Chapter G

Lines, Italian Style

Chapter G

Using the Sierra Hicolor DAC to Make Errant Lines Look Good

There's an Italian saying, the gist of which is, "It need not be true, so long as it's well said." This strikes close to the essential truth of antialiasing: The image need not be accurate, so long as it *looks* like it is. You don't go to the trouble of antialiasing in order to get a mathematically precise representation of an image; you do it so the amazing human eye/brain integrating/pattern matching system will see what you want it to see.

This is a particularly relevant thought at the moment, for we're smack in the middle of discussing the Sierra Hicolor DAC, which makes classic, high-quality antialiasing, of the sort that Targa boards have offered for years, available at mass-market prices. To recap, the Hicolor DAC extends SuperVGA to provide selection among enough colors for serious rendering and antialiasing; 32,768 simultaneous colors, to be exact. The Hicolor DAC falls short of the 24-bpp true color standard, but you aren't likely to find a 24-bpp true color adapter priced anywhere close to what a Hicolor-based board is (although this is changing).

In the previous chapter, we looked at simple, unweighted antialiasing in the context of the VGA's standard 256-color mode, performing antialiasing between exactly three colors—red, green, and blue—with five semi-independent levels of each of the three primary colors available. In this chapter, we'll start off by discussing the basic Hicolor programming model, then we'll do the same sort of antialiasing as we did in the previous chapter—but this time with 32 fully independent levels of each primary color and resolutions up to 800×600, which makes quite a difference indeed.

A Brief Primer on the Sierra Hicolor DAC

The operation of the Hicolor DAC in 32K-color mode is remarkably simple. First, the VGA must be set to a 256-color mode with twice the desired horizontal resolution; for example, a 1600×600 256-color mode would be selected if 800×600 32K-color mode were desired. Then, the Hicolor DAC is set to high-color mode via the command register; in high-color mode, the Hicolor DAC takes each pair of 256-color pixels, joins them together into one 16-bit pixel, converts the red, green, and blue components (described shortly) directly to proportional analog values (no palette is involved), and sends them to the monitor.

There is a serious problem here, however: There is no standard way for an application to select a high-color mode. It's not enough to set up the Hicolor DAC; the VGA must also be set to the appropriate double-resolution 256-color mode, and the sequence for doing that—especially selecting high-speed clocks—varies from VGA to VGA. There is no VESA mode number for high-color modes; there is a VESA programming guideline for high-color modes, but it's certainly not as simple as a mode number. In any case, the VESA interface isn't always available.

Consequently, high-color mode selection is adapter-dependent. Fortunately, many of the Hicolor-based boards are built around the Tseng Labs ET4000 VGA chip, and Tseng provides a BIOS interface for high-color modes. (There's no guarantee that manufacturers using the ET4000 will follow the Tseng interface, but I suspect they will, as it's the closest thing to a standard at the moment.) Unfortunately, when I run the ET4000 BIOS function that reports whether a Hicolor DAC is present on my Toshiba portable without a Hicolor board installed, it hangs my system, so it's not a good idea to rely on the BIOS functions alone.

My solution, shown in Listing G.1, is to first check for a Hicolor DAC and an ET4000 at the hardware level; if both are present, I call the BIOS (which is presumably at least not hostile to the Tseng BIOS high-color extensions at this point) to check for the availability of Hicolor modes, and finally, if all has gone well, to set the desired high-color mode. This is probably overkill, but at least this way you get three kinds of chip ID code to mix and match as you wish.

LISTING G.1 LG-1.C

```
/* Looks for a Sierra Hicolor DAC; if one is present, puts the VGA into the
specified Hicolor (32K color) mode. Relies on the Tseng Labs ET4000 BIOS and
hardware; probably will not work on adapters built around other VGA chips.
Returns 1 for success, 0 for failure; failure can result from no Hicolor DAC,
too little display memory, or lack of an ET4000. Tested with Borland C++
in C mode in the small model. */
```

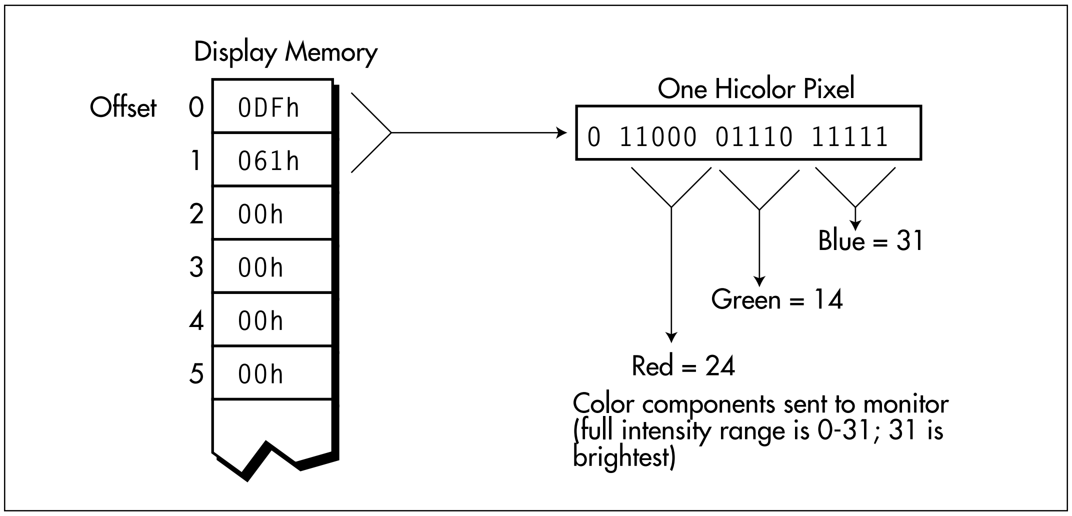
```
#include <dos.h>
#define DAC_MASK  0x3C6 /* DAC pixel mask reg address, also Sierra
                           command reg address when enabled */
#define DAC_WADDR 0x3C8  /* DAC write address reg address */

/* Mode selections: 0x2D=640x350; 0x2E=640x480; 0x2F=640x400; 0x30=800x600 */
int SetHCMode(int Mode) {
   int i, Temp1, Temp2, Temp3;
   union REGS regset;

   /* See if a Sierra SC1148X Hicolor DAC is present, by trying to
      program and then read back the DAC's command register. (Shouldn't be
      necessary when using the BIOS Get DAC Type function, but the BIOS function
      locks up some computers, so it's safer to check the hardware first). */
   inp(DAC_WADDR); /* reset the Sierra command reg enable sequence */
   for (i=0; i<4; i++) inp(DAC_MASK); /* enable command reg access */
   outp(DAC_MASK, 0x00); /* set command reg (if present) to 0x00, and
                            reset command reg enable sequence */
   outp(DAC_MASK, 0xFF); /* command reg access no longer enabled;
                            set pixel mask register to 0xFF */
   for (i=0; i<4; i++) inp(DAC_MASK); /* enable command reg access */
   /* If this is a Hicolor DAC, we should read back the 0 in the
      command reg; otherwise we get the 0xFF in the pixel mask reg */
   i = inp(DAC_MASK); inp(DAC_WADDR); /* reset enable sequence */
   if (i == 0xFF) return(0);

   /* Check for a Tseng Labs ET4000 by poking unique regs, (assumes
      VGA configured for color, w/CRTC addressing at 3D4/5) */
   outp(0x3BF, 3); outp(0x3D8, 0xA0);  /* unlock extended registers */
   /* Try toggling AC R16 bit 4 and seeing if it takes */
   inp(0x3DA); outp(0x3C0, 0x16 | 0x20);
   outp(0x3C0, ((Temp1 = inp(0x3C1)) | 0x10)); Temp2 = inp(0x3C1);
   outp(0x3C0, 0x16 | 0x20); outp(0x3C0, (inp(0x3C1) & ~0x10));
   Temp3 = inp(0x3C1); outp(0x3C0, 0x16 | 0x20);
   outp(0x3C0, Temp1);  /* restore original AC R16 setting */
   /* See if the bit toggled; if so, it's an ET3000 or ET4000 */
   if ((Temp3 & 0x10) || !(Temp2 & 0x10)) return(0);
   outp(0x3D4, 0x33); Temp1 = inp(0x3D5); /* get CRTC R33 setting */
   outp(0x3D5, 0x0A); Temp2 = inp(0x3D5); /* try writing to CRTC */
   outp(0x3D5, 0x05); Temp3 = inp(0x3D5); /*  R33 */
   outp(0x3D5, Temp1);  /* restore original CRTC R33 setting */
   /* If the register was writable, it's an ET4000 */
   if ((Temp3 != 0x05) || (Temp2 != 0x0A)) return(0);

   /* See if a Sierra SC1148X Hicolor DAC is present by querying the
      (presumably) ET4000-compatible BIOS. Not really necessary after
      the hardware check above, but generally more useful; in the
      future it will return information about other high-color DACs. */
   regset.x.ax = 0x10F1;   /* Get DAC Type BIOS function # */
   int86(0x10, &regset, &regset); /* ask BIOS for the DAC type */
   if (regset.x.ax != 0x0010) return(0); /* function not supported */
   switch (regset.h.bl) {
      case 0:  return(0);  /* normal DAC (non-Hicolor) */
      case 1:  break;      /* Sierra SC1148X 15-bpp Hicolor DAC */
      default: return(0);  /* other high-color DAC */
   }

   /* Set Hicolor mode */
   regset.x.ax = 0x10F0;   /* Set High-Color Mode BIOS function # */
   regset.h.bl = Mode;     /* desired resolution */
   int86(0x10, &regset, &regset); /* have BIOS enable Hicolor mode */
   return (regset.x.ax == 0x0010); /* 1 for success, 0 for failure */
}
```

Programming the Hicolor DAC

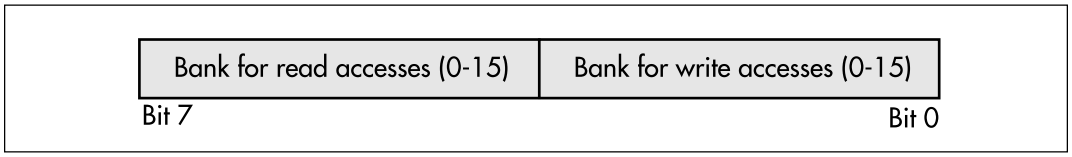
The pixel format of the Hicolor DAC is straightforward: Each pixel is stored in one word of display memory, with the lowest 5 bits forming the blue component, the next 5 bits forming the green component, the next 5 bits forming the red component, and bit 15 ignored, as shown in Figure G.1. Pixels start at even addresses. The bits within a word are organized Intel style, with the byte at the even address containing bits 7–0 (blue and part of green), and the byte at the odd address containing bits 15–8 (red and the rest of green).

Pixels proceed linearly for the length of the bitmap; the organization is the same as 256-color mode, except that each pixel takes up one word, rather than one byte. As in SuperVGA 256-color modes, the bitmap is too long to be addressed in the 64K video memory window, so banking must be used; again, the banking is just like 256-color banking, except that each bank contains only half as many pixels; 32,768, to be exact.

On the ET4000, the Segment Select register at 3CDH controls banking, as shown in Figure G.2. There are 16 banks, each spanning 64K of the total 1-Mb bitmap. Banks can be selected separately for read and write to facilitate scrolling and screen-to-screen copies, although we won't need that in this chapter. Simple enough, but there's a catch: broken rasters.



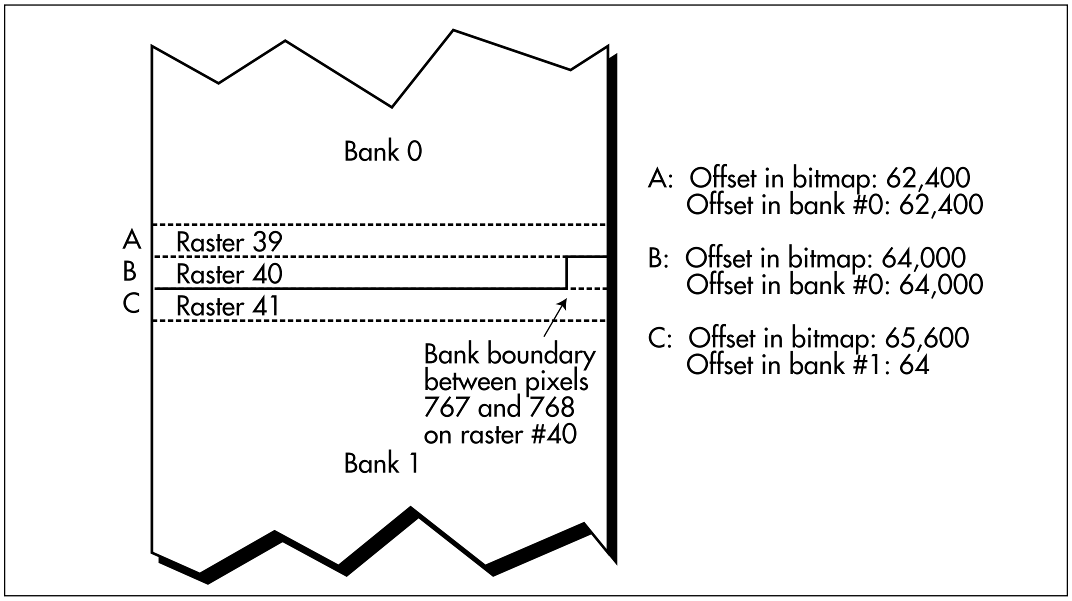
The Hicolor DAC 32K-color pixel format.
Figure G.1



The ET4000 Segment Select register (3CDH).
Figure G.2

Banks are 64K in length. If each Hicolor scan line is 1,600 bytes long, then 65,536/1,600=40 raster lines (lines 0–39) fit in bank 0—with 1,536 bytes of the next line (line 40), also in the first bank. The last 64 bytes of line 40 are in bank 1, as shown in Figure G.3, so the line is split by the bank boundary; hence the term "broken raster." Broken rasters crop up for other bank crossings as well, and make Hicolor programming somewhat slower (the extent of the slowdown depends heavily on the quality of the code) and considerably more complicated.

Broken rasters are not unique to Hicolor modes; 800×600 and 640×480 256-color modes also normally have broken rasters. However, there's a clever workaround for broken rasters in 256-color modes: Stretch the bitmap width to 1K pixels, via the Row Offset register, so that banks split between raster lines (although this works for 800×600 only if the VGA has 1 Mb or more of memory).


Bank 0
A
B
C
Raster 39
Raster 40
Raster 41
Bank boundary between pixels 767 and 768 on raster #40
Bank 1
A: Offset in bitmap: 62,400
Offset in bank #0: 62,400
B: Offset in bitmap: 64,000
Offset in bank #0: 64,000
C: Offset in bitmap: 65,600
Offset in bank #1: 64

A broken raster.
Figure G.3

Sad to say, stretching the bitmap width to 1K pixels doesn't work in Hicolor mode. There's not enough memory on a 1MB SuperVGA to do it at 800×600, but that's not the problem at 640×480. The problem is that a Row Offset register setting of 256 would be required to stretch the bitmap width to 1K pixels—and the Row Offset register only goes up to 255. I'm sure that when the VGA was being designed, 255 seemed like plenty, but then, 640K once seemed like pie in the sky.

The upshot is that Hicolor programming generally requires handling broken rasters. Generally—but not always. I learned of an exception from a a person I know only as "rfrederick" from M&T Online. He or she pointed out that setting the bitmap width—the offset from the start of one line to the start of the next—to 1,928 bytes in 640×480 Hicolor mode eliminates broken rasters (that is, displayed lines that span banks). A bitmap width of 1,928 is selected by setting the Row Offset register (CRTC register 13H) to 241, like so:

```
outpw(0x3d4, 0x13 | (241<<8));
```

Once the width is set, all you have to do is use 1,928 for the offset from one row to the next, rather than 1,280, and you're all set. Actually, the breaks are still there, but they can be ignored because they happen off to the right of the displayed portion of the bitmap. To get the Hicolor drawing code in this chapter to support 1,928-wide Hicolor bitmaps, just change **BitmapWidthInBytes** from 640*2 to 1,928. My online source credited this insight to the folks at Everex, to whom I am indebted.

Without the above workaround, or in 800×600 Hicolor mode, broken rasters are certainly a nuisance, but nonetheless a manageable one; next, we'll see polygon fill code that deals with broken rasters.

Non-Antialiased Hicolor Drawing

Listing G.2 draws a perspective cube in 640×480 32K color mode, with help from the initialization code in Listing G.1, the **DrawPixel**-based low-level polygon fill code in Listing G.3, and the POLYGON.H header file in Listing G.7. (FILCNVXD.C from the previous chapter and Listing 39.4 from Chapter 39 are also required; however, to make things less confusing for you, these will be in the Chapter G subdirectory on the listings diskette.) Not surprisingly, the cube drawn by Listing G.2 looks a lot like the non-antialiased cube we drew in the previous chapter, but isn't as jagged because the resolution is now much higher. Nonetheless, jaggies are still quite prominent, and they remain clearly visible at 800×600. (I've used 640×480 mode so that the code will work on fixed-frequency monitors, but Listing G.2 can be altered for 800×600 mode simply by changing the parameter passed to **SetHCMode** and the value of **BitmapWidthInBytes**.)

Listing G.2 doesn't run very fast when linked to Listing G.3; I suspect you'll like the low-level polygon fill code in Listing G.4 much better. This code handles broken rasters reasonably efficiently, by checking for them at the beginning of each scan line, then splitting up the fill and banking appropriately whenever a bank crossing is detected.

LISTING G.2 LG-2.C

```
/* Demonstrates non-antialiased drawing in 640x480 Hicolor (32K color) mode on
an ET4000-based SuperVGA with a Sierra Hicolor DAC installed. 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;                            \
   FillCnvxPolyDrvr(&Polygon, Color, x, y, DrawHCLineList);}

void main(void);
extern int SetHCMode(int);
extern int FillCnvxPolyDrvr(struct PointListHeader *, int, int, int,
   void (*)());
extern void DrawHCLineList(struct HLineList *, int);
int BitmapWidthInBytes = 640*2; /* # of bytes per raster line */

void main()
{
   struct PointListHeader Polygon;
   static struct Point Face0[] = {{396,276},{422,178},{338,88},{288,178}};
   static struct Point Face1[] = {{306,300},{396,276},{288,178},{210,226}};
   static struct Point Face2[] = {{338,88},{266,146},{210,226},{288,178}};
   union REGS regset;

   /* Attempt to enable 640x480 Hicolor mode */
   if (SetHCMode(0x2E) == 0)
      { printf("No Hicolor DAC detected\n"); exit(0); };

   /* Draw the cube */
   DRAW_POLYGON(Face0, 0x1F, 0, 0);       /* full-intensity blue */
   DRAW_POLYGON(Face1, 0x1F << 5, 0, 0);  /* full-intensity green */
   DRAW_POLYGON(Face2, 0x1F << 10, 0, 0); /* full-intensity red */
   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);
}
```

LISTING G.3 LG-3.C

```
/* Draws all pixels in the list of horizontal lines passed in, in Hicolor
(32K color) mode on an ET4000-based SuperVGA. Uses a slow pixel-by-pixel
approach. Tested with Borland C++ in C mode in the small model. */

#include <dos.h>
#include "polygon.h"
#define SCREEN_SEGMENT     0xA000
#define GC_SEGMENT_SELECT  0x3CD
```

```
void DrawPixel(int, int, int);
extern int BitmapWidthInBytes; /* # of pixels per line */

void DrawHCLineList(struct HLineList * HLineListPtr,
      int Color)
{
   struct HLine *HLinePtr;
   int Y, X;

   /* Point to XStart/XEnd descriptor for the first (top) horizontal line */
   HLinePtr = HLineListPtr->HLinePtr;
   /* Draw each horizontal line in turn, starting with the top one and
      advancing one line each time */
   for (Y = HLineListPtr->YStart; Y < (HLineListPtr->YStart +
         HLineListPtr->Length); Y++, HLinePtr++) {
      /* Draw each pixel in the current horizontal line in turn,
         starting with the leftmost one */
      for (X = HLinePtr->XStart; X <= HLinePtr->XEnd; X++)
         DrawPixel(X, Y, Color);
   }
}

/* Draws the pixel at (X, Y) in color Color in Hicolor mode on an
   ET4000-based SuperVGA */
void DrawPixel(int X, int Y, int Color) {
   unsigned int far *ScreenPtr, Bank;
   unsigned long BitmapAddress;

   /* Full bitmap address of pixel, as measured from address 0 to
      address 0xFFFFF. (X << 1) because pixels are 2 bytes in size */
   BitmapAddress = (unsigned long) Y * BitmapWidthInBytes + (X << 1);
   /* Map in the proper bank. Bank # is upper word of bitmap addr */
   Bank = *(((unsigned int *)&BitmapAddress) + 1);
   /* 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 = (unsigned int)Color;
}
```

LISTING G.4 LG-4.ASM

```
; Draws all pixels in the list of horizontal lines passed in, in
; Hicolor (32K color) mode on an ET4000-based SuperVGA. Uses REP STOSW
; to fill each line. Tested with TASM. C near-callable as:
;      void DrawHCLineList(struct HLineList * HLineListPtr, int Color);

SCREEN_SEGMENT  equu     0a000h
GC_SEGMENT_SELECT equ   03cdh

HLine   struc
XStart  dw      ?       ;X coordinate of leftmost pixel in line
XEnd    dw      ?       ;X coordinate of rightmost pixel in line
HLine   ends

HLineList struc
Lngth   dw      ?       ;# of horizontal lines
YStart  dw      ?       ;Y coordinate of topmost line
HLinePtr  dw    ?      ;pointer to list of horz lines
HLineList ends
```

```
Parms   struc
                dw      2 dup(?) ;return address & pushed BP
HLineListPtr    dw      ?       ;pointer to HLineList structure
Color           dw      ?       ;color with which to fill
Parms   ends

; Advances both the read and write windows to the next 64K bank.
; Note: Theoretically, a delay between IN and OUT may be needed under
; some circumstances to avoid accessing the VGA chip too quickly, but
; in actual practice, I haven't found any delay to be required.
INCREMENT_BANK  macro
        push    ax              ;preserve fill color
        push    dx              ;preserve scan line start pointer
        mov     dx,GC_SEGMENT_SELECT
        in      al,dx           ;get the current segment select
        add     al,11h          ;increment both the read & write banks
        out     dx,al           ;set the new bank #
        pop     dx              ;restore scan line start pointer
        pop     ax              ;restore fill color
        endm

        .model small
        .data
        extrn   _BitmapWidthInBytes:word
        .code
        public _DrawHCLineList
        align   2
_DrawHCLineList proc    near
        push    bp              ;preserve caller's stack frame
        mov     bp,sp           ;point to our stack frame
        push    si              ;preserve caller's register variables
        push    di
        cld                     ;make string instructions inc pointers
        mov     ax,SCREEN_SEGMENT
        mov     es,ax   ;point ES to display memory for REP STOS
        mov     si,[bp+HLineListPtr] ;point to the line list
        mov     ax,[_BitmapWidthInBytes] ;point to the start of the
        mul     [si+YStart]     ; first scan line on which to draw
        mov     di,ax           ;ES:DI points to first scan line to
        mov     al,dl           ; draw; AL is the initial bank #
                                ;upper nibble of AL is read bank #,
        mov     cl,4            ; lower nibble is write bank # (only
        shl     dl,cl           ; the write bank is really needed for
        or      al,dl           ; this module, but it's less confusing
                                ; to point both to the same place)
        mov     dx,GC_SEGMENT_SELECT
        out     dx,al           ;set the initial bank
        mov     dx,di           ;ES:DX points to first scan line
        mov     bx,[si+HLinePtr] ;point to the XStart/XEnd descriptor
                                ; for the first (top) horizontal line
        mov     si,[si+Lngth]   ;# of scan lines to draw
        and     si,si           ;are there any lines to draw?
        jz      FillDone        ;no, so we're done
        mov     ax,[bp+Color]   ;color with which to fill
        mov     bp,[_BitmapWidthInBytes] ;so we can keep everything
                                ; in registers inside the loop
                                ;***stack frame pointer destroyed!***
FillLoop:
        mov     di,[bx+XStart]  ;left edge of fill on this line
        mov     cx,[bx+XEnd]    ;right edge of fill
        sub     cx,di
        jl      LineFillDone    ;skip if negative width
        inc     cx              ;# of pixels to fill on this line
        add     di,di           ;*2 because pixels are 2 bytes in size
        add     dx,bp           ;do we cross a bank during this line?
        jnc     NormalFill      ;no
        jz      NormalFill      ;no
                                ;yes, there is a bank crossing on this
                                ; line; figure out where
        sub     dx,bp           ;point back to start of line
        add     di,dx           ;offset of left edge of fill
        jc      CrossBankBeforeFilling ;raster splits before the left
                                ; edge of fill
        add     cx,cx           ;fill width in bytes (pixels * 2)
        add     di,cx           ;do we split during the fill area?
        jnc     CrossBankAfterFilling ;raster splits after the right
        jz      CrossBankAfterFilling ; edge of fill
                                ;bank boundary falls within fill area;
                                ; draw in two parts, one in each bank
        sub     di,cx           ;point back to start of fill area
        neg     di              ;# of bytes left before split
        sub     cx,di           ;# of bytes to fill to the right of
                                ; the bank split
        push    cx              ;remember right-of-split fill width
        mov     cx,di           ;# of left-of-split bytes to fill
        shr     cx,1            ;# of left-of-split words to fill
        neg     di              ;offset at which to start filling
        rep     stosw           ;fill left-of-split portion of line
        pop     cx              ;get back right-of-split fill width
        shr     cx,1            ;# of right-of-split words to fill
                                ;advance to the next bank
        INCREMENT_BANK          ;point to the next bank (DI already
                                ; points to offset 0, as desired)
        rep     stosw           ;fill right-of-split portion of line
        add     dx,bp           ;point to the next scan line
        jmp     short CountDownLine ; (already advanced the bank)
;======================================================================
        align   2               ;fill area is entirely to the left of
CrossBankAfterFilling:          ; the bank boundary
        sub     di,cx           ;point back to start of fill area
        shr     cx,1            ;CX = fill width in pixels
        jmp     short FillAndAdvance ;doesn't split until after the
                                ; fill area, so handle normally
;======================================================================
        align   2               ;fill area is entirely to the right of
CrossBankBeforeFilling:         ; the bank boundary
        INCREMENT_BANK          ;first, point to the next bank, where
                                ; the fill area resides
        rep     stosw           ;fill this scan line
        add     dx,bp           ;point to the next scan line
        jmp     short CountDownLine ; (already advanced the bank)
;======================================================================
        align   2               ;no bank boundary problems; just fill
NormalFill:                     ; normally
        sub     dx,bp           ;point back to start of line
        add     di,dx           ;offset of left edge of fill
FillAndAdvance:
        rep     stosw           ;fill this scan line
LineFillDone:
        add     dx,bp           ;point to the next scan line
        jnc     CountDownLine   ;didn't cross a bank boundary
        INCREMENT_BANK          ;did cross, so point to the next bank
CountDownLine:
        add     bx,size HLine   ;point to the next line descriptor
        dec     si              ;count off lines to fill
        jnz     FillLoop
FillDone:
        pop     di              ;restore caller's register variables
        pop     si
        pop     bp              ;restore caller's stack frame
        ret
;====================================================================
_DrawHCLineList endp
        end
```

Simple Unweighted Antialiasing

As the saying goes, you can never be too rich, too thin, or have too many colors available. Personally, I only buy one of those three assertions: You really can't have too many colors. Listings G.5 and G.6, together with Listings G.1, G.3, and G.7, FILCNVXD.C from the previous chapter, and Listing 39.4 from Chapter 39, show why. This program draws the same cube, but this time employing the simple, unweighted antialiasing we used in the previous chapter—and taking advantage of the full color range of the Hicolor DAC. The results are excellent: On my venerable NEC MultiSync, at a viewing distance of one foot, all but two of the edges look absolutely smooth, with not the slightest hint of jaggies, and the two imperfect edges show only slight ripples. At two feet, the cube looks perfect. The difference between the non-antialiased and antialiased cubes is astounding, considering that we're working with the same resolution in both cases.

A quick review of the simple antialiasing used in this chapter and the previous one: The image is drawn to a memory buffer at a multiple of the resolution that the actual screen supports. Each pixel on the screen maps to a group of hi-res pixels (subpixels), arranged in a square in the memory buffer. The colors of the subpixels in each square are averaged, and the corresponding screen pixel is set to the average subpixel color.

There's not enough memory to scan out the entire image at high resolution (about 50K is required just to scan out one 800×600 raster line at 4× resolution!), so Listing G.6 scans out just those pixels that lie in a specified band. (Each band corresponds to a single raster line in Listing G.5.) Note that Listing G.6 draws 32-bit pixels to the memory buffer; this is true color, plus an extra byte for flexibility. Consequently, Listing G.6 is a general-purpose tool, and can be used with any sort of adapter, as long as the main program knows how to convert from true color to adapter-specific pixels. Listing G.5 does this by calculating the average intensity in each subpixel group of each of the three primary colors, in the range 0–255, then looking up the gamma corrected equivalent color value for the Hicolor DAC, mapped into the range 0–31.

tip *A quick look at the gamma-corrected color mapping table in Listing G.5 shows why hardware gamma correction is sorely missed in the Hicolor DAC. The brightest half of the color range—from half intensity to full intensity—is spanned by only 9 of the Hicolor DAC's 32 color values. That means that for brighter colors, the Hicolor DAC effectively has only half the color resolution that you'd expect from 5 bits per color gun, and the resolution is even worse at the highest intensities.*

I'd like to take a moment to emphasize that although Listing G.5 works with only the three primary colors, it could just as easily work with the thousands of colors that can be produced as mixes of the three primaries; there are none of the limitations of 256-color mode, and no special tricks (such as biasing the palette according to color frequency) need be used. Inevitably, though, proportionately fewer intermediate blends are available and hence antialiasing becomes less precise when there is less contrast between colors; you're not going to be able to do much antialiasing between a pixel with a green true color value of 250 and another with a value of 255. This is where the lack of gamma correction and the difference between 15-bpp and true color become apparent.

LISTING G.5 LG-5.C

```
/* Demonstrates unweighted antialiased drawing in 640x480 Hicolor (32K 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 the
   color specified by RED, GREEN, AND BLUE, with all vertices
   offset by (x,y), to ScanLineBuffer, at ResMul multiple of
   horizontal and vertical resolution. The address of ColorTemp is
   cast to an int to satisfy the prototype for FillCnvxPolyDrvr; this
   trick will work only in a small data model. */
#define DRAW_POLYGON_HIGH_RES(PointList,RED,GREEN,BLUE,x,y,ResMul) { \
   Polygon.Length = sizeof(PointList)/sizeof(struct Point);          \
   Polygon.PointPtr = PointTemp;                                     \
   /* Multiply all vertical & horizontal coordinates */              \
   for (k=0; k<sizeof(PointList)/sizeof(struct Point); k++) {        \
      PointTemp[k].X = PointList[k].X * ResMul;                      \
      PointTemp[k].Y = PointList[k].Y * ResMul;                      \
   }                                                                 \
   ColorTemp.Red=RED; ColorTemp.Green=GREEN; ColorTemp.Blue=BLUE;    \
   FillCnvxPolyDrvr(&Polygon, (int)&ColorTemp, x, y, DrawBandedList);}
#define SCREEN_WIDTH 640
#define SCREEN_SEGMENT 0xA000

void main(void);
extern void DrawPixel(int, int, char);
extern void DrawBandedList(struct HLineList *, struct RGB *);
extern int SetHCMode(int);

/* Table of gamma corrected mappings of linear color intensities in
   the range 0-255 to the nearest pixel values in the range 0-31,
   assuming a gamma of 2.3 */
```

```
static unsigned char ColorMappings[] = {
    0, 3, 4, 4, 5, 6, 6, 6, 7, 7, 8, 8, 8, 8, 9, 9, 9,10,10,10,
   10,10,11,11,11,11,11,12,12,12,12,12,13,13,13,13,13,13,14,14,
   14,14,14,14,14,15,15,15,15,15,15,15,16,16,16,16,16,16,16,16,
   17,17,17,17,17,17,17,17,17,18,18,18,18,18,18,18,18,18,19,19,
   19,19,19,19,19,19,19,19,20,20,20,20,20,20,20,20,20,20,20,21,
   21,21,21,21,21,21,21,21,21,21,22,22,22,22,22,22,22,22,22,22,
   22,22,22,23,23,23,23,23,23,23,23,23,23,23,23,24,24,24,24,24,
   24,24,24,24,24,24,24,24,24,25,25,25,25,25,25,25,25,25,25,25,
   25,25,25,26,26,26,26,26,26,26,26,26,26,26,26,26,26,26,27,27,
   27,27,27,27,27,27,27,27,27,27,27,27,27,27,28,28,28,28,28,28,
   28,28,28,28,28,28,28,28,28,28,28,29,29,29,29,29,29,29,29,29,
   29,29,29,29,29,29,29,29,30,30,30,30,30,30,30,30,30,30,30,30,
   30,30,30,30,30,30,31,31,31,31,31,31,31,31,31,31};
/* Pointer to buffer in which high-res scanned data will reside */
struct RGB *ScanLineBuffer;
int ScanBandStart, ScanBandEnd;  /* top & bottom of each high-res
                                    band we'll draw to ScanLineBuffer */
int ScanBandWidth;      /* # subpixels across each scan band */
int BitmapWidthInBytes = 640*2;  /* # of bytes per raster line in
                                    Hicolor VGA display memory */
void main()
{
   int i, j, k, m, Red, Green, Blue, jXRes, kXWidth;
   int SubpixelsPerMegapixel;
   unsigned int Megapixel, ResolutionMultiplier;
   long BufferSize;
   struct RGB ColorTemp;
   struct PointListHeader Polygon;
   struct Point PointTemp[4];
   static struct Point Face0[] =
         {{396,276},{422,178},{338,88},{288,178}};
   static struct Point Face1[] =
         {{306,300},{396,276},{288,178},{210,226}};
   static struct Point Face2[] =
         {{338,88},{266,146},{210,226},{288,178}};
   int LeftBound=210, RightBound=422, TopBound=88, BottomBound=300;
   union REGS regset;

   printf("Subpixel resolution multiplier:");
   scanf("%d", &ResolutionMultiplier);
   SubpixelsPerMegapixel = ResolutionMultiplier*ResolutionMultiplier;
   ScanBandWidth = SCREEN_WIDTH*ResolutionMultiplier;

   /* Get enough space for one scan line scanned out at high
      resolution horz and vert (each pixel is 4 bytes) */
   if ((BufferSize = (long)ScanBandWidth*4*ResolutionMultiplier) >
         0xFFFF) {
      printf("Band won't fit in one segment\n"); exit(0); }
   if ((ScanLineBuffer = malloc((int)BufferSize)) == NULL) {
      printf("Couldn't get memory\n"); exit(0); }

   /* Attempt to enable 640x480 Hicolor mode */
   if (SetHCMode(0x2E) == 0)
      { printf("No Hicolor DAC detected\n"); exit(0); };

   /* Scan out the polygons at high resolution one screen scan line at
      a time (ResolutionMultiplier high-res scan lines at a time) */
   for (i=TopBound; i<=BottomBound; i++) {
      /* Set the band dimensions for this pass */
      ScanBandEnd = (ScanBandStart = i*ResolutionMultiplier) +
            ResolutionMultiplier - 1;
      /* Clear the drawing buffer */
      memset(ScanLineBuffer, 0, BufferSize);
      /* Draw the current band of the cube to the scan line buffer */
      DRAW_POLYGON_HIGH_RES(Face0,0xFF,0,0,0,0,ResolutionMultiplier);
      DRAW_POLYGON_HIGH_RES(Face1,0,0xFF,0,0,0,ResolutionMultiplier);
      DRAW_POLYGON_HIGH_RES(Face2,0,0,0xFF,0,0,ResolutionMultiplier);

  /* Coalesce subpixels into normal screen pixels (megapixels) and draw them */
      for (j=LeftBound; j<=RightBound; j++) {
         jXRes = j*ResolutionMultiplier;
         /* For each screen pixel, sum all the corresponding
            subpixels, for each color component */
         for (k=Red=Green=Blue=0; k<ResolutionMultiplier; k++) {
            kXWidth = k*ScanBandWidth;
            for (m=0; m<ResolutionMultiplier; m++) {
               Red += ScanLineBuffer[jXRes+kXWidth+m].Red;
               Green += ScanLineBuffer[jXRes+kXWidth+m].Green;
               Blue += ScanLineBuffer[jXRes+kXWidth+m].Blue;
            }
         }
         /* Calc each color component's average brightness; convert
            that into a gamma corrected portion of a Hicolor pixel,
            then combine the colors into one Hicolor pixel */
         Red = ColorMappings[Red/SubpixelsPerMegapixel];
         Green = ColorMappings[Green/SubpixelsPerMegapixel];
         Blue = ColorMappings[Blue/SubpixelsPerMegapixel];
         Megapixel = (Red << 10) + (Green << 5) + Blue;
         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);
}
```

LISTING G.6 LG-6.C

```
/* Draws pixels from the list of horizontal lines passed in, to a 32-bpp
buffer; 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. Note that
Color here points to an RGB structure that maps directly to the buffer's pixel
format, rather than containing a 16-bit integer. Tested with Borland C++
in C mode in the small model. */

#include "polygon.h"

extern struct RGB *ScanLineBuffer;  /* drawing goes here */
extern int ScanBandStart, ScanBandEnd; /* limits of band to draw */
extern int ScanBandWidth;  /* # of subpixels across scan band */

void DrawBandedList(struct HLineList * HLineListPtr,
   struct RGB *Color)
{
   struct HLine *HLinePtr;
   int Length, Width, YStart = HLineListPtr->YStart, i;
   struct RGB *BufferPtr, *WorkingBufferPtr;
   /* 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 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) {
      /* Fill whole horiz line with Color if it has positive width */
      if ((Width = HLinePtr->XEnd - HLinePtr->XStart + 1) > 0) {
         WorkingBufferPtr = BufferPtr + HLinePtr->XStart;
         for (i = 0; i < Width; i++) *WorkingBufferPtr++ = *Color;
      }
      HLinePtr++;                /* point to next scan line X info */
      BufferPtr += ScanBandWidth; /* point to start of next line */
   }
}
```

LISTING G.7 POLYGON.H

```
/* POLYGON.H: Header file for polygon-filling code */

/* Describes a single point (used for a single vertex) */
struct Point {
   int X;   /* X coordinate */
   int Y;   /* Y coordinate */
};

/* Describes a series of points (used to store a list of vertices that
describe a polygon; each vertex is assumed to connect to the two adjacent
vertices, and the last vertex is assumed to connect to the first) */
struct PointListHeader {
   int Length;                /* # of points */
   struct Point * PointPtr;   /* pointer to list of points */
};

/* Describes the beginning and ending X coordinates of a single
   horizontal line */
struct HLine {
   int XStart; /* X coordinate of leftmost pixel in line */
   int XEnd;   /* X coordinate of rightmost pixel in line */
};

/* Describes a Length-long series of horizontal lines, all assumed to be on
contiguous scan lines starting at YStart and proceeding downward (used to
describe scan-converted polygon to low-level hardware-dependent drawing code)*/
```

```
struct HLineList {
   int Length;               /* # of horizontal lines */
   int YStart;               /* Y coordinate of topmost line */
   struct HLine * HLinePtr;  /* pointer to list of horz lines */
};

/* Describes a color as an RGB triple, plus one byte for other info */
struct RGB { unsigned char Red, Green, Blue, Spare; };
```

Notes on the Antialiasing Implementation

Listing G.5 features user-selectable subpixel resolution, which is the multiple of the screen resolution at which the image should be drawn into the memory buffer. A subpixel resolution of two times normal along both axes (2×) looks much better than nonantialiased drawing, but still has visible jaggies. Subpixel resolution of 4× looks terrific, as mentioned earlier. Higher subpixel resolutions are, practically speaking, reserved for 386 protected mode, because they would require a buffer larger than 64K to hold the high-resolution equivalent of a single scan line.

On the downside, Listing G.5 is very slow, even though the conversion process from true color pixels to Hicolor pixels is limited to the bounding rectangle for the cube being drawn, thereby saving the time that was wasted in the previous chapter drawing the empty space around the cube. It could easily be sped up by (say) an order of magnitude, in a number of ways. First, you could implement an ASM function that's the equivalent of **memset**, but stores longs (dwords) rather than chars (bytes). In the absence of any such C library function, Listing G.6 uses a loop with a pointer to a long; hardly a recipe for high performance.

Listing G.5 could also be sped up by doing the screen pixel construction from each square of subpixels in assembly language using pointers rather than array look-ups. It would also help to organize the screen pixel drawing more as a variant rectangle fill, instead of going through **DrawPixel** every time, so that the screen pointer doesn't have to be recalculated from scratch and the bank doesn't need to be calculated and set for every pixel. Clipping each polygon to the band before rather than after scanning it out would speed things up, as would building an edge list for the polygons once, ahead of time, then advancing it incrementally to scan out each band, rather than doing one complete scan of each polygon for each band. Bigger bands would help; drawing the whole image to the memory buffer in one burst, then converting the entire image to Hicolor pixels in a single operation, would be ideal, but would require a ridiculous amount of memory. (Would you believe, 31MB for one full 800×600 Hicolor screen at 4× resolution?)

Finally, to alter Listing G.5 for 800×600 Hicolor mode, change the parameter passed to **SetHCMode**, the value of **BitmapWidthInBytes**, and the value of **SCREEN_WIDTH**.

Further Thoughts on Antialiasing

The banded true color approach of Listings G.5 and G.6 is easily extended to other antialiasing approaches. For example, you could, if you wished, average together all the subpixels not within a square, but rather within a circle of radius **sqrt(2.0)*Resolution Multiplier /2** around each pixel center. This approach is a little more complicated, but it has one great virtue: An image will be antialiased identically, regardless of its rotation.

Why is the shape of the subpixel area that's collected into a screen pixel important when the maximum resolution we can actually draw with is the resolution of the screen? I'll quote William vanRyper, from the graphics.disp/vga conference on BIX:

"If you antialias an edge on the screen, and let the eye-brain pick the edge somewhere in the gradient between the object color and the background, you can adjust the placement of that perceptual edge by altering the ramp of the gradient. If the number of intermediate values you can choose among is greater than the number of gradient pixels you set (across the edge), you can adjust the position of the perceptual edge in increments of less than a pixel. This means you can locate the antialiased object to sub-pixel precision."

tip

In other words, by using blends of color in a smooth, consistent gradient across a boundary, you can get the eye to pick out the boundary location with a precision that's greater than the resolution of the screen. This is, of course, part and parcel of the wonderful eye/brain magic that allows color to substitute for resolution and makes antialiasing worthwhile.

Given that we can draw images with perceived resolution higher than the screen, consistency in subpixel placement is very important. Unfortunately, our simple square antialiasing does not produce the same results (a consistent color gradient) for an image rotated 45 degrees as it does for an unrotated image—but antialiasing based on a circular subpixel area does. So the shape of the subpixel area used for antialiasing matters because if it's not symmetric in all directions, boundaries will appear to wiggle as images rotate, destroying the image of reality that antialiased animation strives to create.

On the other hand, if you're drawing only static images, use a square subpixel area for antialiasing; it's fast, easy, and looks just fine in that context. As I said at the outset, we're not seeking mathematical perfection here, just a good-looking display for the purpose at hand. If it looks good, it *is* good.