Improvements for Color Dithering

Uwe Meyer-Gruhl, Carsten Steger
Forschungsgruppe Bildverstehen (FG BV)
Informatik IX, Technische Universität München
Orleansstr. 34, 81667 München, Germany
E-mail: {meyergru|stegerc}@informatik.tu-muenchen.de

Abstract

This paper discusses possible extensions of error-diffusion dithering algorithms for color devices that give a more accurate reproduction of colors than previous algorithms. The main difference of the method presented here and others is that usually color channels are regarded separately, whereas we look at them together as one color value that is used to select the closest output color. This alone results in more accurate color reproduction and less ink consumption. In addition, with the advent of better printing technology, error-diffusion dithering can be used to avoid loss of resolution in favor of more levels of gray or color.

Introduction

In the last few years computer peripherals that can support color, like ink-jet printers and high quality video boards, have become inexpensive. However, the software drivers used for such units often do not handle them in the best possible way.

PostScript, for example, has at least one big disadvantage as far as output on *CMYK* devices is concerned. This is one apparent reason why we could not illustrate this paper other than using enlarged samples. We fear that the printing is done with the very techniques criticized in this article, so that the difference between our approach and the usual one may not be seen.

Other methods[1] that try to render digital images on printers in an accurate manner suffer from the same deficiency, despite the fact that some of them use very elaborate schemes of calibration.

A large amount of literature on black-and-white halftoning exists,[2, 3, 4, 5, 6, 7] but the problem of printing on a color printer is rarely addressed. Most authors (notably Ulichney[2]) give the advice that color dithering should be done separately for each output channel. For printers that support the *CMYK*-color-model this procedure leads to the problems described in the next section.

The Problem

PostScript, as a typical example, takes the following steps when rendering a color image on a *CMYK* 4-bit device (if the colors have been specified in the *RGB* color space)[8]:

1. Convert $RGB \longrightarrow CMY$, i.e. $C = 1 - R$, $M = 1 - G$, $Y = 1 - B$.
2. Look for the minimum of C, M, and Y, and call it K.
3. Calculate the black generation $BG(k)$ and the undercolor removal $UCR(k)$, which both can be user-supplied functions that map $[0, 1] \mapsto [0, 1]$. Take $C' = C - UCR(K)$, $M' = M - UCR(K)$, $Y' = Y - UCR(K)$, and $K' = BG(K)$. The black portion is subtracted from all colors and applied directly. This is done to use *real* black instead of a *CMY* mixture, which visually turns out brownish or greenish. In most cases gray levels reproduced using a *CMY* mixture appear non-gray, to say the least.
4. Apply optional transfer functions to each C', M', Y', and K' channel.
5. Halftone the results in four discrete 1-bit device color space channels (*DeviceCMYK* in PostScript terminology) that consist of dot screens, which are usually rotated to angles of 15°, 75°, 0°, and 45°. The dot screens are rotated at different angles to get the best visual impression and to reduce moiré patterns. The screen for black is rotated 45° because the human visual system is least sensitive to lattices in this orientation (the least visible color screen, yellow, is not rotated). Because the cosines and sines of these angles are irrational numbers, the centers of the halftone-dots in one channel are distributed statistically with respect to any other channel.

Say, for example, we want to output a square which has $RGB = (0.5{,}0.5{,}0.0)$. This would be $CMY = (0.5{,}0.5{,}1.0)$. The black fraction in that color is 0.5, so that $CMYK = (0.0{,}0.0{,}0.5{,}0.5)$ results (provided that $BG(k) = k$ and $UCR(k) = k$ for simplicity).

Remember, we want a color that is 50% yellow and 50% black, a dark yellow. We could have asked PostScript for that color directly, had we used the *CMYK* color space in the first place.

PostScript now has two relevant dot screens (Y and K), in each of which 50% of all dots are set. Since these dot screens are rotated, the distribution is purely statistical. Thus we have four possible combinations:

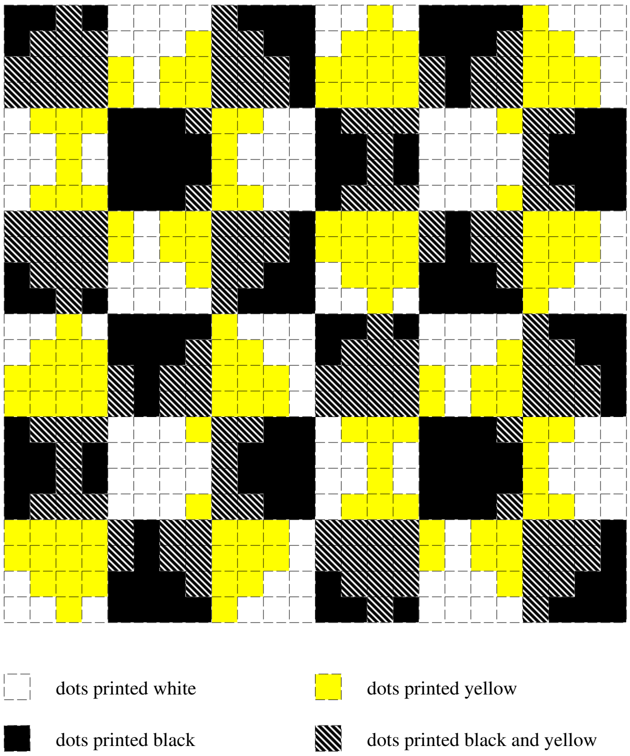


Figure 1: Clustered dot ordered dither of a color that has CMYK = (0,0,0.5,0.5) *with halftone screens at 45° (black) and 0° (yellow).*

Black	Yellow	Probability	Resulting color
0	0	25%	White
0	1	25%	Yellow
1	0	25%	Black
1	1	25%	Black (!)

Figure 1 shows this effect with a clustered-dot halftone-screen where the centers of the clustered dots are spaced 4 pixels apart. The relevant dither matrices were taken from Ulichney's book.[2] This example slightly simplifies the situation because PostScript goes to extraordinary lengths to ensure that the centers of the clustered dots are spaced exactly 4 dots apart. In figure 1 the centers of the dots of the screen at 0° (the screen for the yellow-component of the output) are slightly more than 4 dots apart (4.2426 dots) to simplify the presentation. Nevertheless this example shows the kind and magnitude of the error that occurs.

The resulting color now has *CMYK* = (0.0,0.0,0.25,0.5), which is much less saturation than desired. In addition to this, the use of dot screens actually limits the resolution in favor of more different levels of color.

Stone et al.[1] present an elaborate procedure to print color images that are as close as possible to the original images. The autors measure the color gamut of the monitors and the printers used. For the Cromalin printer gamut the authors note[1]: "This indicates that the most saturated colors are darker than those in the monitor gamut and less saturated that the monitor colors." For the thermal printer (a Panasonic EMCP500 thermal transfer printer with a resolution of 400 DPI) they write: "The gamut is slightly narrower, indicating that the colors overall are less saturated." Our analysis shows that a result like this was to be expected if printing is done with *independent* halftone screens.

Wrong Solutions

In the PostScript color model, there are two possible cures for this situation:

1. Use of other functions than identity for $BG(k)$ and $UCR(k)$, but this is far from perfect and does not remedy the situation when the colors were specified using the *CMYK* color space in the first place.

2. Modification of the transfer functions for C, M, Y, and K. However, (according to the PostScript manuals) it is not possible to use any other information than the color value for the channel itself in the transfer function. If we used a function like $\sqrt{k}$ for the black channel, for example, to make our test color brighter, the result would be a distorted gray response curve for "pure" grays. This holds likewise for emphasizing the *CMY* color channels.

Possible Alternatives

We chose to use a modified Floyd-Steinberg error diffusion dithering algorithm that is modeled after the error diffusion algorithm given by Ulichney.[2] The main reason for this was that no loss in resolution results. However, the modifications needed are not as straightforward as one might expect.

The simplest extension would be to use a 24-bit color space, and apply a FS-algorithm in the separate *RGB* or *CMYK* channels. This leads to the very problems described above.

The main insight is to notice that the channels cannot be handled separately. The choice is one of eight colors (cyan, magenta, yellow, red, green, blue, white, black) which can be reproduced by the printer. Actually, because colorspaces are inherently three-dimensional, only three of the four *CMYK* bits carry information ($2^3 = 8$). Thus, black is merely a linear combination of other colors in the *CMY* model.

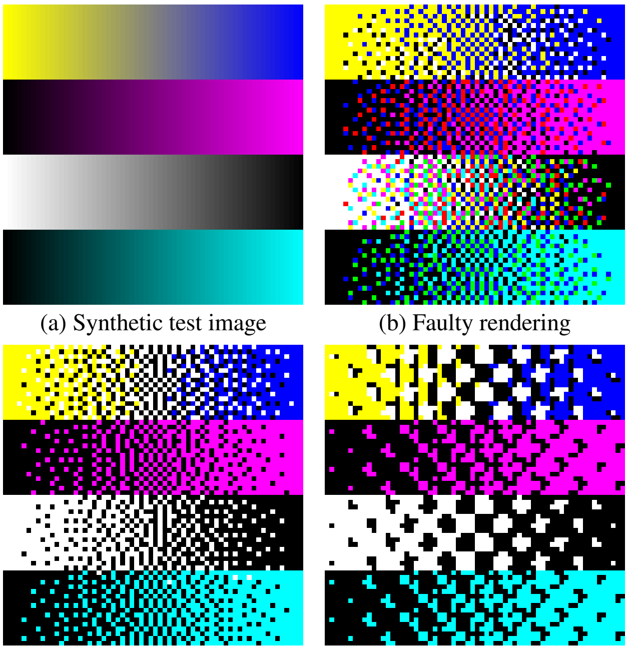
Some problems remain, even if the decision function looks like this:

```
typedef enum {
  BLACK = 0, BLUE = 1,
  GREEN = 2, CYAN = 3,
  RED = 4, MAGENTA = 5,
  YELLOW = 6, WHITE = 7
} PrinterColor;

PrinterColor Color(R,G,B)
{
  return (R > 0.5) << 2 +
      (G > 0.5) << 1 + (B > 0.5);
}
```

Statistically, all eight colors would be equally probable. Furthermore, color is wasted, since no undercolor removal is done, except for exact black (1/8).

Suppose there is a gray portion covering a part of the page, which has *RGB* = (0.5,0.5,0.5). If by chance (say a nearby red area or a random error) a red point *RGB* =



(a) Synthetic test image (b) Faulty rendering

(c) Improved rendering (d) Clustered-dot rendering

Figure 2: Synthetic test image and different renderings

(1.0,0.0,0.0) is used in a line, the next point *has* to be cyan $RGB = (0.0,1.0,1.0)$, to compensate for the error. This leads to another red point in the next step, which leads to a cyan one in the next and so on...

The same thing can happen if we start out with blue (then the color changes between blue and yellow) or with green (green/magenta). All those combinations should give the same visual result as a black/white change, i.e. a 50% gray area.

In fact, they do not, as everyone knows.[9] Moreover, these unwanted combinations use three times the color a black/white checkboard pattern does (consider two pixels, then a black/white pattern would use one dot per two pixels, whereas a red/cyan uses three, because red is magenta plus yellow). Furthermore, even more expensive colors (black tends to be cheaper) are used.

This problem does not only apply to gray levels, but also to "real" colors. One can end up in mixing blue and green (both use two dots/pixel) to achieve a 50% cyan (which uses only half a dot on average)!

To illustrate the erroneous behaviour of this approach, we have included a synthetic test image in figure 2(a) and a rather coarse rendering of it in figure 2(b). The algorithm used is the blue noise error diffusion scheme[2] applied to each channel separately. Thresholding is done in the *CMY* color space and the error is distributed in the *RGB* color space. The error buffer is initialized with zeroes. Note that the gray ramp is rendered using every color available to the printer. In the second ramp red and blue are used erroneously, while in the fourth ramp green and blue are used wrongly. Each of these colors is printed by using two colors, so a lot of ink is wasted in the second and fourth ramp, and even more in the third ramp.

Viable Solutions

One can, however, do the undercolor removal in the *last* step of the algorithm. When the desired color (in our example $RGB = (0.5,0.5,0.5)$) is examined more closely, it turns out that the non-colored portion in it is far greater than the colored portion. So, if in the FS-algorithm the desired color after calculating the previous errors is $RGB = (0.6,0.4,0.4)$, then the black part is 0.4, whereas the red part is only 0.2. In that case, one might better pick black than the color in question. Note, however, that the *essential* aspect of our approach is to regard the color channels as one color value, not separate signals that can be dithered independently. This leads to:

```
PrinterColor Color(R,G,B)
{
  C = 1.0 - R;
  M = 1.0 - G;
  Y = 1.0 - B;
  K = min(C,M,Y);
  C = C - K;
  M = M - K;
  Y = Y - K;
  if (max(C,M,Y,K) == K)
    return (K > 0.5 ? BLACK : WHITE);
  else
    return (R > 0.5) << 2 +
        (G > 0.5) << 1 + (B > 0.5);
}
```

In our approach an implicit assumption is made: that the dots for all channels are in the same spot, which might not be true for offset printing with color separations, but it seems to be true for most medium-resolution ink-jet printers like the Hewlett-Packard DeskJet series. The model we have tested was the 550C, and strangely enough, the manufacturer has decided to remove support for the *CMYK* color model in their newest product, the 1200C. Now only *CMY* is used, but resulting "black" is replaced by "real" black. Printer drivers that do not take this into account will produce the errors mentioned above, including excessive ink usage.

Tests have shown that with our scheme, the use of *CMY* can be drastically reduced (each color from 37.5% to 25%). Some more black is used (25% instead of 12.5%) instead for color images. The figures above refer to statistically random images, first dithered in *CMY*, replacing black with *K*, which results in 12.5% black pixels (and 37.5% *C*, *M*, and *Y*), if a uniform and independent distribution with 50% in all *CMY* channels is used. Then our approach was tested, which results in 25% for each channel.

The visual impression is even better, since the greenish color in black parts of the printed images disappears completely. Figure 2(c) shows the results of our new approach. Here, the algorithm is the same as above, including the initialization of the error buffer to all zeroes, except for the different selection logic in the thresholding part. In the first ramp no colors other than the required colors (i.e. yellow, black, white, and blue) are used. The same holds for the second and third ramp. Only the required colors are used. In the fourth ramp four dots are printed white instead of black or cyan. This means that a total of four dots are printed in wrong colors, which equals $\approx 0.1\%$ of all dots printed.

Sadly, this approach can not be applied to PostScript directly in form of a file to be downloaded to the PostScript interpreter, since the PostScript system does not allow this. Changes have to be made in the driver stage itself. We have done this for Display PostScript on a NeXT computer, using the 24-bit *RGB* color model and then dithering for *CMYK*. Additionally, a printer driver for a HP DeskJet 550C was developed for the GhostScript interpreter, and is in use within our department. However, Tektronix has recently introduced a new printer, the Phaser 340, that has an original Adobe PostScript interpreter, but seemingly uses an error diffusion algorithm for output. We were not able to find out whether this was made possible by new features provided by PostScript or by modifications done by Tektronix in the driver stage.

Improvements and Future Work

One could think of a way to even calibrate the output within the limitations of the output device and media: If one of the eight used colors turns out to be not quite as good as it should, it should suffice to just change the value used in the FS-algorithm to correct the error.

Suppose that the red color has a light touch of blue in it, then the value for a red point would be changed from (1.0,0.0,0.0) to, say, (1.0,0.0,0.1). The error-diffusion capabilities of the FS-algorithm would correct the error accordingly, although a "full" red could never be achieved.

One has to note that we made certain simplifications throughout this article. We did not mention, e.g., that a transfer function *must* be applied to all C, M, Y, and K channels in all real-world applications, because when a dot from an ink-jet printer is not ideally rectangular, but circular, and bigger than the cell it should occupy, a black-and-white checkboard pattern would result in a nearly black output. These transfer functions can be determined e.g. by a procedure similar to that given by Stone et al.[1]

Schemes that take the size of the dots printed into account and determine the area covered with each color may not work. Clapper and Yule[10] show that because of multiple internal reflections in the paper the intensity that results from halftoning depends in a nonlinear manner on the area that is covered by the paint. In an extreme case, a page whose area is covered by 50% black ink (as opposed to 50% black dots) can look as if the intensity was 85% black.

Because some printers can only print clustered dots (some laser printers still fall into this category), experiments were carried out to determine whether it is feasible to extend our scheme to clustered-dot printing. Figure 2(d) shows a result that was obtained by using a modified version of the smooth dot diffusion algorithm given by Knuth[7] that uses the color selection scheme above. This algorithm was chosen because it allows clustered-dot printing with an error diffusion algorithm. The example shows that no pixel is printed with an inappropriate color and that the dots are clustering rather well. Unfortunately, since this printing scheme roughly corresponds to printing with halftone screens that are all oriented in a 45°-direction, it can happen that the above mentioned moiré patterns occur. We do not know at the moment how this effect can be avoided, since by using other class matrices for each channel, we would lose the ability to process every channel of every pixel at the same instant, and therefore the ability to consider each color at every pixel as a whole. But, fortunately, for most pictures the patterns do not seem to be noticeable enough to disturb the viewer. Nevertheless, further research has to be carried out to obtain a perfectly working algorithm for clustered-dot dithering for color printers. However, we will not pursue these thoughts further because with the advent of better printing technology, error-diffusion dithering can be used on almost any printer. Arguments like Ulichney's[2] that clustered-dot ordered dither has to be used with laser printers because of their inability to print single pixels will probably not hold much longer.

References

1. Maureen C. Stone, William B. Cowan, John C. Beatty, Color Gamut Mapping and the Printing of Digital Color Images, *ACM Transactions on Graphics*, Vol. 7, No. 4, October 1988, pp. 249–292.
2. Robert Ulichney, *Digital Halftoning*, MIT Press, Cambridge, Massachusetts, USA, 1987.
3. J. Sullivan, R. Miller, G. Pios, Image Halftoning using a visual model in error diffusion, *Journal of the Optical Society of America A*, Vol. 10, No. 8, August 1993, pp. 1714–1724.
4. Theophano Mitsa, Kevin J. Parker, Digital halftoning technique using a blue-noise mask, *Journal of the Optical Society of America A*, Vol. 9, No. 11, November 1992, pp. 1920–1929.
5. J. Sullivan, L. Ray, R. Miller, Design of Minimum Visual Modulation Halftone Patterns, *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 21, No. 1, January/February 1991, pp. 33–38.
6. Reiner Eschbach, Keith T. Knox, Error-diffusion algorithm with edge enhancement, *Journal of the Optical Society of America A*, Vol. 8, No. 12, December 1991, pp. 1844–1850.
7. Donald E. Knuth, Digital Halftones by Dot Diffusion, *ACM Transactions on Graphics*, Vol. 6, No. 4, October 1987, pp. 245–273.
8. Adobe Systems Inc., *PostScript Language Reference Manual, Second Edition*, Addison Wesley, Reading, Massachusetts, USA, 1990.
9. Michael G. Lamming, Warren L. Rhodes, A Simple Method for Improved Color Printing of Monitor Images, *ACM Transactions on Graphics*, Vol. 9, No. 4, October 1990, pp. 345–375.
10. F. R. Clapper, J. A. C. Yule, The Effect of Multiple Internal Reflections on the Densities of Half-tone Prints on Paper, *Journal of the Optical Society of America*, Vol. 43, No. 7, July 1953, pp. 600–603.