements of these values do not. For example, the pixel values 0x01, 0x04, 0x10, and 0x40 all map to 25 percent blue. By using a table look-up to map sets of four double-resolution pixels to screen pixel values, more than half the palette could be freed up for drawing with other colors.

## Unweighted Antialiasing: How Good?

Is the antialiasing used in Listing F.2 the finest possible antialiasing technique? It is not. It is an *unweighted* antialiasing technique, meaning that no accounting is made for how close to the center of a pixel a polygon edge might be. The edges are also biased a half-pixel or so in some cases, so registration with the underlying image isn't perfect. Nonetheless, the technique used in Listing F.2 produces attractive results, which is what really matters; keep in mind that *all* screen displays are approximations, and unweighted antialiasing is certainly good enough for PC animation applications. Unweighted antialiasing can also support good performance, although this is not the case in Listings F.2 and F.3, where I have opted for clarity rather than performance. Increasing the number of lines drawn on each pass, or reducing the area processed to the smallest possible bounding rectangle, would help improve performance, as, of course, would the use of assembly language.

For further information on antialiasing, you might check out the standard reference: *Computer Graphics: Principles and Practice,* by Foley and Van Dam. Michael Covington's "Smooth Views," in the May, 1990 *Byte*, provides a short but meaty discussion of unweighted line antialiasing.

As relatively good as it looks, Listing F.2 is still watered-down antialiasing, even of the unweighted variety. For all our clever palette stacking, we have only five levels of each primary color available; that's a far cry from the 32 levels of the Hicolor DAC, or the 256 levels of true color. The limitations of 256-color modes, even with the palette, are showing through.

In the next chapter, we'll take a look at how much better 15-bpp antialiasing can be.