Color Recovery: True-Color 8-Bit Interactive Graphics

Anthony C. Barkans
Hewlett-Packard

For many years, the only practical way to display high-quality true-color images was on a computer having a graphics subsystem with at least 24 color planes. The high cost of color graphics devices with 24 planes led many users to choose 8-, 12-, or 16-plane systems. Unfortunately, using a system with fewer color planes sacrifices some color capabilities to save money.

An innovation called Color Recovery provides a method for displaying millions of colors within the cost constraints of an eight-plane system. Figure 1 (next page) compares Color Recovery image quality to that of other color systems. The top image (Figure 1a) shows a close-up of a jet stored as a full 24-bit-per-pixel true-color image. Figure 1b shows the same jet displayed using a traditional 8-bit-per-pixel system. In Figure 1c we see the jet image resulting from using Color Recovery in an 8-bit-per-pixel mode on a Hewlett-Packard HCRX-8 graphics device.

Of course, pretty pictures aren't enough. Therefore, one of the primary design goals for Color Recovery was to supply the additional color capabilities without giving up interactive performance. Another goal was to be able to work with all types of applications running in a windowed environment. An implementation of Color Recovery described in this article meets these goals.

Background

Traditional interactive graphics systems with eight color planes can display only 2^8 (256) colors. Two approaches have been employed to get the best results with limited colors. The first, called palletized color, pseudo color, or indexed color, selects a set of 256 colors and limits the application to using only that fixed set of colors. For many applications, such as word processing and business graphics, this approach works well.

However, applications needing more than 256 colors, such as realistically shaded MCAD images or human faces in video sequences, require another approach. A technique called dithering[1] can simulate more colors for these applications. It approximates a single color by displaying two other colors at intermixed pixel locations. The primary problem with dithering is that since people tend to work close to the display, dithered images appear grainy or textured. For an example, see the dithered jet shown in Figure 1b.

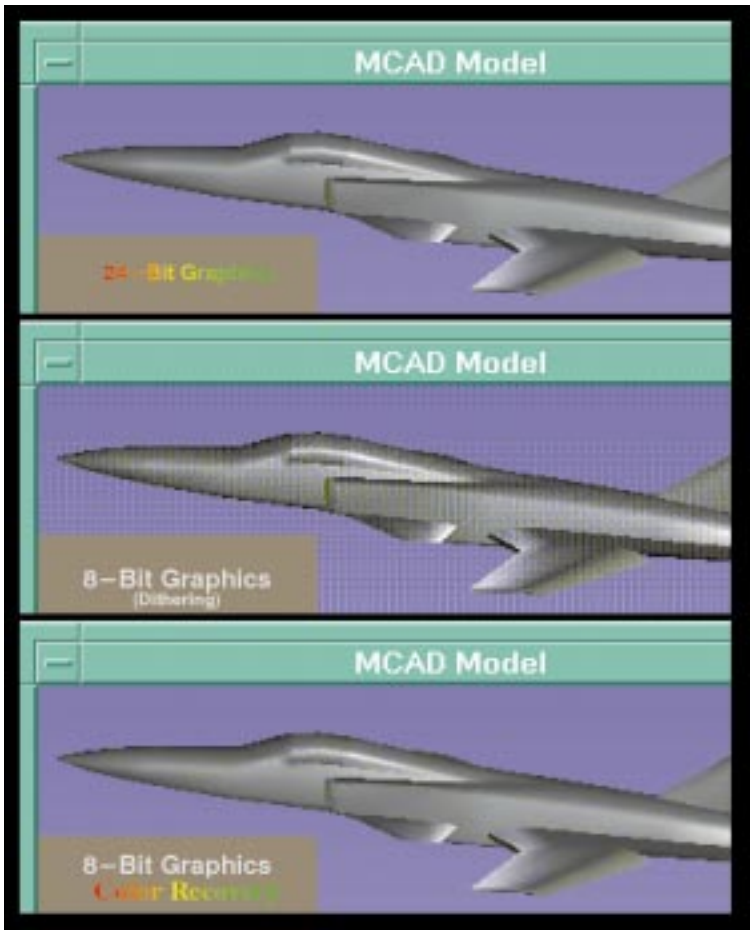
The cost advantages of eight-plane systems have motivated several researchers to derive techniques to display high-quality images using only eight color planes. These investigators have turned to image compression to preserve image quality. One noticeable example developed in the late 1980s by Edsun Laboratories[2] used a modified run-length encoding algorithm. However, there are many well-known limitations to using data compression in an interactive environment, as addressed in a classic 1970s text by Newman and Sproull.[3] Two primary problems arise in working with a compressed image. First, the compressed data is difficult to modify. Second, pixel-level operations such as BitBlt's, logical operations (pixel-level AND, OR, XOR, and so on), and blending operations cannot be performed easily. In addition, the encoded frame buffer can be viewed only on systems with a specialized decoder. These limitations would apply whether the compression is as simple as run-length encoding or as complex as JPEG.

Color Recovery combines dithering with a novel filtering technique to yield image quality comparable to 24-bit color. Using only eight color planes, it provides cost-efficient graphics.

We must realize that in essence two types of interactive environments exist. In the first, the user interacts with the data; this includes CAD applications or games. The second sends data in a one-way stream to the user and includes applications that display video. Note that the first case requires the most flexible image storage scheme, since a user's action can change any individual image. The second type, which produces no image modification within a frame (such as video), can exploit the orderly sequential scanning of the image. In this case compression has had some success.

For example, the HP series 700 can use disk storage for images compressed in either JPEG or MPEG format. The CPU then decodes the images and sends them to

0272-1716/97/$10.00 © 1997 IEEE



1 (a) 24-bit true-color image from an interactive sequence. (b) Same data set of the jet displayed on a traditional 8-bit-per-pixel system that uses dither to simulate true color. (c) Same data set of the jet displayed on an 8-bit-per-pixel system that supports Color Recovery.

the display at interactive rates.[4] Additionally, in some systems, data stored in a limited depth frame buffer can be compressed and then displayed with more colors than an eight-color plane system might be expected to produce. One such system using a compression algorithm called HAM (Hold and Modify) was embodied in the old Amiga computers[5] from Commodore. However, for a general-purpose interactive windowed environment capable of supporting all applications, data compression techniques have not succeeded very well.

Although this article focuses on interactive graphics, there is a technique called descreening[6] that is sometimes used in the printing industry to reconstruct a continuous-tone image from a half-tone image. Descreening analyzes the image to find the dither pattern used to encode the image. Then a mask of how the dither pattern covers the image is constructed. Next, the pixels within the image are compared to the dither pattern to estimate the original value. This method gives good results, but it is computationally expensive and is typically done as a postprocessing step on an entire image. Color Recovery, on the other hand, is an interactive process that does not require knowledge of how the dither pattern aligns to the image.

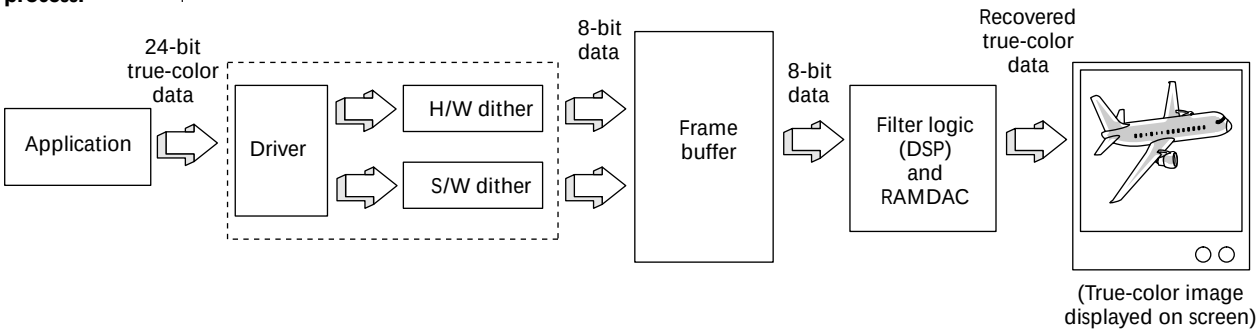
Color Recovery

Simply explained, Color Recovery performs the task your visual system does when using an ordinary dithered system. With a dithered system, your visual system is used to reconstruct the true-color data from the displayed image. The Color Recovery system performs the color reconstruction using a filtering circuit, thus presenting your eye with a true-color image. Figure 2 shows the entire process. In essence, a Color Recovery system takes 24-bit true-color data generated by an application and dithers it to 8-bit data for storage in the frame buffer. Then, as the 8-bit data is scanned from the frame buffer to the display, it passes through specialized digital signal processing (DSP) hardware that reconstructs the original true-color image. The recovered true-color data then goes to the display, where it appears in millions of unique colors.

Color Recovery theory

To gain insight into how Color Recovery works, we can view the process in both the spatial and the frequency domains (as suggested to me by T. Malzbender). As I will discuss later, the Color Recovery filter is adaptive and therefore not amenable to simple linear systems theory. However, for regions of uniform color content, the filter is fixed and permits a frequency analysis, as shown in Figure 3. (For simplicity, the analysis is shown in one dimension, as opposed to the two dimensions that

2 A schematic illustration of the Color Recovery process.



the process optimally requires to work on the 2D screen of a computer display.)

To begin, assume that we wish to show a uniform field of color on the display. The color is specified to some arbitrary precision—for example, the color could be typical 24-bit true color with 8 bits each of red, green, and blue. In the spatial domain, the color would be flat across any single scan line, such as a scan line of the sky in Figure 1a. This appears at the top of Figure 3 as the original true-color data in the spatial domain. Since the color is assumed to be a flat field, it has only a DC component in the frequency domain. The top of Figure 3 shows this as the original true-color data in the frequency domain.

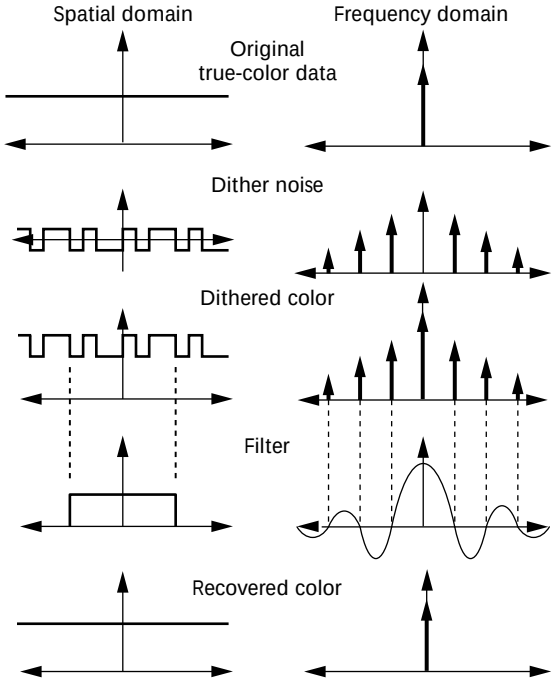
Viewed by itself, dither appears as noise. Using a simple ordered dither gives rise to periodic dither noise as shown in the second set of drawings in Figure 3. When the dither is combined with the original true-color data, the resulting dithered color will typically have a pattern that uses two quantized levels to approximate the original data. Note that the DC component of the dithered signal should be unaffected by the dithering process. However, since the human eye can typically discern the two quantized levels, patterns appear in the dithered color. The dithered color shown in the third set of drawings in Figure 3 represents the data stored in the frame buffer. This is the data typically displayed by most dithered systems, as shown in Figure 1b.

Much research has focused on making dithered images look better using two classes of techniques. The first breaks up the ordered dither.[7] An analysis of this technique shows that its application spreads the dither energy across a broader spectrum in the frequency domain. This reduces the energy in the biggest noise spikes, as shown in the frequency domain drawings of dither noise in Figure 3.

The second technique to make dithered images look better more closely matches the quantizing levels to the response of the human eye.[8] An analysis of the nonlinear quantization method shows that it allows the dither noise energy to be larger where the human eye is less sensitive and reduce the dither energy where the eye is more sensitive. To sum up, these two techniques seek an optimal way to redistribute the dither energy.

With Color Recovery, true-color data is dithered and then stored in the frame buffer. However, instead of redistributing the dither energy, Color Recovery attempts to remove the dither noise with a filter. The Color Recovery filter is shown in the fourth set of drawings in Figure 3. In the spatial domain, this is a simple box filter with width equal to the periodicity of the dither noise. In the frequency domain, this filter has a transfer function of a Sinc. Choosing this width for the box filter causes the zero crossings of the Sinc to coincide with locations where the dither noise contributes energy.

The box filter removes all dither noise perfectly in areas of constant color. Note that the amplitude of the Sinc can be set to unity by controlling the amplitude of the box filter. This causes the signal's DC component to pass through the filter undisturbed. Therefore, in a region of uniform color the filter does a perfect job of recovering the true color and eliminating the noise, as shown by the recovered color in the bottom set of drawings in Figure 3.



3 The Color Recovery process in the spatial domain and the frequency domain.

The claim of a perfect filter in regions of constant color is not an overstatement. Most who have studied filtering theory have seen it presented in digital signal processing textbooks. These show the typical filter recovering a signal from an environment of white noise. In these cases, the ideal filter would be a box in the frequency domain. Since such a filter cannot be realized, some noise passes with the signal through the filter. However, dither noise is unique in that its energy spectrum can be controlled as shown in Figure 3. Controlling the noise being injected onto the color signal allows perfect reconstruction of the color signal.

Color Recovery dither process

For the dither cell to work best with Color Recovery, it must meet two criteria. First, it must be periodic, and second, it must match the shape of the filter region. Using the simple ordered dithering technique discussed in most computer graphics textbooks results in a periodic dither. (Any dither that repeats based on the pixel address, such as that described in the sidebar on p. 70, is periodic.) The second criterion, that of the dither cell matching the filter shape, requires coordinating filter shape design with the design of the dither. However, once selected, the filter size is easily accommodated in the dither design.

Note that when using Color Recovery the size of the dither/filter region determines how well a color can be recovered. A region of 2^N pixels can recover about N bits of color per component. In theory, this means we can recover any number of bits of color if we make the dither/filter region large enough. For example, a 10-bit component of color could be recovered using a single bit of storage per pixel if a 1,024-pixel dither/filter

Overview of Color Theory and Dither

Color theory

Computer monitors use red, green, and blue (RGB) to produce true-color images. A reasonable question to ask is, "Why use these particular colors?"

The human eye contains three types of receptor cells. One set of cells responds to red, another to green, and a third to blue. Thus any visible color is "sensed" by the varying response of these three types of cells. Thus, in computer graphics, we can mix varying amounts of red, green and/or blue to create any color. For example, forcing both the red and green CRT color beams to be on at any single location will result in a dot that appears yellow to the human eye. The human eye can also perceive a new color when the component colors are mixed spatially. As an example, a checkerboard of red and green pixels will be perceived as yellow when viewed from a distance. Dither exploits this spatial color mixing to form new colors.

Another reasonable question to ask is, "Why is 24 bits of color per pixel called *true color*?" Using 24 bits per pixel for color usually allocates 8 bits to each of the color channels. This means that each color can be set at one of 256 levels. Note that under ideal conditions the eye can sense more than 256 levels of a single color. However, the range of any display cannot match the entire range of the human eye. Therefore, a realistic goal is to display a surface shaded from black to full intensity such that the shading appears continuous. This requires quantizing the color to match the eye's response within the subrange of the display medium. For instance, feature film designed to be shown in a dark theater may use 10 bits per color component. A typical CRT used in an office environment can meet the goal with judicious use of 8 bits per color component.

The term *true color* should therefore refer to the reproduction of a smoothly varying color such that the underlying digital quantization of a color is not discernible by the human eye. However, most users are familiar with CRTs and thus take the use of 24 bits of color per pixel to mean *true color*.

Dither

A typical CRT-based graphics system specifies any displayable color component using eight binary bits. Let's examine the red component as an example. When there is no red in a pixel, the red component is specified with a binary value of 00000000, which is a decimal 0. A full bright red is specified as a binary number 11111111, a decimal 255. Of course, high-end display systems can store and display the full 24 bits of true-color information. But lower-cost systems typically have only eight bits per pixel to store the color information and must therefore approximate the true-color image.

The most common method used to do this is dither. Dither uses three bits for the red and green components, leaving only two bits for blue. (We use fewer bits for blue because the human eye is less sensitive to blue.) With fewer bits available per color component, the quantization of the colors becomes apparent, as seen in the middle jet image (Figure 1b).

Basically, dither approximates a color using a combination of colors at adjacent pixels. Thus, viewed from a distance the image will appear to be the correct color. Since dithered systems can store only a limited number of bits in the frame buffer, the dithering logic must select the best set of values to use. Think of each 8-bit binary component of color as a number with a binary point that has three bits in front of it and five bits behind (I designate this "three point five" format). This format is useful because the final result of the dithering operation is a 3-bit number that approximates the integer part of the true-color value. For example, assume the true-color value for red is given as the binary number 01011000. In a three point five representation, the number becomes 010.11000 binary, which we can call 2.75 decimal.

In this example, the dithered value of the red color is stored in the frame buffer as a 3-bit value. If we assume these 3-bit values are integers, using integer values 2 and 3 for the final dithered data would be desirable. For instance, the dither could set 3/4 of the pixels to level 3 and 1/4 to level 2.

The dithering algorithm starts by having a table

region is used. However, certain practical considerations limit this. One very practical implementation is to use an 8-bit frame buffer that stores data in 3-3-2 format. In this case, a dither region of 2^5 (32) pixels for each color component can recover five additional bits. Using a 32-pixel dither region, an area in the image of uniform color can have the same visual quality as an 8-8-7 image.

For example, the sky behind the jet in the Color Recovered image shown in Figure 1c has been recovered to within one bit of the original 24-bit true-color data shown in Figure 1a. With the recovery of five bits of color data per component, regions of near-constant color in an 8-bit Color Recovery image and a true-color 24-bit image are visually indistinguishable. Therefore, the implementation of Color Recovery uses a dither table with 32 entries organized as 2×16. (I will discuss the reason for this odd shape later in the article.)

Color Recovery filtering process

The Color Recovery filtering process is best described by stepping through two examples. The first example assumes that an image consists of an entire screen covered with a shade of red. Using the 2.75 color (in the sidebar) to dither to three bits will result in 3/4 of the pixels stored in the frame buffer set to 3 and 1/4 of the pixels set to 2. It is easy to see that if we average any four pixels

of values indexed by the window *X* and *Y* address. With the original 8-bit color components in three point five format, the values stored in the dither table range from 0.0 to almost 1.0. We add the output of the table to the original 8-bit color component, then truncate the value to the desired number of bits for storage in the frame buffer.

As a simple example, assume that we continue to work with a red component originally specified as the binary number 01011000. Assume also that we use a 2×2 dither (as shown in Table A) to store the dithered color data using 3 bits. (The notation 2×2 dither means that when the dither pattern is aligned to the display window it will repeat in a 2×2 grid across the image.) To use a 2×2 dither, we use the least significant bit (LSB) of the *X* and *Y* address to index the dither table.

At the upper left of the window, *X* and *Y* addresses are both 0. To dither the data for this pixel location, we follow the steps listed in Table B.

At address 0,0 we would therefore store a 011 binary in the frame buffer for red. Applying the above dither would result in three of the four pixels within every 2×2 pixel block having a value of 011. The fourth pixel in each block—those with the LSB of *Y* set to 1 and the LSB of *X* set to 0—will have a 010 stored in the frame buffer, as shown in Figure A. When viewing a region of this red from a distance, it appears to be the correct value of 010.11000.

Note that in this example we have worked on only the red component of color. In a typical system a similar operation would also be done on the green and blue data. However, the blue would be dithered to only two bits. Thus the original 24-bit true-color data would be dithered at each pixel and stored in the frame buffer as three bits each of red and green and two bits of blue (3-3-2 format).

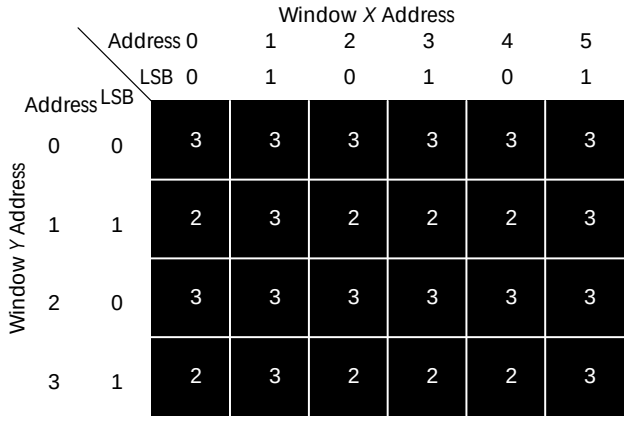
Examining the image of the dithered jet (Figure 1b), we can see that it is dithered using a method similar to this example. Note that from a distance the eye blends the colors in the dithered image such that they appear correct. However, the fundamental problem with dither is that most dithered images are viewed up close, making the dithering pattern noticeable in the image.

Table A. An example 2×2 dither.

LSB of *Y*	LSB of *X*	Dither Value Binary	Dither Value Decimal
0	0	.10100	.625
0	1	.01100	.375
1	0	.00100	.125
1	1	.11100	.875

Table B. Steps for dithering data at pixel location 0,0.

	In Binary	In Decimal	Comments
Input color	010.11000	2.750	Original true-color data for red
Dither value	.10100	.625	Value looked up from dither table
Result	011.01000	3.250	Result of adding dither to true/input color
Truncated 3-bit value	011	3	Red data stored in dithered frame buffer for pixel address 0,0



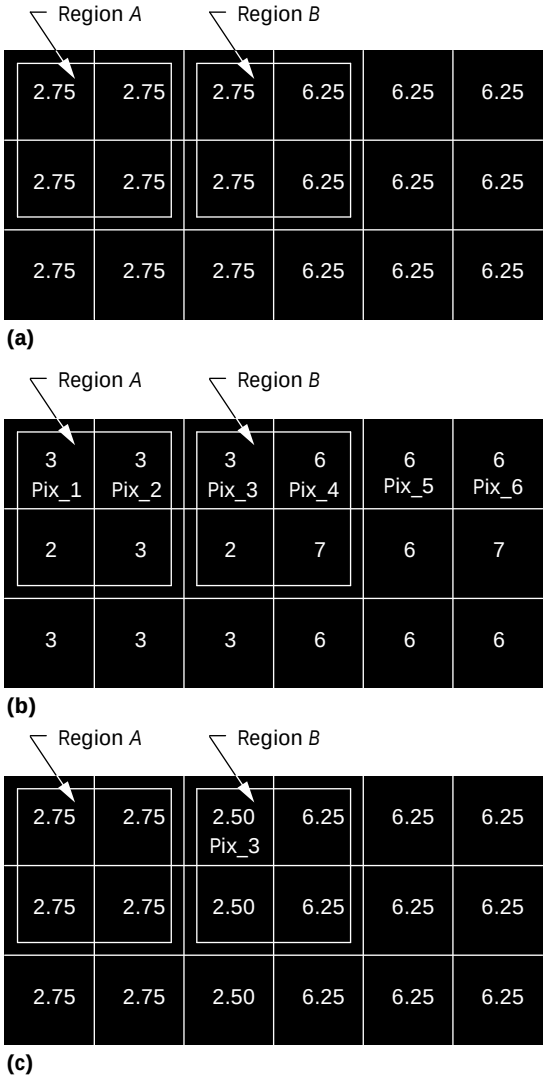
A Results of applying dither. Each box represents a pixel location on the display screen. For example, address (0,0) is defined as the upper left corner of the display. The numbers stored at each pixel location represent the results of applying the dither values given in Table A to a component of color originally specified as 01011000 binary (2.75 in our notation).

in any 2×2 region, we will recover the original color. This can be done using the example data as follows:

$$([3 * 3] + [2 * 1])/4 = 2.75$$

This averaging works very well in regions of constant color, such as the sky behind the jet. However, one fundamental issue must be addressed for Color Recovery to be viable: how to handle edges in the image. If edges are not accounted for, the resultant image will blur. The second example to show the Color Recovery filtering process will assume that there is an edge in the image data. The 2D representations of an area of a display screen (shown in Figure 4, next page) illustrate this example.

Each box in Figure 4 represents a pixel location on the display screen. In Figure 4a the numbers represent the original true-color data for one of the components (such as red) in a 24-bit-per-pixel system. (The numbers appear in a decimal notation described in the sidebar.) Figure 4b shows the same region after simple dithering. This is the data stored in the frame buffer. Figure 4c shows the pixel values after processing by the Color Recovery filter. Note that the pixel values in Figure 4c represent the color data that would be displayed on the computer display.



4 (a) Full-precision pixel data for a component of color. This is similar to data that would be stored for the image shown in Figure 1a. (b) The color data from Figure 4a after it has been dithered and placed in the frame buffer. This is the data that would be displayed in a typical dithered system, with results similar to Figure 1b. (c) The pixels written to the display when using Color Recovery with the frame buffer data as shown in Figure 4b. This is similar to the image shown in Figure 1c.

Region A in each of these figures is an area of constant color, whereas region B encompasses an edge. For illustration purposes the dither region is assumed to be 2×2 pixels.

When it is time to display Pix_1, the data for the four pixels shown as Region A (Figure 4b) would be sent to the filter. The data stored in the region could be summed and then divided by the number of pixels in the region. The sum of the pixels in Region A is 11, and 11 divided by 4 is 2.75. Thus the output of the filter when evaluating Pix_1 would be 2.75. This output value would be displayed on the computer display at Pix_1's location. Note that the output of the filter is the exact value of the original data at that point in Figure 4a.

The next pixel along the scan line to be evaluated is Pix_2. The filter region for Pix_2 includes the two rightmost pixels of region A and the two leftmost pixels of region B (see Figure 4b). Applying the filter operation for Pix_2 again results in the output value matching the original true-color data at that location in Figure 4a.

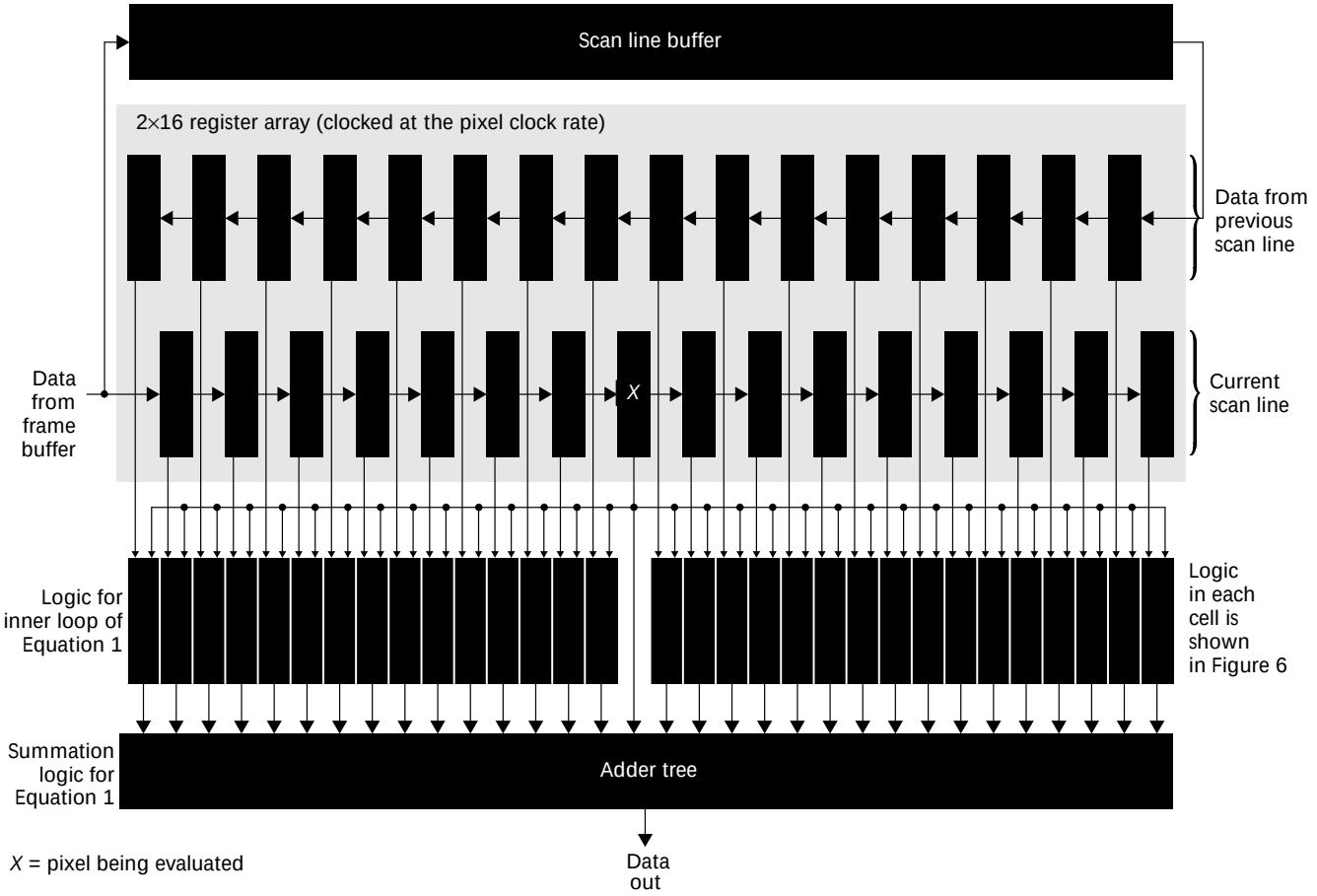
In evaluating Pix_3, summing the pixels in region B and then dividing them by the number of pixels in the region gives us 4.50, which differs greatly from the original data value of 2.75 shown in Figure 4a. Using the value of 4.50 at Pix_3 will result in the edge smearing. To solve this problem, we can use a special edge detector that looks for edges in noisy data. This lets us compare each pixel in the filter region with a value within ±1 of the pixel being evaluated. When we evaluate Pix_3, the data stored in the frame buffer is a 3 (Figure 4b). Thus, only pixels within region B that have a value of 2, 3, or 4 stored in the frame buffer would pass the edge comparison. We then sum the values that pass the edge detector and divide the total by the number of pixels that pass the edge comparison. In the Pix_3 evaluation, only Pix_3 and the pixel below it would pass. Summing the two passing values together and dividing by 2 gives us 2.50. This value differs slightly from the original value of 2.75 but gives us a better estimation to the original than the 4.50 obtained without the edge detection. The displayed values for the entire example region are shown in Figure 4c.

We must note three things about the displayed value at location Pix_3. First, dynamically altering the filter shape brings the resultant value closer to the original data than the value produced by the nonaltered filter function. Second, using the edge detectors to adjust the filter shape causes a mismatch between the effective filter shape and the shape of the dither cell. Referring back to Figure 3, we can see that when the filter shape no longer matches the shape of the dither cell, some dither noise will pass through the filter. Third, the visual system responds differently to edges than flat fields of color.[9] For example, the well-known Mach banding effect[1] results from how the human eye perceives edges.

Small color errors at image edges are often not detected by the human eye and therefore do not pose a significant problem. However, loss of sharpness along an edge is more noticeable. Therefore, a primary concern for the design of Color Recovery is making the edge detectors as sensitive as possible.

Software considerations

With dithered frame buffers in common use today, it is well known that existing software can work with a dithered frame buffer. The primary difference between a common dithered system and a Color Recovery system is that with Color Recovery software does not read the value being displayed on the screen. For example, the sky behind the jet in Figure 1c appears smooth. However, the data stored in the frame buffer is dithered



5 Block diagram of the Color Recovery filter.

and when read would appear more like the sky shown in Figure 1b. To prevent this, we could read a region of the frame buffer and perform the filtering operations in software. This could be slow, however, and there is no guarantee that the filter will output the exact same color value as originally specified. (For an example, refer back to Pix_3 shown in Figure 4c.)

Note that this problem is common among systems that support a dithered frame buffer. That is, once true-color data is written to any dithered frame buffer, there is no guarantee that frame buffer reads can return the true-color data. However, since the data stored in the frame buffer is not directly displayed when using Color Recovery, we must consider what type of capabilities to report when a window is created. That is, should the system assume a frame buffer orientation and report an 8-plane device or assume a display orientation and report a true-color device? This problem arises because historically an assumption has been made that the frame buffer directly maps to display capability.

On all HP graphics workstations introduced since January 1994, we have chosen to enable Color Recovery as the default for 3D applications when using an 8-bit color visual. Thus, when you use the Starbase library, PHIGS, or PEXlib, and open an application in a true-color mode, Color Recovery will normally be enabled. Of course, setting an application to use a pseudocolor map will disable Color Recovery and give the application the desired pseudocolor capability. Because Xlib is more tied into the old color map model than the 3D libraries, Xlib applications leave Color Recovery off. (You can enable it, however, when using Xlib with the application doing the dithering.)

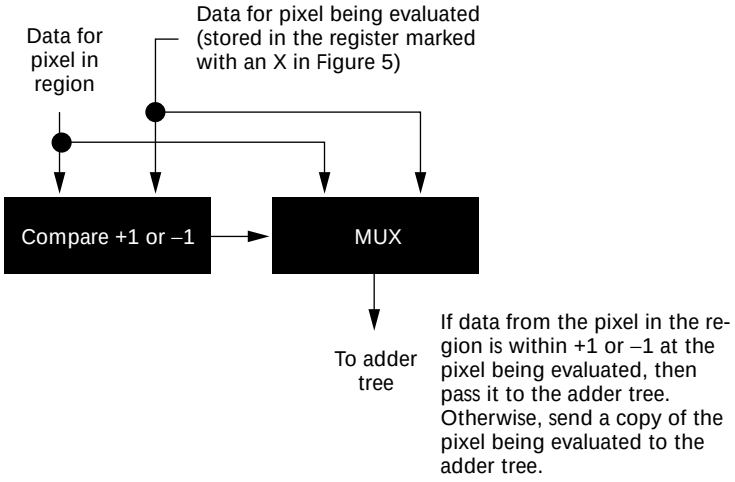
In addition to HP-supplied APIs, Next uses Color Recovery in its Next Step operating system. We have also looked at other APIs and found that Color Recovery will generally work as long as we can change the supplied dithering algorithm.

Implementation

We implemented Color Recovery based on the assumption that it would be most useful in entry-level graphics products. We define these as products providing storage for only 8 bits of color per pixel in the frame buffer. These products that benefit most from Color Recovery must carefully control product costs. Therefore, the implementation effort was driven by a strong sense of cost versus end user benefit.

Dither/filter region shape

As mentioned earlier, the dither region in the implementation uses 32 pixels organized to cover a 2×16 region. The optimum shape would be closer to square, such as 4×8. However, the filter circuit needs storage for the pixels within the region. A 2×16 circuit requires that the current scan line's pixel and the data for the scan line above be available (see Figure 5). Note that in the current implementation, storage for a scan line of data requires about as much circuitry as the rest of the filtering logic. Therefore, using a 4×8 region would



6 Block diagram of a cell in the inner loop logic of Figure 5.

require three scan-line buffers, approximately doubling the cost of the Color Recovery logic. Although the filter shape was chosen because of cost, there is a trade-off. Most of the artifacts detectable in the current implementation result from the asymmetric filter shape.

Filter function logic

As explained earlier, the filter function averages the data within a region by summing the data for the pixels that pass an edge comparison. The sum is then divided by the number of pixels that pass the edge comparison. Typically, building such a circuit is difficult and costly because of the video clock speed (135 MHz or greater) required. Therefore, the implementation used a circuit that does not require a divide operation.

The implementation details of the filter function are complex. However, if we ignore the high-speed pipeline issues and some adjustments required to optimize image quality, then we can reduce the filter function to the following equation:

$$\sum_{i=0}^{K-1}\left(\begin{array}{l}((FB\_Data_i)(W_i)) + \\ ((Evaluated\_pixel)(\overline{W_i}))\end{array}\right) \quad \textbf{(1)}$$

where K = number of pixels in the filter, $W_i = 1$ when a pixel passes edge comparison, and $W_i = 0$ when a pixel fails edge comparison.

The idea is that if a pixel passes the edge comparison, we include it in the total. If a pixel fails the edge comparison, then we substitute the data for the pixel being evaluated for the failing pixel. The overriding assumption is that the pixel being evaluated reasonably approximates the actual true-color data. In the worst case, all the pixels around the sample fail the edge comparison and the dithered color is used for that location. Since dithering uses a reasonable sample at each location, this extreme case still results in a reasonable image.

To see how this works, let's look at two examples. In the first, assume that the pixel being evaluated is a single red dot specified using 01011000 binary (2.75 in our decimal numbering system). This is the same color used in some of the examples described earlier. However, this time assume that it is dithered to a value of 011. Also assume that this pixel is surrounded by green. Note that the edge comparison is done on a per-color channel basis, with the results from the three color channels being combined. Therefore, in this example all the pixels in the region except the pixel being evaluated will fail the edge comparison. In this case, we will add a red value of 011 thirty-two times. The results of the summation will be a red value of 01100000 (3.00 in our decimal numbering system). Although this is not exact, it will appear as a red dot in the middle of a green region—in other words, a reasonable approximation.

In the second example, assume a region filled with red specified to the same 8-bit binary value of 01011000. Also assume the simple dither described in the sidebar is used. In this case, 3/4 of the pixels will be stored as 011. The other 1/4 will be stored as 010. Since none of the pixels fail the edge comparison, we will sum 24 pixels with the value of 011 and eight with the value 010. The results of the summation will be a binary value of 01011000 (2.75 in our decimal system). In this region, the output of Color Recovery will exactly match the true-color data input.

Hardware details

The filtering logic, shown in a systems context in Figure 2, is expanded in Figure 5. As the frame buffer is scanned, each pixel in the display is sequentially sent to the logic shown in Figure 5. The left side of the figure shows the path taken as the data for each pixel is read from the frame buffer and then enters the filtering logic. The data is sent to a pipeline register for immediate use and to a scan line buffer for use when the next scan line is being evaluated. The 32 registers shown in Figure 5 store the data for the 2×16 region being evaluated. These registers are clocked at the pixel clock rate. Note that the data for each pixel on the display will pass through the location marked "X" (near the center of the figure). A pixel at location X is the pixel being evaluated. This means that the results of applying Equation 1 are assigned to the display at the screen address of X.

The 32 pixels stored in the pipeline registers shown in Figure 5 are sent through blocks of logic that perform the inner loop evaluation of Equation 1. This inner loop is essentially an edge detector. The logic shown in Figure 6 includes only pixels with values within ±1 of the pixel being evaluated in the summation. Note that the comparison is done per color channel with the results from the three color channels being combined. Thus the color from each of the three color channels must be within ±1 of the color of the pixel being evaluated to be included in the summation. The summation logic is simply an adder tree that sums the results of the pixels passing the edge comparison. The filter function is performed in parallel for all pixels within the filter region.

Given that the Color Recovery filtering circuit performs about 9 billion (9×10^9) operations per second, it

is surprisingly small. The entire filtering circuit used in the current implementation requires about 35,000 transistors. This is very small compared to the number of transistors required to increase the number of color planes. For example, increasing the number of color planes from eight to 16 on a typical SVGA display (Super VGA with 1,024 × 768 resolution) requires about 8,000,000 transistors, which is 1 Mbyte of additional frame buffer memory. Its small size thus makes the Color Recovery circuitry inexpensive enough to be included in low end, 8-bit-per-pixel graphics systems.

Visual results

Four types of image features cause the Color Recovery filter to change shape, which makes areas near these features more difficult to reconstruct. These problematic features include

1. edges,
2. steep color gradients,
3. areas of very low contrast, and
4. small features such as text and vectors.

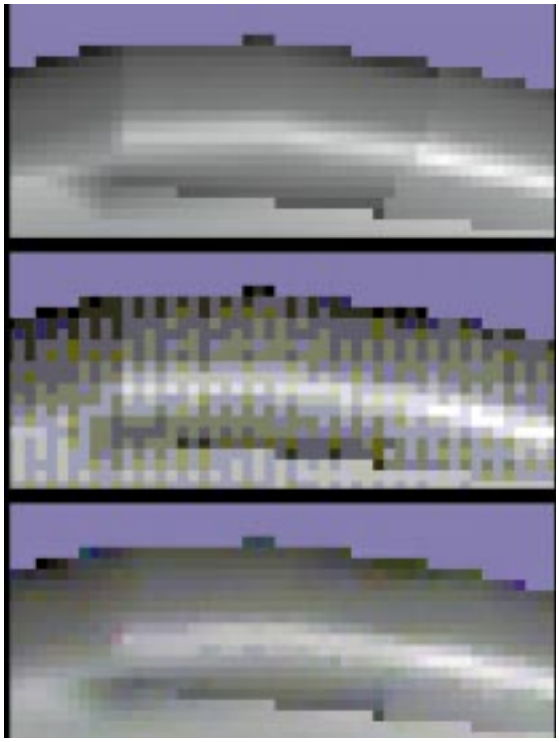
We can examine these problematic features with sets of three images. The top image in each set is always displayed in a 24-bit-per-pixel true-color mode. The middle image is displayed using a traditional 8-bit-per-pixel dither. The bottom image in each set is displayed using 8-bit-per-pixel Color Recovery as implemented on the HP HCRX-8.

We can examine the first three problem areas using a test pattern consisting of an image of an aliased jet. This test pattern typifies the type of image that would be rendered while running an interactive application. Figure 1 shows the entire test image; Figure 7 shows a close-up view of the area around the jet's cockpit. The contrast around the cockpit's edge is such that the Color Recovery edge detectors work very well; the color errors in this region are difficult for the human eye to discern. When displayed at normal screen resolution, color errors due to these edge types do not pose a major problem. However, Figure 7 also shows another problem type—that of steep color slopes. Note that in some places the highlights on the cockpit cause the edge detectors to detect an edge where one does not exist. Although visually less severe than dithering artifacts, it nevertheless can be seen in some images at normal screen resolution.

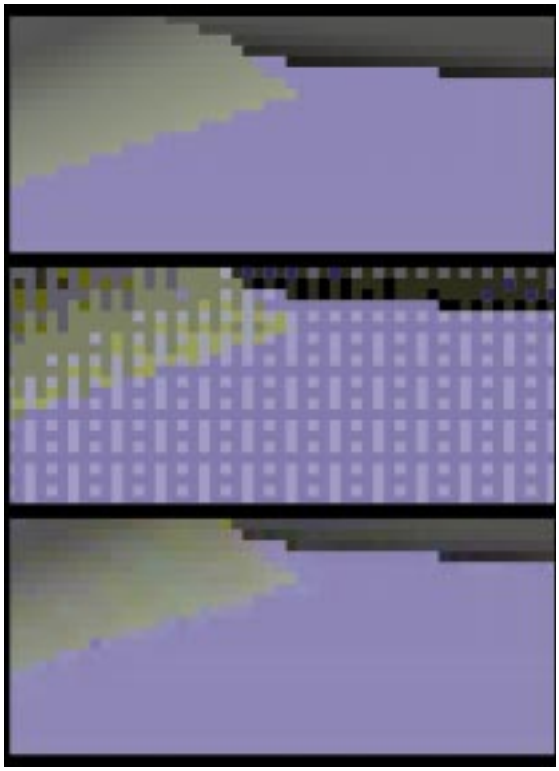
The third problem type, very low contrast, appears in Figure 8. This shows a close up view of where the back wing connects to the body of the jet. Here we see where the edge detectors miss the edge because some pixels from the background are dithered to the same color as the wing. When displayed at normal resolution, this is the most noticeable artifact in the image, as seen in Figure 1.

We examine the fourth problem type, small features, in Figure 9 (next page). On the left side we can see a small fan of aliased vectors. (Note that for this discussion text can be thought of as aliased line segments.) Since there is high contrast at the edges of the vectors, Color Recovery works well.

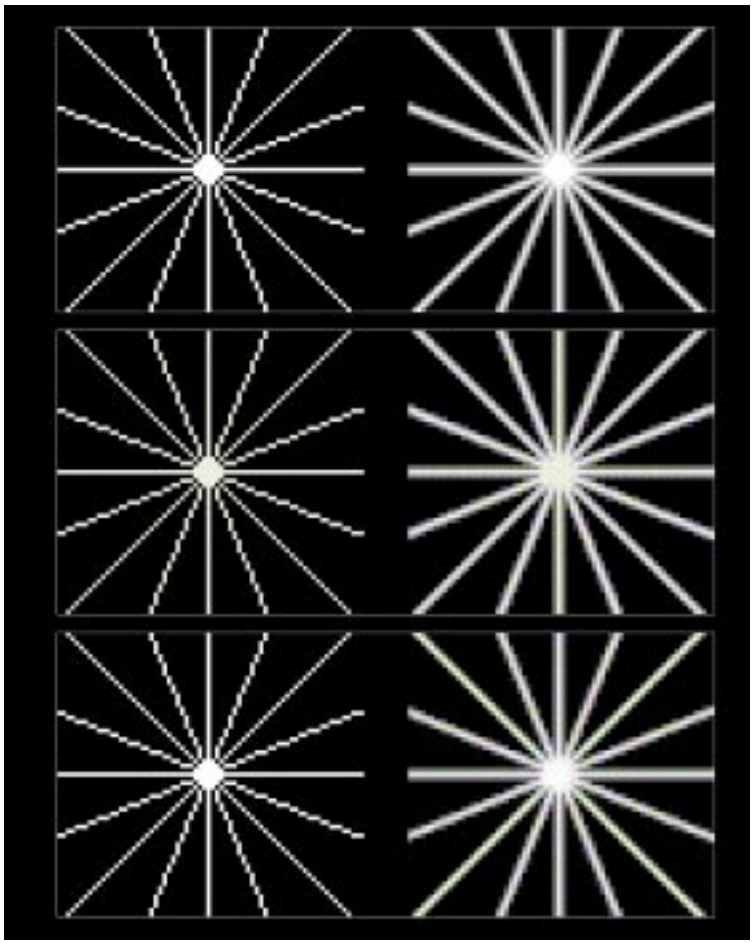
The right side of Figure 9 shows the special case of antialiased vectors. In this case, the color hue shifts slightly along the vectors due to the dithering selected for Color Recovery. Note that antialiased vectors do not cover rectangular regions such as polygons. Therefore the dither used along the vectors does not have a DC basis of 0. Adding to this problem, as part of the optimization for polygon regions, the Color Recovery dither



7 Zooming in on the cockpit region.



8 Zooming in on the wing region.



9 Test pattern of aliased and antialiased vectors.

uses a different table for each color channel. This means that each color component has a different DC offset along the vector. This results in the visual color shifting visible in the figure. However, we can alleviate this color shifting by using the same dither table for all three color channels. This makes the DC basis the same for the three color channels.

Therefore, instead of the color shifting to different hues along the vector, it will be displayed at a slightly different gray than desired. This difference in gray level is a less objectionable artifact. Another approach would be to use a vector-oriented dither, as Wells suggested.[8] We rejected these special case approaches for vectors because they place a substantial demand on the software development. (Currently software does not change the dither based on the primitive being rendered.) We also rejected a hardware solution because the current graphics hardware does not track the type of primitive that a pixel comes from when dither is applied. Adding this tracking logic did not seem appropriate for the entry-level systems targeted by this implementation.

Technology extensions

One issue not explored very thoroughly is use of a convolution filter to preprocess Color Recovery images. Simulations revealed that using a simple convolution filter to enhance edges was useful for video sequences. This edge enhancement before the frame-buffer dithering results in better edge detection in the filtering logic. However, we have not attempted to find an optimal convolution filter tuned to the Color Recovery process.

This article has focused on an implementation of Color Recovery that stores 8 bits per pixel and recovers 24 bits of color per pixel. This is certainly not the only possibility. We can create any combination of storage bits and recovered color bits. However, our primary problem remains edge detection. The fewer storage bits, the more difficult it is to recover the edges. The inverse is also true—the more storage bits, the better the edge detection.

For example, several simulations have been performed in which 12 bits of color per pixel are stored (4 bits for each color component). The images were then processed using Color Recovery to recover 24 bits per pixel. In this case the visual artifacts reduced to a point where only the most critical eye can discern the Color Recovery image from the original 24-bit image. In addition, a simulation using data stored in a 5-5-5 format and a 1×8 dither/filter region was made. Note that a 1×8 region does not require the scan line buffer shown in Figure 5. The results of this simulation were visually indistinguishable from the original 24-bit-per-pixel image.

In addition to increasing the number of bit planes, we tried several other edge detection improvements. Some of these were used in the HCRX-8, among them, combining the outputs of the edge detectors of the three color channels to produce a single edge-detected flag. Other edge detection methods such as using a more symmetric filter shape were not incorporated into the product. However, using all the edge detection methods I know of has allowed me to see stunning images stored as only 8 dithered bits per pixel. ■

Acknowledgments

Many people have helped me transform Color Recovery from an idea into a reality. The list is too long to print here; therefore, I hope everyone whose name would be on that list knows I appreciate their efforts. However, I must list several people by name. I will always be grateful to Paul Martin, Brian Miller, Randy Fiscus, and Dave McAllister for their extra efforts in this endeavor.

References

1. J. Foley et al., *Computer Graphics: Principles and Practice*, 2nd Ed. Addison-Wesley, 1990.

2. J. Leonard et al., "A 66-MHz DSP-Augmented RAMDAC for Smooth-Shaded Graphics Applications," *IEEE J. of Solid-State Circuits*, Vol. 26, No. 3, Mar. 1991, pp. 217-228.
3. W. Newman and R. Sproull, *Principles of Interactive Computer Graphics*, 2nd Ed., McGraw-Hill, New York, 1979.
4. S. Undy et al, "A VLSI Chip-set for Graphics and Multimedia Workstations," *IEEE Micro,* Vol. 14, No. 2, Apr. 1994, pp. 10-22.
5. R. Cook, "The Amiga: Is it Time You Took a Second Look?," *Computer Graphics World*, Vol. 16, No. 8, Aug. 1993, pp. 30-38
6. M. Analoui and J. Allebach, "New Results on Reconstruction of Continuous-Tone from Halftone," *IEEE 1992 Int'l Conf. on Accoustics, Speech, and Signal Processing* (Proc. ICASSP-92), Vol. 3, IEEE Press, Piscataway, N.J., 1992, pp. 313-316.
7. R. Ulichney, "Video Rendering," *Digital Technical J.*, Vol. 5, No. 2, Spring 1993, pp. 9-18.
8. S. Wells, G. Williamson, and S. Carrie, "Dithering for 12-Bit True-Color Graphics," *IEEE CG&A*, Vol. 11, No. 5, Sep. 1991, pp. 18-29.
9. T. Cornsweet, *Visual Perception*, Academic Press, London, 1970.

Anthony C. Barkans is a technical staff member at the Graphics Hardware Lab in Hewlett-Packard's Technical Computing R&D Center. Research interests include embedding unique image generation algorithms in hardware. He is part of the team that designed the Visualize family of graphics accelerators. Over the last several years he has been involved with the design of several generations of Hewlett-Packard graphics products. He received a BS in electrical engineering from the University of Utah.

Contact Barkans at Hewlett-Packard, 3404 Harmony Rd., Fort Collins, CO 80525.