DHALFTXT

June 20, 1991

Original name: DITHER.TXT
Original date: January 2, 1989

==========

ORIGINAL FOREWORD BY LEE CROCKER

What follows is everything you ever wanted to know (for the time being) about digital halftoning, or dithering. I'm sure it will be out of date as soon as it is released, but it does serve to collect data from a wide variety of sources into a single document, and should save you considerable searching time.

Numbers in brackets (e.g. [4] or [12]) are references. A list of these works appears at the end of this document.

Because this document describes ideas and algorithms which are constantly changing, I expect that it may have many editions, additions, and corrections before it gets to you. I will list my name below as original author, but I do not wish to deter others from adding their own thoughts and discoveries. This is not copyrighted in any way, and was created solely for the purpose of organizing my own knowledge on the subject, and sharing this with others. Please distribute it to anyone who might be interested.

If you add anything to this document, please feel free to include your name below as a contributor or as a reference. I would particularly like to see additions to the "Other books of interest" section. Please keep the text in this simple format: no margins, no pagination, no lines longer than 79 characters, and no non-ASCII or non-printing characters other than a CR/LF pair at the end of each line. It is intended that this be read on as many different machines as possible.

Original Author:

Lee Daniel Crocker [73407,2030]

Contributors:

Paul Boulay [72117,446]

Mike Morra [76703,4051]

==========

COMMENTS BY MIKE MORRA

I first entered the world of imaging in the fall of 1990 when my employer, Epson America Inc., began shipping the ES-300C color flatbed scanner. Suddenly, here I was, a field systems analyst who had worked almost exclusively with printers and PCs, thrust into a new and arcane world of look-up tables and dithering and color reduction and .GIF files! I realized right away that I had a lot of catching up to do (and it needed to be done quickly), so I began to frequent the CompuServe Information Service's Graphics Support Forum on a very regular basis.

Lee Crocker's excellent paper called DITHER.TXT was one of the first pieces of information that I came across, and it went a very long way toward answering a lot of questions that I'd had about the subject of dithering. It also provided me with the names of other essential reference works upon which Lee had based his paper, and I immediately began an eager search for these other references.

In the course of my self-study, however, I found that DITHER.TXT does presume the reader's familiarity with some fundamental imaging concepts, which meant that I needed to do a little "cramming." I get the impression that Lee was directing his paper more toward graphics programmers than to complete neophytes like me. I decided that I would rewrite and append to DITHER.TXT and try to incorporate some of the more elementary information that I'd absorbed along the way. In doing so, I hope that it will make it even more comprehensive, and thus even more useful to first-time users.

I elected to rename the revised file and chose the name DHALF.TXT in homage to the term "digital halftoning," as used in Robert Ulichney's splendid reference work. Notwithstanding, this paper is still very much Lee's original work, and I certainly do not propose that I have created something new and original here. It is also quite possible that in changing the presentation of some of the material therein, I may have unwittingly corrupted Lee's original intent and delivery, and this was also not my intention.

Accordingly, I've submitted this paper to the Graphics Support Forum as a draft work only, at least for the time being. Quite honestly, I don't know whether it would be appropriate as a replacement to DITHER.TXT, or as a second, distinct document. Too, I may very well have misconstrued or misinterpreted some factual information in my revision. As such, I welcome criticism and comment from all the original authors and contributors, and

any readers, with the hope that their feedback will help me to address these issues.

If this revision it is received favorably, I will submit it to the public domain; if it is met with brickbats (for whatever reason), I will withdraw it. Whatever the outcome, though, it will at least represent a very rewarding learning experience on my part!

With the unselfish help of many of the denizens of the Graphics Support Forum, I was ultimately able to thrash out (in my own mind) the answers to my questions that I needed. I'd like to publicly thank the whole Forum community in general for putting up with my unending barrage of questions and inquiries over the past few months <g>. In particular, I would thank John Swenson, Chris Young, and (of course) Lee Crocker for their invaluable assistance.

Mike Morra [76703,4051]
June 20, 1991

==========

What is Digital Halftoning?

Throughout much of the course of computer imaging technology, experimenters and users have been challenged with attempting to acceptably render digitized images on display devices which were incapable of reproducing the full spectrum of intensities or colors present in the source image. The challenge is even more pronounced in today's world of personal computing because of the technology gap between image generation and image rendering equipment.

Today, we now have affordable 24-bit image scanners which can generate nearly true-to-life scans having as many as 256 shades of gray, or in excess of 16.7 million colors. Mainstream display technology, however, still lags behind with 16- and 256-color VGA/SVGA video monitors and printers with binary (black/white) "marking engines" as the norm. Without specialized techniques for color reduction -- the process of finding the "best fit" of the display device's available gray shades and/or colors -- the imaging experimenter would be plagued with blotchy, noisy, off-color images.

(As of this writing, "true color" 24-bit video display devices, capable of reproducing all of the color/intensity information in the source image, are now beginning to migrate downward into the PC environment, but they exact a premium in cost and processor power which many users are loathe to pay. So-called "high-color" video displays -- typically 16-bit, with 32,768-color capability -- are moving into the mainstream, but color reduction techniques would still be required with these devices.)

The science of digital halftoning (more commonly referred to as dithering, or spatial dithering) is one of the techniques used to achieve satisfactory image rendering and color reduction. Initially, it was principally associated with the rendering of continuous-tone (grayscale) images on "binary" (i.e. 1-bit) video displays which could only display full black or full white pixels, or on printers which could produce only full black spots on a printed page. Indeed, Ulichney [3] gives a definition of digital halftoning as "... any algorithmic process which creates the illusion of continuous-tone images from the judicious arrangement of binary picture elements."

Ulichney's study, as well as the earlier literature on the subject (and this paper itself), discusses the process mostly in this context. Since we in the PC world are still saddled primarily with black/white marking engines in our hardcopy devices, this binary interpretation of digital halftoning is still very pertinent. However, as we will see later in this discussion, the concept can also be extended to include display devices (typically video monitors) which support limited grayscale or color palettes. Accordingly, we can broaden the traditional definition of digital halftoning to refer to rendering an image on any display device which is unable to show the entire range of colors or gray shades that are contained in the source image.

==========

Intensity/Color Resolution

The concept of resolution is essential to the understanding of digital halftoning. Resolution can be defined as "fineness" and is used to describe the level of detail in a digitally sampled signal.

Typically, when we hear the term "resolution" applied to images, we think of what's known as "spatial resolution," which is the basic sampling rate for the image. It describes the fineness of the "dots" (pixels or ink/toner spots) which comprise the image, i.e. how many of them are present along each horizontal and vertical inch. However, we can also speak of "intensity resolution" or "color resolution," which describes the fineness of detail available at each spot, i.e. the number of different gray shades or colors in the image. (I will go back and forth between the two terms depending on the type of image being discussed, but the reader should be aware that the concepts are analogous to each other.)

As you might expect, the higher the resolution of a digital sample, the better it can reproduce high frequency detail in the particular domain described by that resolution. A VGA display, for example, has a relatively good spatial resolution of 640 x 480 and a relatively poor color resolution of 8 bits (256 colors). By comparison, an NTSC color television receiver has a spatial resolution of approximately 350 x 525 and an excellent, nearly infinite color resolution. Thus, images rendered on a VGA screen will be

quite sharp, but rather blotchy in color. The same image displayed on the television receiver will not be as crisp, but will have much more accurate color rendition.

It is often possible to "trade" one kind of resolution for another. If your display device has a higher spatial resolution than the image you are trying to reproduce, it can show a very good image even if its color resolution is less. This is what most of us know as "dithering" and is the subject of this paper. (The other tradeoff, i.e., trading color resolution for spatial resolution, is called "anti-aliasing," and is not discussed here.)

For the following discussions I will assume that we are given a grayscale image with 256 shades of gray, which are assigned intensity values from 0 (black) through 255 (white), and that we are trying to reproduce it on a black and white output device, e.g. something like an Epson impact dotmatrix printer, or an HP LaserJet laser printer. Most of these methods can be extended in obvious ways to deal with displays that have more than two levels (but still fewer than the source image), or to color images. Where such extension is not obvious, or where better results can be obtained, I will go into more detail.

==========

Fixed Thresholding

A good place to start is with the example of performing a simple (or fixed) thresholding operation on our grayscale image in order to display it on our black and white device. This is accomplished by establishing a demarcation point, or threshold, at the 50% gray level. Each dot of the source image is compared against this threshold value: if it is darker than the value, the device plots it black, and if it's lighter, the device plots it white.

What happens to the image during this operation? Well, some detail survives, but our perception of gray levels is completely gone. This means that a lot of the image content is obliterated. Take an area of the image which is made up of various gray shades in the range of 60-90%. After fixed thresholding, all of those shades (being darker than the 50% gray threshold) will be mapped to solid black. So much for variations of intensity.

Another portion of the image might show an object with an increasing, diffused shadow across one of its surfaces, with gray shades in the range of 20-70%. This gradual variation in intensity will be lost in fixed thresholding, giving way to two separate areas (one white, one black) and a distinct, visible boundary between them. The situation where a transition from one intensity or shade to another is very conspicuous is known as contouring.

==========

Artifacts

Phenomena like contouring, which are not present in the source image but produced by the digital signal processing, are called artifacts. The most common type of artifact is the Moire' pattern. If you display or print an image of several lines, closely spaced and radiating from a single point, you will see what appear to be flower-like patterns. These are not part of the original image but are an illusion produced by the jaggedness of the display. We will encounter and discuss other forms of artifacts later in this paper.

==========

Error Noise

Returning to our fixed-thresholded (and badly-rendered) image, how could we document what has taken place to make this image so inaccurate? Expressing it in technical terms, a relatively large amount of error "noise" is present in the fixed-thresholded image. The error value is the difference between the image's original intensity at a given dot and the intensity of the displayed dot. Obviously, very dark values like 1 or 2 (which are almost full black) incur very small errors when they are rendered as a 0 value (black) dot. On the other hand, a gross error is incurred when a 129 value dot (a medium gray) is displayed at 255 value (white), for instance.

Simply put, digital halftoning redistributes this "noise energy" in a way which makes it less visible. This brings up an important concept: digital halftoning does not INCREASE the noise energy. In some of the literature, reference is made to the "addition of dither noise," which might give this impression. This is not the case, however: effective digital halftoning acts upon the low-frequency component of the error noise (the component which contributes to graininess) and scatters it in higher-frequency components where it is not as obvious.

==========

Classes of digital halftoning algorithms

The algorithms we will discuss in this paper can be subdivided into four categories:

1. Random dither

2. Patterning
3. Ordered dither
4. Error-diffusion halftoning

Each of these methods is generally better than those listed before it, but other considerations such as processing time, memory constraints, etc. may weigh in favor of one of the simpler methods.

To convert any of the first three methods into color, simply apply the algorithm separately for each primary color and mix the resulting values. This assumes that you have at least eight output colors: black, red, green, blue, cyan, magenta, yellow, and white. Though this will work for error diffusion as well, there are better methods which will be discussed in more detail later.

==========

Random dither

Random dithering could be termed the "bubblesort" of digital halftoning algorithms. It was the first attempt (documented as far back as 1951) to correct the contouring produced by fixed thresholding, and it has traditionally been referenced for comparison in most studies of digital halftoning. In fact, the name "ordered dither" (which will be discussed later) was chosen to contrast random dither.

While it is not really acceptable as a production method, it is very simple to describe and implement. For each dot in our grayscale image, we generate a random number in the range 0 - 255: if the random number is greater than the image value at that dot, the display device plots the dot white; otherwise, it plots it black. That's it.

This generates a picture with a lot of "white noise", which looks like TV picture "snow". Although inaccurate and grainy, the image is free from artifacts. Interestingly enough, this digital halftoning method is useful in reproducing very low-frequency images, where the absence of artifacts is more important than noise. For example, a whole screen containing a gradient of all levels from black to white would actually look best with a random dither. With this image, other digital halftoning algorithms would produce significant artifacts like diagonal patterns (in ordered dithering) and clustering (in error diffusion halftones).

I should mention, of course, that unless your computer has a hardware-based random number generator (and most don't), there may be some artifacts from the random number generation algorithm itself. For efficiency, you can take the random number generator "out of the loop" by generating a list of random numbers beforehand for use in the dither. Make sure that the list is larger than the number of dots in the image or you may get artifacts from the reuse

of numbers. The worst case would be if the size of your list of random numbers is a multiple or near-multiple of the horizontal size of the image; in this case, unwanted vertical or diagonal lines will appear.

As unattractive as it is, random dithering can actually be related to a pleasing, centuries-old art know as mezzotinting (the name itself is an Italianized derivative of the English "halftone"). In a mezzotint, the skilled craftsman worked a soft metal (usually copper) printing plate, and roughened or ground the dark regions of the image by hand and in a seemingly random fashion. Analyzing it in scientific terms (which would surely insult any mezzotinting artisan who might read this!) the pattern created is not very regular or periodic at all, but the absence of low frequency noise leads to a very attractive image without much graininess. A similar process is still in use today, in the form of modern gravure printing.

==========

"Classical" halftoning

Let's take a short departure from the digital domain and look at the traditional or "classical" printing technique of halftoning. This technique is over a century old, dating back to the weaving of silk pictures in the mid 1800's. Modern halftone printing was invented in the late 1800's, and halftones of that period are even today considered to be attractive renditions of their subjects.

Essentially, halftoning involves the printing of dots of different sizes in an ordered and closely spaced pattern in order to simulate various intensities. The early halftoning artisans realized that when we view a very small area at normal viewing distances, our eyes perform a blending or smoothing function on the fine detail within that area. As a result, we perceive only the overall intensity of the area. This is known as spatial integration.

Although the tools of halftoning (the "screens" and screening process used to generate the varying dots of the printed image) have undergone improvements throughout the years, the fundamental principles remain unchanged. This includes the 45-degree "screen angle" of the lines of dots, which was known even to the earliest halftone artisans as giving more pleasing images than dot lines running horizontally and vertically.

==========

Patterning

This was the first digital technique to pay homage to the classical

halftone. It takes advantage of the fact that the spatial resolution of display devices had improved to the point where one could trade some of it for better intensity resolution. Like random dither, it is also a simple concept, but is much more effective.

For each possible value in the image, we create and display a pattern of pixels (which can be either video pixels or printer "spots") that approximates that value. Remembering the concept of spatial integration, if we choose the appropriate patterns we can simulate the appearance of various intensity levels -- even though our display can only generate a limited set of intensities.

For example, consider a 3 x 3 pattern. It can have one of 512 different arrangements of pixels: however, in terms of intensity, not all of them are unique. Since the number of black pixels in the pattern determines the darkness of the pattern, we really have only 10 discrete intensity patterns (including the all-white pattern), each one having one more black pixel than the previous one.

But which 10 patterns? Well, we can eliminate, right off the bat, patterns like:

```
---        X--        --X        X--
XXX   or   -X-   or   -X-   or   X--
---        --X        X--        X--
```

because if they were repeated over a large area (a common occurrence in many images [1]) they would create vertical, horizontal, or diagonal lines. Also, studies [1] have shown that the patterns should form a "growth sequence:" once a pixel is intensified for a particular value, it should remain intensified for all subsequent values. In this fashion, each pattern is a superset of the previous one; this similarity between adjacent intensity patterns minimizes any contouring artifacts.

Here is a good pattern for a 3-by-3 matrix which subscribes to the rules set forth above:

```
---   ---   ---   -X-   -XX   -XX   -XX   -XX   XXX   XXX
---   -X-   -XX   -XX   -XX   -XX   XXX   XXX   XXX   XXX
---   ---   ---   ---   ---   -X-   -X-   XX-   XX-   XXX
```

This pattern matrix effectively simulates a screened halftone with dots of various sizes. In large areas of constant value, the repetitive pattern formed will be mostly artifact-free.

No doubt, the reader will realize that applying this patterning process to our image will triple its size in each direction. Because of this,

patterning can only be used where the display's spatial resolution is much greater than that of the image.

Another limitation of patterning is that the effective spatial resolution is decreased, since a multiple-pixel "cell" is used to simulate the single, larger halftone dot. The more intensity resolution we want, the larger the halftone cell used and, by extension, the lower the spatial resolution.

In the above example, using 3 x 3 patterning, we are able to simulate 10 intensity levels (not a very good rendering) but we must reduce the spatial resolution to 1/3 of the original figure. To get 64 intensity levels (a very acceptable rendering), we would have to go to an 8 x 8 pattern and an eight-fold decrease in spatial resolution. And to get the full 256 levels of intensity in our source image, we would need a 16 x 16 pattern and would incur a 16-fold reduction in spatial resolution. Because of this size distortion of the image, and with the development of more effective digital halftoning methods, patterning is only infrequently used today.

To extend this method to color images, we would use patterns of colored pixels to represent shades not directly printable by the hardware. For example, if your hardware is capable of printing only red, green, blue, and black (the minimal case for color dithering), other colors can be represented with 2 x 2 patterns of these four:

```
    Yellow = R G    Cyan = G B      Magenta = R B       Gray = R G
             G R           B G                B R              B K
```

(B here represents blue, K is black). In this particular example, there are a total of 31 such distinct patterns which can be used; their enumeration is left "as an exercise for the reader" (don't you hate books that do that?).

==========

Clustered vs. dispersed patterns

The pattern diagrammed above is called a "clustered" pattern, so called because as new pixels are intensified in each pattern, they are placed adjacent to the already-intensified pixels. Clustered-dot patterns were used on many of the early display devices which could not render individual pixels very distinctly, e.g. printing presses or other printers which smear the printed spots slightly (a condition known as dot gain), or video monitors which introduce some blurriness to the pixels. Clustered-dot groupings tend to hide the effect of dot gain, but also produce a somewhat grainy image.

As video and hardcopy display technology improved, newer devices (such as

electrophotographic laser printers and high-res video displays) were better able to accurately place and size their pixels. Further research showed that, especially with larger patterns, the dispersed (non-clustered) layout was more pleasing. Here is one such pattern:

```
---   X--   X--   X--   X-X   X-X   X-X   XXX   XXX   XXX
---   ---   ---   --X   --X   X-X   X-X   X-X   XXX   XXX
---   ---   -X-   -X-   -X-   -X-   XX-   XX-   XX-   XXX
```

Since clustering is not used, dispersed-dot patterns produce less grainy images.

==========

Ordered dither

While patterning was an important step toward the digital reproduction of the classic halftone, its main shortcoming was the spatial enlargement (and corresponding reduction in resolution) of the image. Ordered dither represents a major improvement in digital halftoning where this spatial distortion was eliminated and the image could then be rendered in its original size.

Obviously, in order to accomplish this, each dot in the source image must be mapped to a pixel on the display device on a one-to-one basis. Accordingly, the patterning concept was redefined so that instead of plotting the whole pattern for each image dot, THE IMAGE DOT IS MAPPED ONLY TO ONE PIXEL IN THE PATTERN. Returning to our example of a 3 x 3 pattern, this means that we would be mapping NINE image dots into this pattern.

The simplest way to do this in programming is to map the X and Y coordinates of each image dot into the pixel (X mod 3, Y mod 3) in the pattern.

Returning to our two patterns (clustered and dispersed) as defined earlier, we can derive an effective mathematical algorithm that can be used to plot the correct pixel patterns. Because each of the patterns above is a superset of the previous, we can express the patterns in a compact array form as the order of pixels added:

```
8  3  4                    1  7  4
6  1  2        and         5  8  3
7  5  9                    6  2  9
```

Then we can simply use the value in the array as a threshold. If the value of the original image dot (scaled into the 0-9 range) is less than the number in the corresponding cell of the matrix, we plot that pixel black; otherwise, we plot it white. Note that in large areas of constant value, we will get repetitions of the pattern just as we did with patterning.

As before, clustered patterns should be used for those display devices which blur the pixels. In fact, the clustered-dot ordered dither is the process used by most newspapers, and in the computer imaging world the term "halftoning" has come to refer to this method if not otherwise qualified.

As noted earlier, the dispersed-dot method (where the display hardware allows) is preferred in order to decrease the graininess of the displayed images. Bayer [2] has shown that for matrices of orders which are powers of two there is an optimal pattern of dispersed dots which results in the pattern noise being as high-frequency as possible. The pattern for a 2x2 and 4x4 matrices are as follows:

```
1  3        1  9  3 11        These patterns (and their rotations
4  2       13  5 15  7        and reflections) are optimal for a
            4 12  2 10        dispersed-dot ordered dither.
           16  8 14  6
```

Ulichney [3] shows a recursive technique can be used to generate the larger patterns. (To fully reproduce our 256-level image, we would need to use an 8x8 pattern.)

The Bayer ordered dither is in very common use and is easily identified by the cross-hatch pattern artifacts it produces in the resulting display. This artifacting is the major drawback of an otherwise powerful and very fast technique.

==========

Dithering with "blue noise"

Up to this point in our discussion, we have (with the exception of dithering with white noise) discussed digital halftoning schemes which rely on the application of some fairly regular mathematical processes in order to redistribute the error noise of the image. Unfortunately, the regularity of these algorithms leads to different kinds of artifacting which detracts from the rendered image. In addition, these images all tend to reflect the display device's row-and-column dot pattern to some extent, and this further contributes to the "mechanical" character of the output image.

Dithering with white noise, on the other hand, introduces enough randomness to suppress the artifacting and the gridlike appearance, but the low-frequency component of this noise introduces graininess.

Obviously, what is needed is a method which falls somewhere in the middle of these two extremes. In theoretical terms, if we could take white noise and remove its low-frequency content, this would be an ideal way to disperse the error content of our image. Many of the digital halftoning developers, making an analogy to the audio world, refer to this concept as dithering with blue noise. (In audio theory, "pink noise," which is often used as a diagnostic and testing tool, is white noise from which some level of high-frequency content has been filtered.)

Alas, while an audio-frequency analog low-pass filter is a relatively simple device to construct and operate, implementing a digital high-pass filter in program code -- and one which operates efficiently enough so as not to degrade display response time -- is no trivial task.

==========

Error-diffusion halftoning

After considerable research, it was found that a set of techniques known as error diffusion (also termed error dispersion or error distribution) accomplished this quite effectively. In fact, error diffusion generates the best results of any of the digital halftoning methods described here. Much of the low-frequency noise component is suppressed, producing images with very little grain. Error-diffusion halftones also display a very pleasing randomness, without the visual sensation of rows and columns of dots; this effect is known as the "grid defiance illusion."

As in other areas of life, though, there ain't no such thing as a free lunch. Error diffusion is, by nature, the slowest method of digital halftoning. In fact, there are several variants of this technique, and the better they get, the slower they are. However, one will realize a very significant improvement in the quality of the processed images which easily justifies the time and computational power required.

Error diffusion is very simple to describe. For each point in our image, we first find the closest intensity (or color) available. We then calculate the difference between the image value at that point and that nearest available intensity/color: this difference is our error value. Now we divide up the error value and distribute it to some of the neighboring image areas which we have not visited (or processed) yet. When we get to these later dots, we add in the portions of error values which were distributed there from the preceding dots, and clip the cumulative value to an allowed range if needed. This new, modified value now becomes the image value that we use for processing this point.

If we are dithering our sample grayscale image for output to a black-and-white device, the "find closest intensity/color" operation is just a simple thresholding (the closest intensity is going to be either black or white). In color imaging -- for instance, color-reducing a 24-bit true color Targa file to an 8-bit, mapped GIF file -- this involves matching the input color to the closest available hardware color. Depending on how the display hardware manages its intensity/color palette, this matching process can be a difficult task. (This is covered in more detail in the "Color issues" section later in this paper.)

Up till now, all other methods of digital halftoning were point operations, where any adjustments that were made to a given dot had no effect on any of the surrounding dots. With error diffusion, we are doing a "neighborhood operation." Dispersing the error value over a larger area is the key to the success of these methods.

The different ways of dividing up the error can be expressed as patterns called filters. In the following sections, I will list a number of the most commonly-used filters and some info on each.

==========

The Floyd-Steinberg filter

This is where it all began, with Floyd and Steinberg's [4] pioneering research in 1975. The filter can be diagrammed thus:

```
          *   7
      3   5   1     (1/16)
```

In this (and all subsequent) filter diagrams, the "*" represents the pixel currently being scanning, and the neighboring numbers (called weights) represent the portion of the error distributed to the pixel in that position. The expression in parentheses is the divisor used to break up the error weights. In the Floyd-Steinberg filter, each pixel "communicates" with 4 "neighbors." The pixel immediately to the right gets 7/16 of the error value, the pixel directly below gets 5/16 of the error, and the diagonally adjacent pixels get 3/16 and 1/16.

The weighting shown is for the traditional left-to-right scanning of the image. If the line were scanned right-to-left (more about this later), this pattern would be reversed. In either case, the weights calculated for the subsequent line must be held by the program, usually in an array of some sort, until that line is visited later.

Floyd and Steinberg carefully chose this filter so that it would produce a checkerboard pattern in areas with intensity of 1/2 (or 128, in our sample image). It is also fairly easy to execute in programming code, since the division by 16 is accomplished by simple, fast bit-shifting instructions (this is the case whenever the divisor is a power of 2).

==========

The "false" Floyd-Steinberg filter

Occasionally, you will see the following filter erroneously called the Floyd-Steinberg filter:

```
          *   3
          3   2     (1/8)
```

The output from this filter is nowhere near as good as that from the real Floyd-Steinberg filter. There aren't enough weights to the dispersion, which means that the error value isn't distributed finely enough. With the entire image scanned left-to-right, the artifacting produced would be totally unacceptable.

Much better results would be obtained by using an alternating, or serpentine, raster scan: processing the first line left-to-right, the next line right-to-left, and so on (reversing the filter pattern appropriately). Serpentine scanning -- which can be used with any of the error-diffusion filters detailed here -- introduces an additional perturbation which contributes more randomness to the resultant halftone. Even with serpentine scanning, however, this filter would need additional perturbations (see below) to give acceptable results.

==========

The Jarvis, Judice, and Ninke filter

If the false Floyd-Steinberg filter fails because the error isn't distributed well enough, then it follows that a filter with a wider distribution would be better. This is exactly what Jarvis, Judice, and Ninke [6] did in 1976 with their filter:

```
              *   7   5
     3   5    7   5   3
     1   3    5   3   1   (1/48)
```

While producing nicer output than Floyd-Steinberg, this filter is much slower to implement. With the divisor of 48, we can no longer use bit-shifting to calculate the weights but must invoke actual DIV (divide) processor instructions. This is further exacerbated by the fact that the filter must communicate with 12 neighbors; three times as many in the Floyd-Steinberg filter. Furthermore, with the errors distributed over three lines, this means that the program must keep two forward error arrays, which requires extra memory and time for processing.

==========

The Stucki filter

P. Stucki [7] offered a rework of the Jarvis, Judice, and Ninke filter in 1981:

```
            *   8   4
    2   4   8   4   2
    1   2   4   2   1   (1/42)
```

Once again, division by 42 is quite slow to calculate (requiring DIVs). However, after the initial 8/42 is calculated, some time can be saved by producing the remaining fractions by shifts. The Stucki filter has been observed to give very clean, sharp output, which helps to offset the slow processing time.

==========

The Burkes filter

Daniel Burkes [5] of TerraVision undertook to improve upon the Stucki filter in 1988:

```
            *   8   4             The Burkes filter
    2   4   8   4   2   (1/32)
```

Notice that this is just a simplification of the Stucki filter with the bottom row removed. The main improvement is that the divisor is now 32, which allows the error values to be calculated using shifts once more, and the number of neighbors communicated with has been reduced to seven.

Furthermore, the removal of one row reduces the memory requirements of the filter by eliminating the second forward array which would otherwise be needed.

==========

The Sierra filters

In 1989, Frankie Sierra came out with his three-line filter:

```
            *   5   3                   The Sierra3 filter
    2   4   5   4   2
        2   3   2       (1/32)
```

A year later, Sierra followed up with a two-line modification:

```
            *   4   3                   The Sierra2 filter
    1   2   3   2   1   (1/16)
```

and a very simple "Filter Lite," as he calls it:

```
        *   2                           The Sierra-2-4A filter
    1   1               (1/4)
```

Even this very simple filter, according to Sierra, produces better results than the original Floyd-Steinberg filter.

==========

Miscellaneous filters

Many image processing software packages offer one or more of the filters listed above as dithering options. In nearly every case, the Floyd-Steinberg filter (or a variant thereof) is included. The Bayer ordered dither is sometimes offered, although the Floyd-Steinberg filter will do a better job in essentially the same processing time. Higher-quality filters like Burkes or Stucki are usually also present.

All of the filters described above are used on display devices which have "square pixels." This is to say that the display lays out the pixels in

rows and columns, aligned horizontally and vertically and spaced equally in both directions. This applies to the commonly-used video modes in VGA and SVGA: 640 x 480, 800 x 600, and 1024 x 768, with a 4:3 "aspect ratio." It would also include HP-compatible and PostScript desktop laser printers using 300dpi marking engines.

Some displays may use "rectangular pixels," where the horizontal and vertical spacings are unequal. This would include various EGA and CGA video modes and other specialized video displays, and most dot-matrix printers. In many cases, the filters described earlier will do a decent job on rectangular pixel grids, but an optimized filter would be preferred. Slinkman [10] describes one such filter for his 640 x 240 monochrome display with a 1:2 aspect ratio.

In other cases, video displays might use a "hexagonal grid" of pixels, where rows of pixels are offset or staggered, in much the same fashion used on broadcast television receivers. This is illustrated below:

Hexagonal grids are given a very thorough treatment by Ulichney, should you be interested in further information.

While technically not an error-diffusion filter, a method proposed by Gozum [11] offers color resolutions in excess of 256 colors by plotting red, green, and blue pixel "triplets" or triads to simulate an "interlaced" television display (sacrificing some horizontal resolution in the process). Again, I would refer interested readers to his document for more information.

==========

Special considerations

The speed disadvantages of the more complex filters can be eliminated somewhat by performing the divisions beforehand and using lookup tables instead of doing the math inside the loop. This makes it harder to use various filters in the same program, but the speed benefits are enormous.

It is critical with all of these algorithms that when error values are added to neighboring pixels, the resultant summed values must be truncated to fit

within the limits of hardware. Otherwise, an area of very intense color may cause streaks into an adjacent area of less intense color.

This truncation is known as "clipping," and is analogous to the audio world's concept of the same name. As in the case of an audio amplifier, clipping adds undesired noise to the data. Unlike the audio world, however, the visual clipping performed in error-diffusion halftoning is acceptable since it is not nearly so offensive as the color streaking that would occur otherwise. It is mainly for this reason that the larger filters work better -- they split the errors up more finely and produce less clipping noise.

With all of these filters, it is also important to ensure that the sum of the distributed error values is equal to the original error value. This is most easily accomplished by subtracting each fraction, as it is calculated, from the whole error value, and using the final remainder as the last fraction.

==========

Further perturbations

As alluded to earlier, there are various techniques for the reduction of digital artifacts, most of which involve using a little randomness to lightly "perturb" a regular algorithm (particularly the simpler ones). It could be said that random dither takes this concept to the extreme.

Serpentine scanning is one of these techniques, as noted earlier. Other techniques include the addition of small amounts of white noise, or randomizing the positions of the error weights (essentially, using a constantly-varying pattern). As you might imagine, any of these methods incur a penalty in processing time.

Indeed, some of the above filters (particularly the simpler ones) can be greatly improved by skewing the weights with a little randomness [3].

==========

Nearest available color

Calculating the nearest available intensity is trivial with a monochrome image; calculating the nearest available color in a color image requires more work.

A table of RGB values of all available colors must be scanned sequentially for each input pixel to find the closest. The "distance" formula most often used is a simple pythagorean "least squares". The difference for each color

is squared, and the three squares added to produce the distance value. This value is equivalent to the square of the distance between the points in RGB-space. It is not necessary to compute the square root of this value because we are not interested in the actual distance, only in which is smallest. The square root function is a monotonic increasing function and does not affect the order of its operands. If the total number of colors with which you are dealing is small, this part of the algorithm can be replaced by a lookup table as well.

When your hardware allows you to select the available colors, very good results can be achieved by selecting colors from the image itself. You must reserve at least 8 colors for the primaries, secondaries, black, and white for best results. If you do not know the colors in your image ahead of time, or if you are going to use the same map to dither several different images, you will have to fill your color map with a good range of colors. This can be done either by assigning a certain number of bits to each primary and computing all combinations, or by a smoother distribution as suggested by Heckbert [8].

An alternate method of color selection, based on a tetrahedral color space, has been proposed by Crawford [12]. His algorithm has been optimized for either dispersed-dot ordered dither or Floyd-Steinberg error diffusion with serpentine scan.

==========

Hardware halftoning

In some cases, image scanning hardware may be able to digitally halftone and dither the image "on the fly" as it is being scanned. The data produced by the "raw" scan is then already in a 1- or 2-bit/pixel format. While this feature would probably be unsuitable for cases where the image would need further processing (see the "Loss of image information" section below), it is very useful where the operator wants to generate a final image, ready for printing or displaying, with little or no subsequent processing.

As an example, the Epson ES-300C color scanner (and its European equivalent, the Epson GT-6000) offers three internal halftone modes. One is a standard "halftone" algorithm, i.e. a clustered-dot ordered dither. The other two are error-diffusion filters (one "sharp," the other "soft") which are proprietary Epson-developed filters.

==========

Loss of image information incurred by digital halftoning

It is important to emphasize here that digital halftoning is a ONE-WAY operation. Once an image has been halftoned or dithered, although it may look like a good reproduction of the original, INFORMATION IS PERMANENTLY LOST. Many image processing functions fail on dithered images; in fact, you would not want to dither an image which had already been dithered to some extent.

For these reasons, digital halftoning must be considered primarily as a way TO PRODUCE AN IMAGE ON HARDWARE THAT WOULD OTHERWISE BE INCAPABLE OF DISPLAYING IT. This would hold true wherever a grayscale or color image needs to be rendered on a bilevel display device. In this situation, one would almost never want to store the dithered image.

On the other hand, when color images are dithered for display on color displays with a lower color resolution, the dithered images are more useful. In fact, the bulk of today's scanned-image GIF files which abound on electronic BBSs and information services are 8-bit (256 color), color mapped and dithered files created from 24-bit true-color scans. Only rarely are the 24-bit files exchanged, because of the huge amount of data contained in them.

In some cases, these mapped GIF files may be further processed with special paint/processing utilities, with very respectable results. However, the previous warning still applies: one can never obtain the same image fidelity when operating on the mapped GIF file as they could if they were operating on the true-color image file.

Generally speaking, digital halftoning and dithering should be the last stage in producing a physical display from a digitally stored image. The data representing an image should always be kept in full detail in case you should want to reprocess it in any way. As affordable display technology improves, the day may soon come where you might possess the hardware to allow you to use all of the original image information without the need for digital halftoning or color reduction.

==========

Sample code

Despite my best efforts in expository writing, nothing explains an algorithm better than real code. With that in mind, presented here are a few programs which implement some of the concepts presented in this paper.

1) This code (in the C programming language) dithers a 256-level monochrome image onto a black-and-white display with the Bayer ordered dither.

```
/* Bayer-method ordered dither.  The array line[] contains the intensity
** values for the line being processed.  As you can see, the ordered
** dither is much simpler than the error dispersion dither.  It is also
** many times faster, but it is not as accurate and produces cross-hatch
** patterns on the output.
*/

unsigned char line[WIDTH];

int pattern[8][8] = {
    { 0, 32,  8, 40,  2, 34, 10, 42},   /* 8x8 Bayer ordered dithering  */
    {48, 16, 56, 24, 50, 18, 58, 26},   /* pattern.  Each input pixel   */
    {12, 44,  4, 36, 14, 46,  6, 38},   /* is scaled to the 0..63 range */
    {60, 28, 52, 20, 62, 30, 54, 22},   /* before looking in this table */
    { 3, 35, 11, 43,  1, 33,  9, 41},   /* to determine the action.     */
    {51, 19, 59, 27, 49, 17, 57, 25},
    {15, 47,  7, 39, 13, 45,  5, 37},
    {63, 31, 55, 23, 61, 29, 53, 21} };

int getline();              /* Function to read line[] from image      */
                            /* file; must return EOF when done.        */
putdot(int x, int y);       /* Plot white dot at given x, y.           */

dither()
{
    int x, y;

    while (getline() != EOF) {
        for (x=0; x<WIDTH; ++x) {
            c = line[x] >> 2;           /* Scale value to 0..63 range   */

            if (c > pattern[x & 7][y & 7]) putdot(x, y);
        }
        ++y;
    }
}
```

2) This program (also written in C) dithers a color image onto an 8-color display by error-diffusion using the Burkes filter.

```
/* Burkes filter error diffusion dithering algorithm in color.  The array
** line[][] contains the RGB values for the current line being processed;
** line[0][x] = red, line[1][x] = green, line[2][x] = blue.
*/

unsigned char line[3][WIDTH];
unsigned char colormap[3][COLORS] = {
      0,   0,   0,     /* Black       This color map should be replaced */
    255,   0,   0,     /* Red         by one available on your hardware */
```

```
      0, 255,   0,     /* Green                                          */
      0,   0, 255,     /* Blue                                           */
    255, 255,   0,     /* Yellow                                         */
    255,   0, 255,     /* Magenta                                        */
      0, 255, 255,     /* Cyan                                           */
    255, 255, 255 };   /* White                                          */

int getline();              /* Function to read line[][] from image    */
                            /* file; must return EOF when done.        */
putdot(int x, int y, int c); /* Plot dot of given color at given x, y.  */

dither()
{
    static int ed[3][WIDTH] = {0};    /* Errors distributed down, i.e., */
                                      /* to the next line.              */
    int x, y, h, c, nc, v,            /* Working variables              */
        e[4],                         /* Error parts (7/8,1/8,5/8,3/8). */
        ef[3];                        /* Error distributed forward.     */
    long dist, sdist;                 /* Used for least-squares match.  */

    for (x=0; x<WIDTH; ++x) {
        ed[0][x] = ed[1][x] = ed[2][x] = 0;
    }
    y = 0;                            /* Get one line at a time from    */
    while (getline() != EOF) {        /* input image.                   */

        ef[0] = ef[1] = ef[2] = 0;    /* No forward error for first dot */

        for (x=0; x<WIDTH; ++x) {
            for (c=0; c<3; ++c) {
                v = line[c][x] + ef[c] + ed[c][x];  /* Add errors from  */
                if (v < 0) v = 0;                   /* previous pixels  */
                if (v > 255) v = 255;               /* and clip.        */
                line[c][x] = v;
            }

            sdist = 255L * 255L * 255L + 1L;      /* Compute the color  */
            for (c=0; c<COLORS; ++c) {            /* in colormap[] that */
                                                  /* is nearest to the  */
#define D(z) (line[z][x]-colormap[c][z])          /* corrected color.   */

                dist = D(0)*D(0) + D(1)*D(1) + D(2)*D(2);
                if (dist < sdist) {
                    nc = c;
                    sdist = dist;
                }
            }
            putdot(x, y, nc);         /* Nearest color found; plot it.  */

            for (c=0; c<3; ++c) {
```

```
                v = line[c][x] - colormap[c][nc]; /* V = new error; h = */
                h = v >> 1;                       /* half of v, e[1..4] */
                e[1] = (7 * h) >> 3;              /* will be filled     */
                e[2] = h - e[1];                  /* with the Floyd and */
                h = v - h;                        /* Steinberg weights. */
                e[3] = (5 * h) >> 3;
                e[4] = h = e[3];

                ef[c] = e[1];                     /* Distribute errors. */
                if (x < WIDTH-1) ed[c][x+1] = e[2];
                if (x == 0) ed[c][x] = e[3]; else ed[c][x] += e[3];
                if (x > 0) ed[c][x-1] += e[4];
            }
        }
        ++y;
    }
}
```

3) This program (in somewhat incomplete, very inefficient pseudo-C) implements error diffusion dithering with the Floyd and Steinberg filter. It is not efficiently coded, but its purpose is to show the method, which I believe it does.

```
/* Floyd/Steinberg error diffusion dithering algorithm in color.  The array
** line[][] contains the RGB values for the current line being processed;
** line[0][x] = red, line[1][x] = green, line[2][x] = blue.  It uses the
** external functions getline() and putdot(), whose purpose should be easy
** to see from the code.
*/

unsigned char line[3][WIDTH];
unsigned char colormap[3][COLORS] = {
      0,   0,   0,     /* Black       This color map should be replaced   */
    255,   0,   0,     /* Red         by one available on your hardware.  */
      0, 255,   0,     /* Green       It may contain any number of colors */
      0,   0, 255,     /* Blue        as long as the constant COLORS is   */
    255, 255,   0,     /* Yellow      set correctly.                      */
    255,   0, 255,     /* Magenta                                         */
      0, 255, 255,     /* Cyan                                            */
    255, 255, 255 };   /* White                                           */

int getline();              /* Function to read line[] from image file;  */
                            /* must return EOF when done.                */
putdot(int x, int y, int c); /* Plot dot of color c at location x, y.    */

dither()
{
    static int ed[3][WIDTH] = {0};      /* Errors distributed down, i.e., */
                                        /* to the next line.              */
```

```
    int x, y, h, c, nc, v,              /* Working variables              */
        e[4],                           /* Error parts (7/8,1/8,5/8,3/8). */
        ef[3];                          /* Error distributed forward.     */
    long dist, sdist;                   /* Used for least-squares match.  */

    for (x=0; x<WIDTH; ++x) {
        ed[0][x] = ed[1][x] = ed[2][x] = 0;
    }
    y = 0;                              /* Get one line at a time from    */
    while (getline() != EOF) {          /* input image.                   */

        ef[0] = ef[1] = ef[2] = 0;      /* No forward error for first dot */

        for (x=0; x<WIDTH; ++x) {
            for (c=0; c<3; ++c) {
                v = line[c][x] + ef[c] + ed[c][x];  /* Add errors from    */
                if (v < 0) v = 0;                   /* previous pixels    */
                if (v > 255) v = 255;               /* and clip.          */
                line[c][x] = v;
            }

            sdist = 255L * 255L * 255L + 1L;        /* Compute the color  */
            for (c=0; c<COLORS; ++c) {              /* in colormap[] that */
                                                    /* is nearest to the  */
#define D(z) (line[z][x]-colormap[c][z])            /* corrected color.   */

                dist = D(0)*D(0) + D(1)*D(1) + D(2)*D(2);
                if (dist < sdist) {
                    nc = c;
                    sdist = dist;
                }
            }
            putdot(x, y, nc);           /* Nearest color found; plot it.  */

            for (c=0; c<3; ++c) {
                v = line[c][x] - colormap[c][nc];   /* V = new error; h = */
                h = v >> 1;                         /* half of v, e[1..4] */
                e[1] = (7 * h) >> 3;                /* will be filled     */
                e[2] = h - e[1];                    /* with the Floyd and */
                h = v - h;                          /* Steinberg weights. */
                e[3] = (5 * h) >> 3;
                e[4] = h = e[3];

                ef[c] = e[1];                       /* Distribute errors. */
                if (x < WIDTH-1) ed[c][x+1] = e[2];
                if (x == 0) ed[c][x] = e[3]; else ed[c][x] += e[3];
                if (x > 0) ed[c][x-1] += e[4];
            }
        } /* next x */
```

```
        ++y;
    } /* next y */
}
```

==========

Bibliography

[1] Foley, J.D. and A. van Dam, Fundamentals of Interactive Computer Graphics, Addison-Wesley, Reading, MA, 1982.

This is a standard reference for many graphic techniques which has not declined with age. Highly recommended. This edition is out of print but can be found in many university and engineering libraries. NOTE: This book has been updated and rewritten, and this new version is currently in print as:

Foley, J.D., A. van Dam, S.K. Feiner, and J.F. Hughes; Computer Graphics: Principles and Practice. Addison-Wesley, Reading, MA, 1990.

This rewrite omits some of the more technical data of the 1982 edition, but has been updated to include information on error-diffusion and the Floyd-Steinberg filter. Currently on computer bookstore shelves and rather expensive (around $75 list price).

[2] Bayer, B.E., "An Optimum Method for Two-Level Rendition of Continuous Tone Pictures," IEEE International Conference on Communications, Conference Records, 1973, pp. 26-11 to 26-15.

A short article proving the optimality of Bayer's pattern in the dispersed-dot ordered dither.

[3] Ulichney, R., Digital Halftoning, The MIT Press, Cambridge, MA, 1987.

This is the best book I know of for describing the various black and white dithering methods. It has clear explanations (a little higher math may come in handy) and wonderful illustrations. It does not contain any code, but don't let that keep you from getting this book. Computer Literacy normally carries it but the title is often sold out.

[MFM note: I can't describe how much information I got from this book! Several different writers have praised this reference to the skies, and I can only concur. Some of it went right over my head -- it's heavenly for someone who is thrilled by Fourier analysis -- but the rest of it is a clear and excellent treatment of the subject. I had to request it on an interlibrary loan, but it was worth the two weeks' wait and the 25 cents it cost me for

the search. University or engineering libraries would be your best bet, as would technical bookstores.]

[4] Floyd, R.W. and L. Steinberg, "An Adaptive Algorithm for Spatial Gray Scale." SID 1975, International Symposium Digest of Technical Papers, vol 1975m, pp. 36-37.

Short article in which Floyd and Steinberg introduce their filter.

[5] Daniel Burkes is unpublished, but can be reached at this address:

Daniel Burkes
TerraVision, Inc.
2351 College Station Road, Suite 563
Athens, GA 30305

or via CIS at UID# 72077,356. The Burkes error filter was submitted to the public domain on September 15, 1988 in an unpublished document, "Presentation of the Burkes error filter for use in preparing continuous-tone images for presentation on bi-level devices." The file BURKES.ARC, in LIB 15 (Publications) of the CIS Graphics Support Forum, contains this document as well as sample images.

[6] Jarvis, J.F., C.N. Judice, and W.H. Ninke, "A Survey of Techniques for the Display of Continuous Tone Pictures on Bi-Level Displays," Computer Graphics and Image Processing, vol. 5, pp. 13-40, 1976.

[7] Stucki, P., "MECCA - a multiple-error correcting computation algorithm for bilevel image hardcopy reproduction." Research Report RZ1060, IBM Research Laboratory, Zurich, Switzerland, 1981.

[8] Heckbert, P. "Color Image Quantization for Frame Buffer Display." Computer Graphics (SIGGRAPH 82), vol. 16, pp. 297-307, 1982.

[9] Frankie Sierra is unpublished, but can be reached via CIS at UID# 76356,2254. Pictorial presentations of his filters can be found in LIB 17 (Developer's Den) of the CIS Graphics Support Forum as the files DITER1.GIF, DITER2.GIF, DITER6.GIF, DITER7.GIF, DITER8.GIF, and DITER9.GIF.

[10] J.F.R. "Frank" Slinkman is unpublished, but can be reached via CIS at UID# 72411,650. The file NUDTHR.ARC in LIB 17 (Developer's Den) of the CIS Graphics Support Forum contains his document "New Dithering Method for Non-Square Pixels" as well as sample images and encoding program.

[11] Lawrence Gozum is unpublished, but can be reached via CIS at UID# 73437,2372. His document "Notes of IDTVGA Dithering Method" can be found in LIB 17 (Developer's Den) of the CIS Graphics Support Forum as the file IDTVGA.TXT.

[12] Robert M. Crawford is unpublished, but can be reached via CIS at UID# 76356,741. The file DGIF.ZIP in LIB 17 (Developer's Den) of the CIS Graphics Support Forum contains documentation, sample images, and demo program.

==========

Other works of interest:

Knuth, D.E., "Digital Halftones by Dot Diffusion." ACM Transactions on Graphics, Vol. 6, No. 4, October 1987, pp 245-273.

Surveys the various methods available for mapping grayscale images to B&W for high-quality phototypesetting and laser printer reproduction. Presents an algorithm for smooth dot diffusion. (With 22 references.)

Newman, W.M. and R.F.S. Sproull, Principles of Interactive Computer Graphics, 2nd edition, McGraw-Hill, New York, 1979.

Similar to Foley and van Dam in scope and content.

Rogers, D.F., Procedural Elements for Computer Graphics, McGraw-Hill, New York, 1985.

More of a conceptual treatment of the subject -- for something with more programming code, see the following work. Alas, the author errs in his discussion of the Floyd-Steinberg filter and uses the "false" filter pattern discussed earlier.

Rogers, D.F. and J. A. Adams, Mathematical Elements for Computer Graphics, McGraw-Hill, New York, 1976.

A good detailed discussion of producing graphic images on a computer. Plenty of sample code.

Kuto, S., "Continuous Color Presentation Using a Low-Cost Ink Jet Printer," Proc. Computer Graphics Tokyo 84,

Image Dithering: Eleven Algorithms and Source Code

Dithering: An Overview

Today's graphics programming topic - dithering - is one I receive a lot of emails about, which some may find surprising. You might think that dithering is something programmers shouldn't have to deal with in 2012. Doesn't dithering belong in the annals of technology history, a relic of times when "16 million color displays" were something programmers and users could only dream of? In an age when cheap mobile phones operate in full 32bpp glory, why am I writing an article about dithering?

Actually, dithering is still a surprisingly applicable technique, not just for practical reasons (such as preparing a full-color image for output on a non-color printer), but for artistic reasons as well. Dithering also has applications in web design, where it is a useful technique for reducing images with high color counts to lower color counts, reducing file size (and bandwidth) without harming quality. It also has uses when reducing 48 or 64bpp RAW-format digital photos to 24bpp RGB for editing.

And these are just image dithering uses - dithering still has extremely crucial roles to play in audio, but I'm afraid I won't be discussing audio dithering here. Just image dithering.

In this article, I'm going to focus on three things:

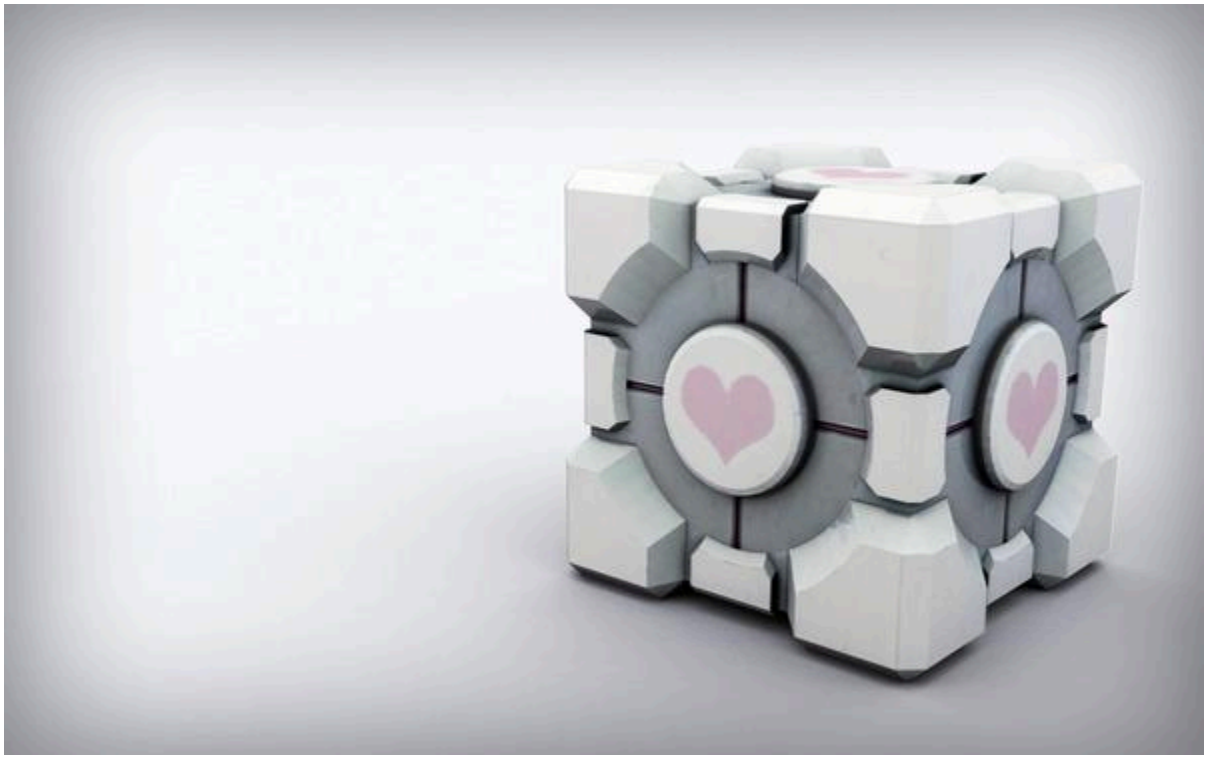
- a basic discussion of how image dithering works

- eleven specific two-dimensional dithering formulas, including famous ones like “Floyd-Steinberg”
- how to write a general-purpose dithering engine

Update 11 June 2016: some of the sample images in this article have been updated to better reflect the various dithering algorithms. Thank you to commenters who noted problems with the previous images!

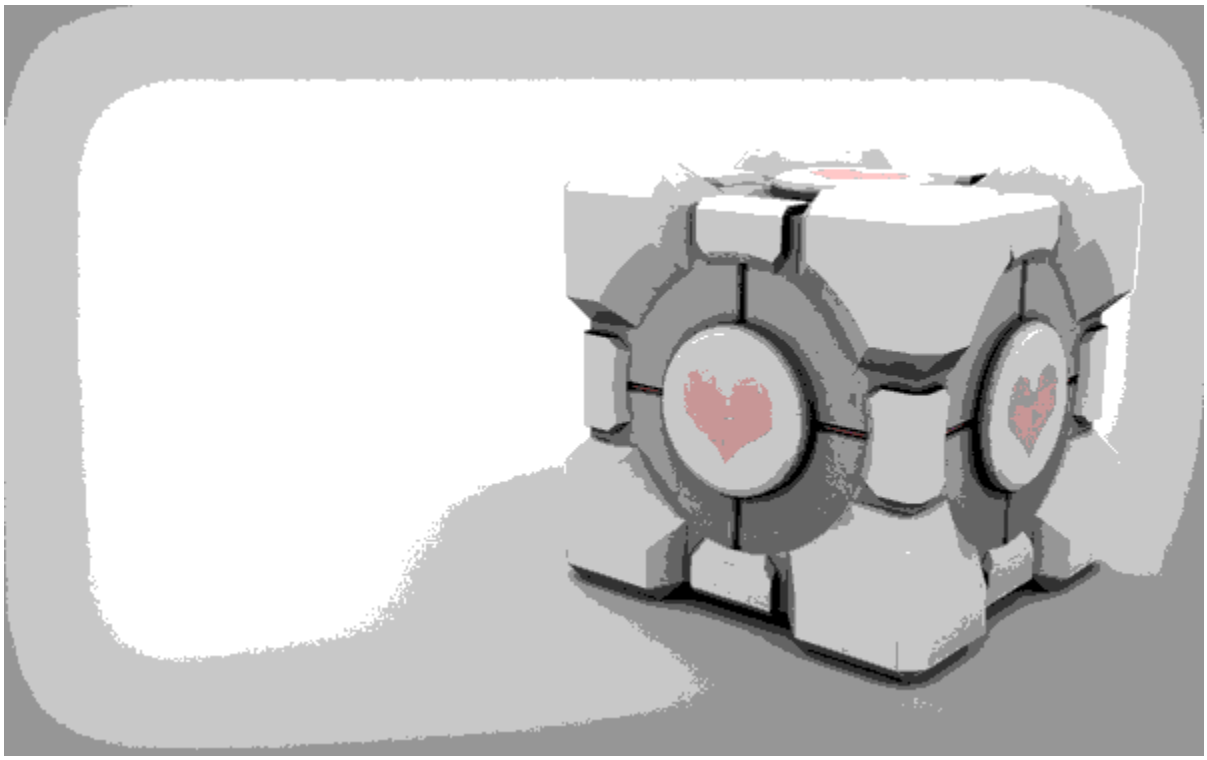
Dithering: Some Examples

Consider the following full-color image, a wallpaper of the famous “companion cube” from *Portal*:



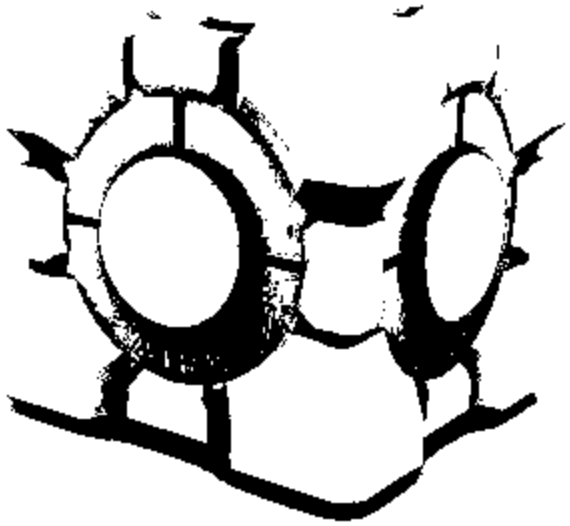
This will be our demonstration image for this article. I chose it because it has a nice mixture of soft gradients and hard edges.

On a modern LCD or LED screen - be it your computer monitor, smartphone, or TV - this full-color image can be displayed without any problems. But consider an older PC, one that only supports a limited palette. If we attempt to display the image on such a PC, it might look something like this:



This is the same image as above, but restricted to a websafe palette.

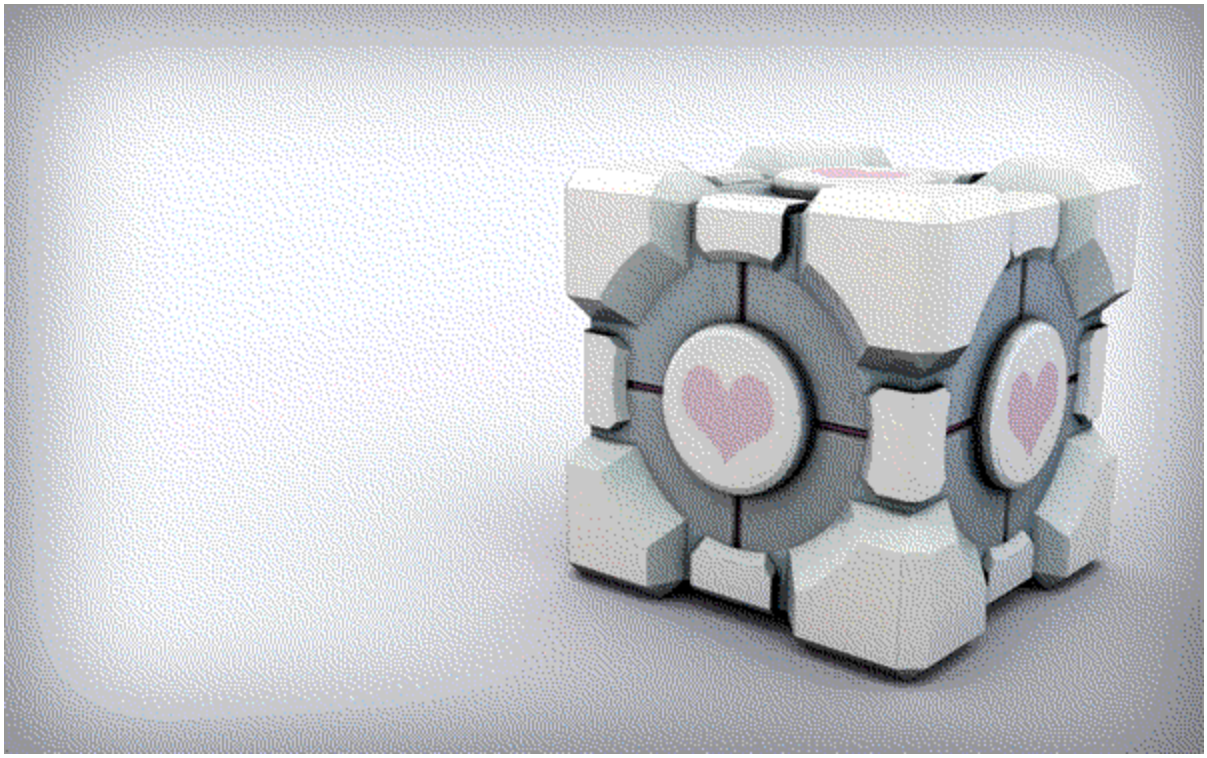
Pretty nasty, isn't it? Consider an even more dramatic example, where we want to print the cube image on a black-and-white printer. Then we're left with something like this:



At this point, the image is barely recognizable.

Problems arise any time an image is displayed on a device that supports less colors than the image contains. Subtle gradients in the original image may be replaced with blobs of uniform color, and depending on the restrictions of the device, the original image may become unrecognizable.

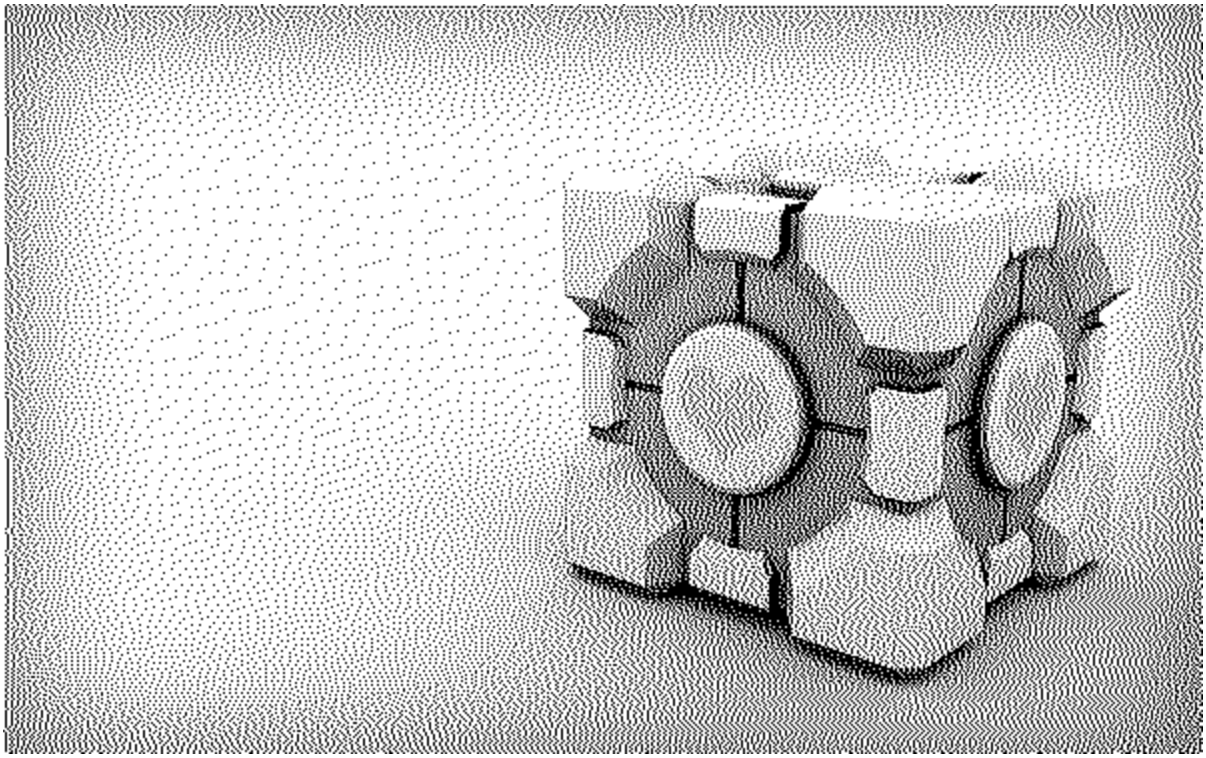
Dithering is an attempt to solve this problem. Dithering works by approximating unavailable colors with available colors, by mixing and matching available colors in a way that mimicks unavailable ones. As an example, here is the cube image once again reduced to the colors of a theoretical old PC - only this time, dithering has been applied:



A big improvement over the non-dithered version!

If you look closely, you can see that this image uses the same colors as its non-dithered counterpart - but those few colors are arranged in a way that makes it *seem* like many more colors are present.

As another example, here is a black-and-white version of the image with similar dithering applied:



The specific algorithm used on this image is “2-row Sierra” dithering.

Despite only black and white being used, we can still make out the shape of the cube, right down to the hearts on either side. Dithering is an extremely powerful technique, and it can be used in ANY situation where data has to be represented at a lower resolution than it was originally created for. This article will focus specifically on images, but the same techniques can be applied to any 2-dimensional data (or 1-dimensional data, which is even simpler!).

The Basic Concept Behind Dithering

Boiled down to its simplest form, dithering is fundamentally about *error diffusion*.

Error diffusion works as follows: let's pretend to reduce a grayscale photograph to black and white, so we can print it on a printer that only supports pure black (ink) or pure white (no ink). The first pixel in the image is dark gray, with a value of 96 on a scale from 0 to 255, with zero being pure black and 255 being pure white.

Here is a visualization of the RGB values in our example.

When converting such a pixel to black or white, we use a simple formula - is the color value closer to 0 (black) or 255 (white)? 96 is closer to 0 than to 255, so we make the pixel black.

At this point, a standard approach would simply move to the next pixel and perform the same comparison. But a problem arises if we have a bunch of "96 gray" pixels - they all get turned to black, and we're left with a huge chunk of empty black pixels, which doesn't represent the original gray color very well at all.

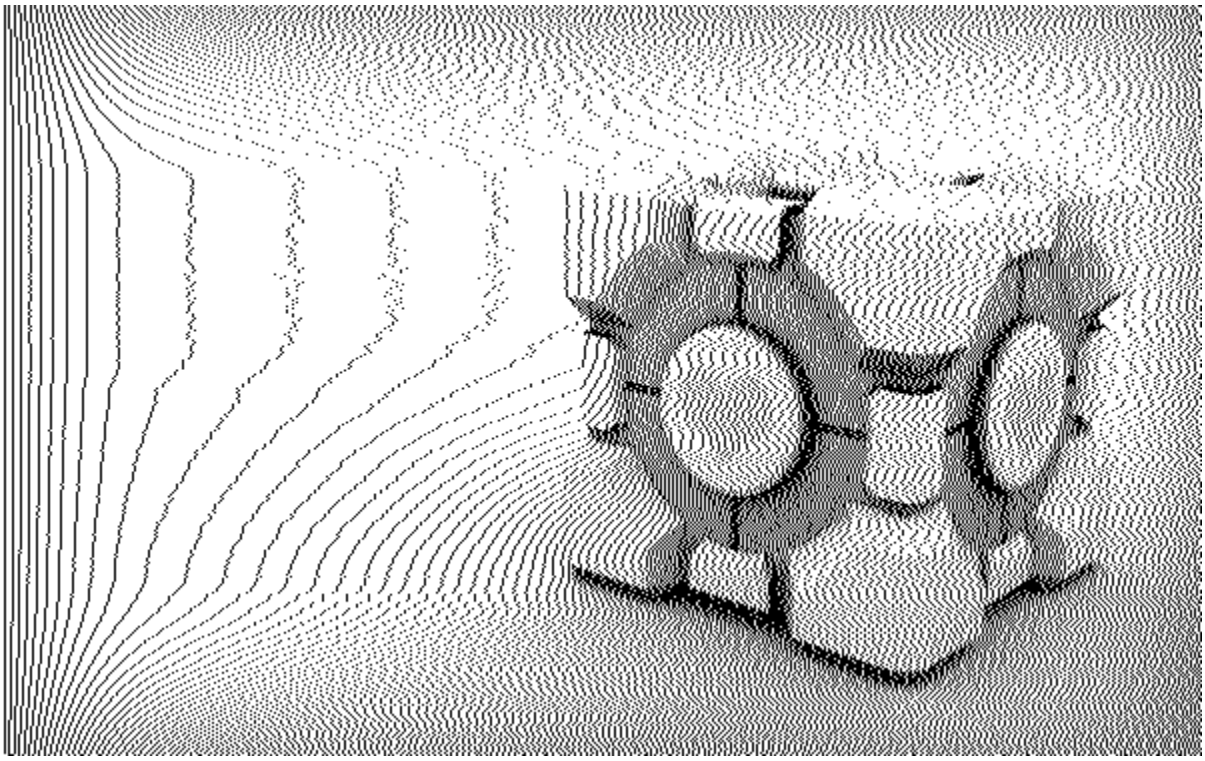
Error diffusion takes a smarter approach to the problem. As you might have inferred, error diffusion works by "diffusing" - or spreading - the error of each calculation to neighboring pixels. If it finds a pixel of 96 gray, it too determines that 96 is closer to 0 than to 255 - and so it makes the pixel black. But then the algorithm makes note of the "error" in its conversion - specifically, that the gray pixel we have forced to black was actually 96 steps away from black.

When it moves to the next pixel, the error diffusion algorithm adds the error of the previous pixel to the current pixel. If the next pixel is also 96 gray, instead of simply forcing that to black as well, the algorithm adds the error of 96 from the previous pixel. This results in a value of 192, which is actually closer to 255 - and thus closer to white!

So it makes this particular pixel white, and it again makes note of the error - in this case, the error is -63, because 192 is 63 less than 255, which is the value this pixel was forced to.

As the algorithm proceeds, the “diffused error” results in an alternating pattern of black and white pixels, which does a pretty good job of mimicking the “96 gray” of the section - much better just forcing the color to black over and over again. Typically, when we finish processing a line of the image, we discard the error value we’ve been tracking and start over again at an error of “0” with the next line of the image.

Here is an example of the cube image from above with this exact algorithm applied - specifically, each pixel is converted to black or white, the error of the conversion is noted, and it is passed to the next pixel on the right:



This is the simplest possible application of error diffusion dithering.

Unfortunately, error diffusion dithering has problems of its own. For better or worse, dithering always leads to a spotted or stippled appearance. This is an inevitable side-effect of working with a small number of available colors - those colors are going to be repeated over and over again, because there are only so many of them.

In the simple error diffusion example above, another problem is evident - if you have a block of very similar colors, and you only push the error to the right, all the “dots” end up in the same place! This leads to funny lines of dots, which is nearly as distracting as the original, non-dithered version.

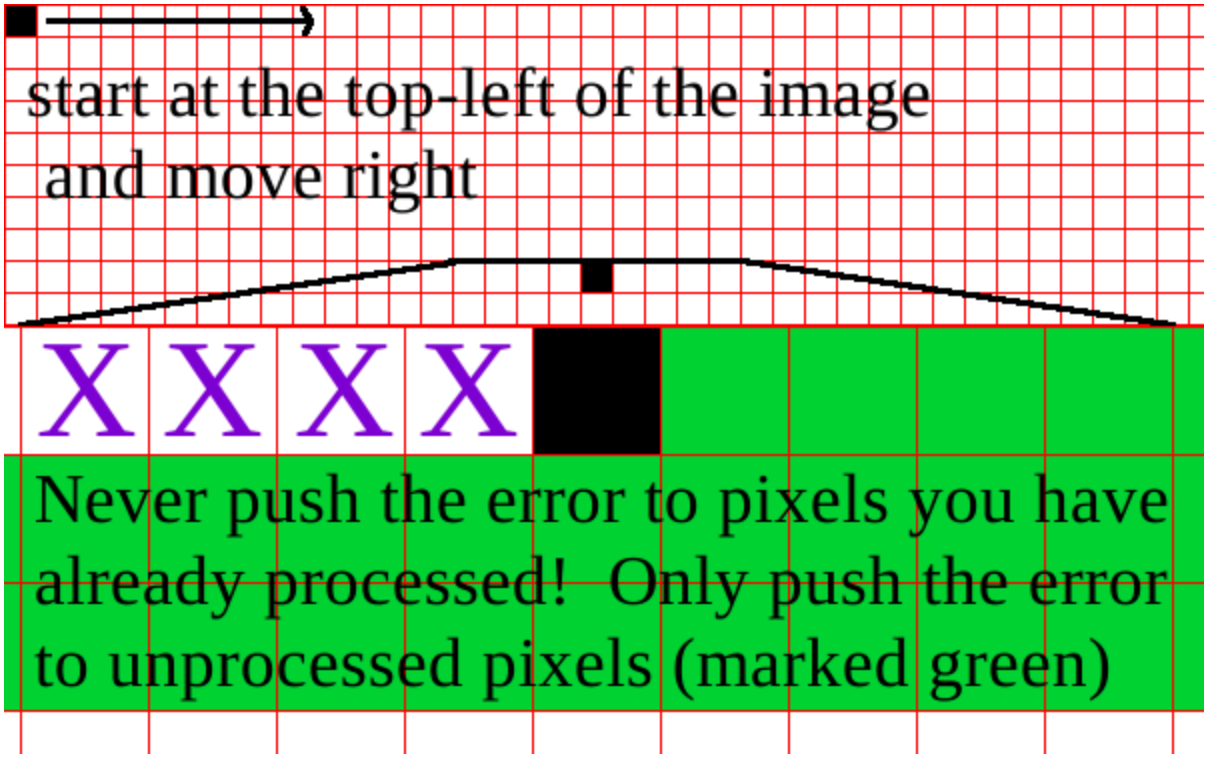
The problem is that we’re only using a *one-dimensional* error diffusion. By only pushing the error in one direction (right), we don’t distribute it very well. Since an image has two dimensions - horizontal and vertical - why not push the error in multiple directions? This will spread it out more evenly, which in turn will avoid the funny “lines of speckles” seen in the error diffusion example above.

Two-Dimensional Error Diffusion Dithering

There are many ways to diffuse an error in two dimensions. For example, we can spread the error to one or more pixels on the right, one or more pixels on the left, one or more pixels up, and one or more pixels down.

For simplicity of computation, all standard dithering formulas push the error *forward*, never backward. If you loop through an image one pixel at a time, starting at the top-left and moving right, you never want to push errors *backward* (e.g. left and/or up). The reason for this is obvious - if you push the error backward, you have to revisit pixels you’ve already processed, which leads to more errors being pushed backward, and you end up with an infinite cycle of error diffusion.

So for standard loop behavior (starting at the top-left of the image and moving right), we only want to push pixels *right* and *down*.



Apologies for the crappy image - but I hope it helps illustrate the gist of proper error diffusion.

As for how specifically to propagate the error, a great number of individuals smarter than I have tackled this problem head-on. Let me share their formulas with you.

(Note: these dithering formulas are available multiple places online, but the best, most comprehensive reference I have found is this one.)

Floyd-Steinberg Dithering

The first - and arguably most famous - 2D error diffusion formula was published by Robert Floyd and Louis Steinberg in 1976. It diffuses errors in the following pattern:

```
      X    7
3     5    1

    (1/16)
```

In the notation above, “X” refers to the current pixel. The fraction at the bottom represents the divisor for the error. Said another way, the Floyd-Steinberg formula could be written as:

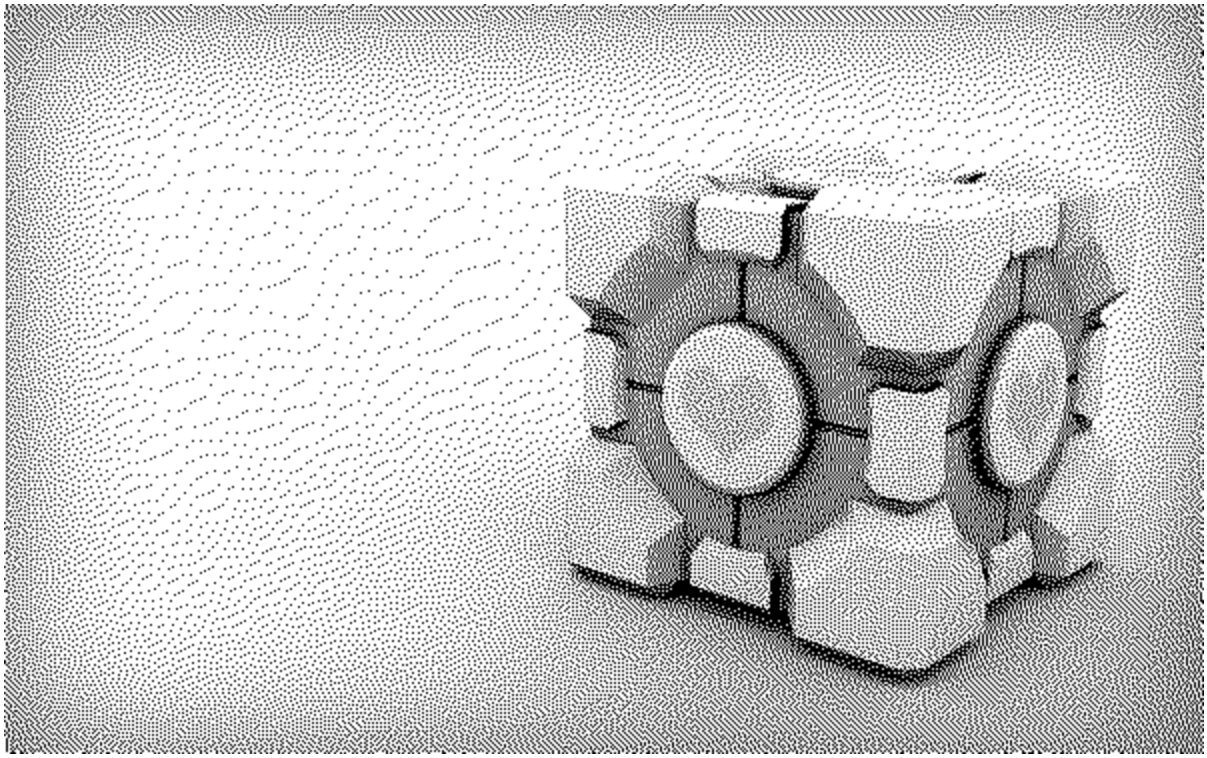
```
          X      7/16
3/16    5/16     1/16
```

But that notation is long and messy, so I’ll stick with the original.

To use our original example of converting a pixel of value “96” to 0 (black) or 255 (white), if we force the pixel to black, the resulting error is 96. We then propagate that error to the surrounding pixels by dividing 96 by 16 (= 6), then multiplying it by the appropriate values, e.g.:

```
          X      +42
+18      +30      +6
```

By spreading the error to multiple pixels, each with a different value, we minimize any distracting bands of speckles like the original error diffusion example. Here is the cube image with Floyd-Steinberg dithering applied:



Not bad, eh?

Floyd-Steinberg dithering is easily the most well-known error diffusion algorithm. It provides reasonably good quality, while only requiring a single forward array (a one-dimensional array the width of the image, which stores the error values pushed to the next row). Additionally, because its divisor is 16, bit-shifting can be used in place of division - making it quite fast, even on old hardware.

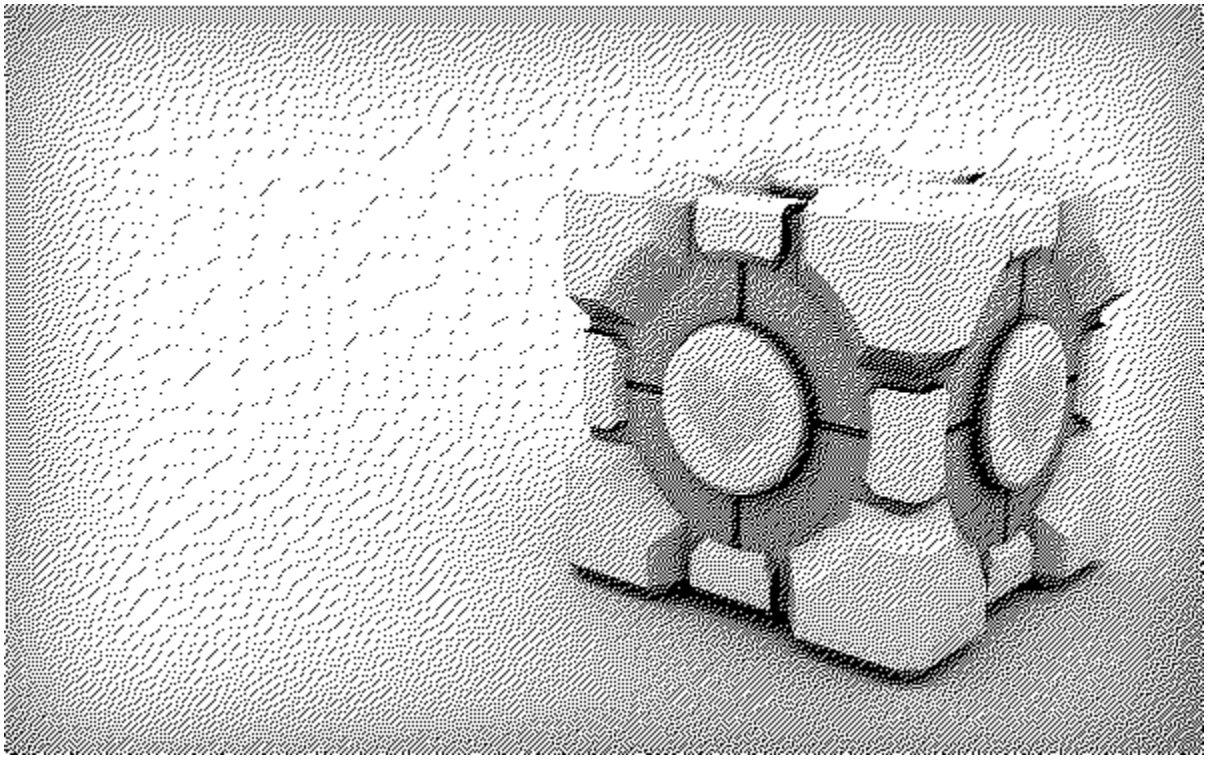
As for the 1/3/5/7 values used to distribute the error - those were chosen specifically because they create an even checkerboard pattern for perfectly gray images. Clever!

One warning regarding “Floyd-Steinberg” dithering - some software may use other, simpler dithering formulas and call them “Floyd-Steinberg”, hoping people won’t know the difference. This excellent dithering article describes one such “False Floyd-Steinberg” algorithm:

```
    X    3
    3    2

    (1/8)
```

This simplification of the original Floyd-Steinberg algorithm not only produces markedly worse output - but it does so without any conceivable advantage in terms of speed (or memory, as a forward-array to store error values for the next line is still required).

But if you're curious, here's the cube image after a "False Floyd-Steinberg" application:



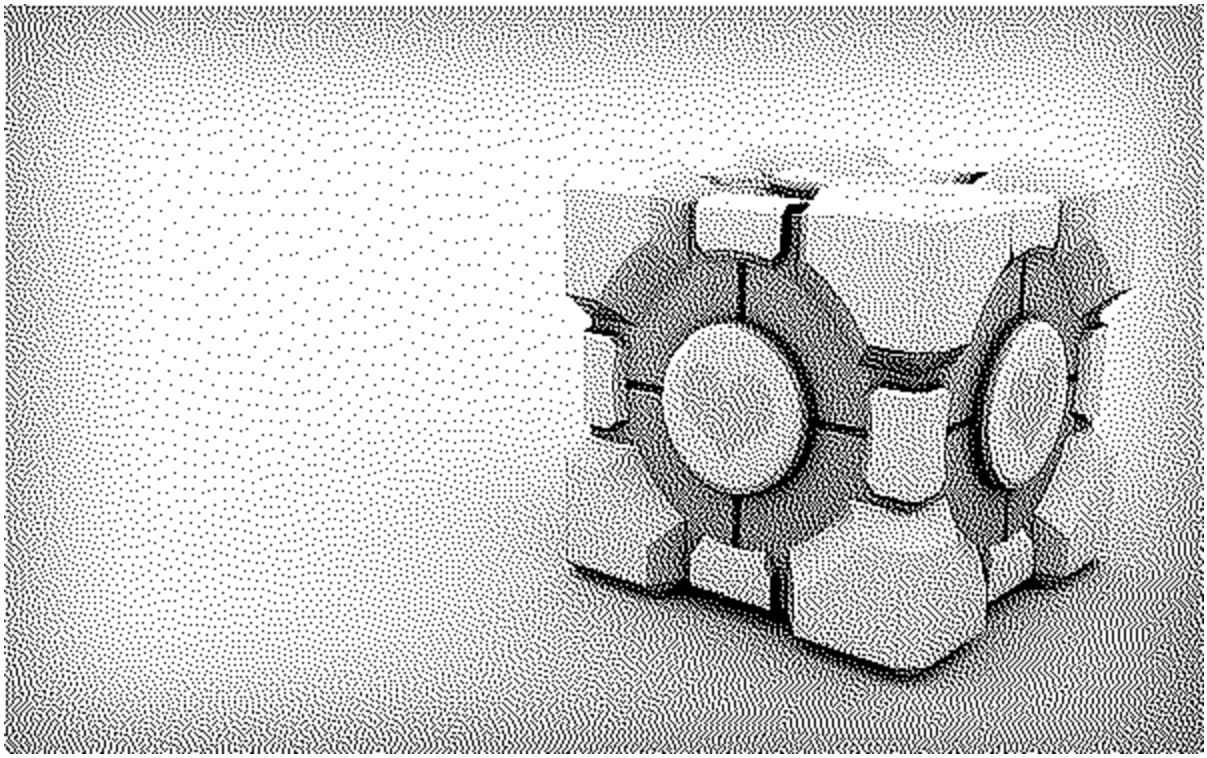
Jarvis, Judice, and Ninke Dithering

In the same year that Floyd and Steinberg published their famous dithering algorithm, a lesser-known - but much more powerful - algorithm was also published. The Jarvis, Judice, and Ninke filter is significantly more complex than Floyd-Steinberg:

```
          X    7    5
3    5    7    5    3
1    3    5    3    1

         (1/48)
```

With this algorithm, the error is distributed to three times as many pixels as in Floyd-Steinberg, leading to much smoother - and more subtle - output. Unfortunately, the divisor of 48 is not a power of two, so bit-shifting can no longer be used - but only values of 1/48, 3/48, 5/48, and 7/48 are used, so these values can each be calculated but once, then propagated multiple times for a small speed gain.

Another downside of the JJN filter is that it pushes the error down not just one row, but *two* rows. This means we have to keep two forward arrays - one for the next row, and another for the row after that. This was a problem at the time the algorithm was first published, but on modern PCs or smartphones this extra requirement makes no difference. Frankly, you may be better off using a single error array the size of the image, rather than erasing the two single-row arrays over and over again.



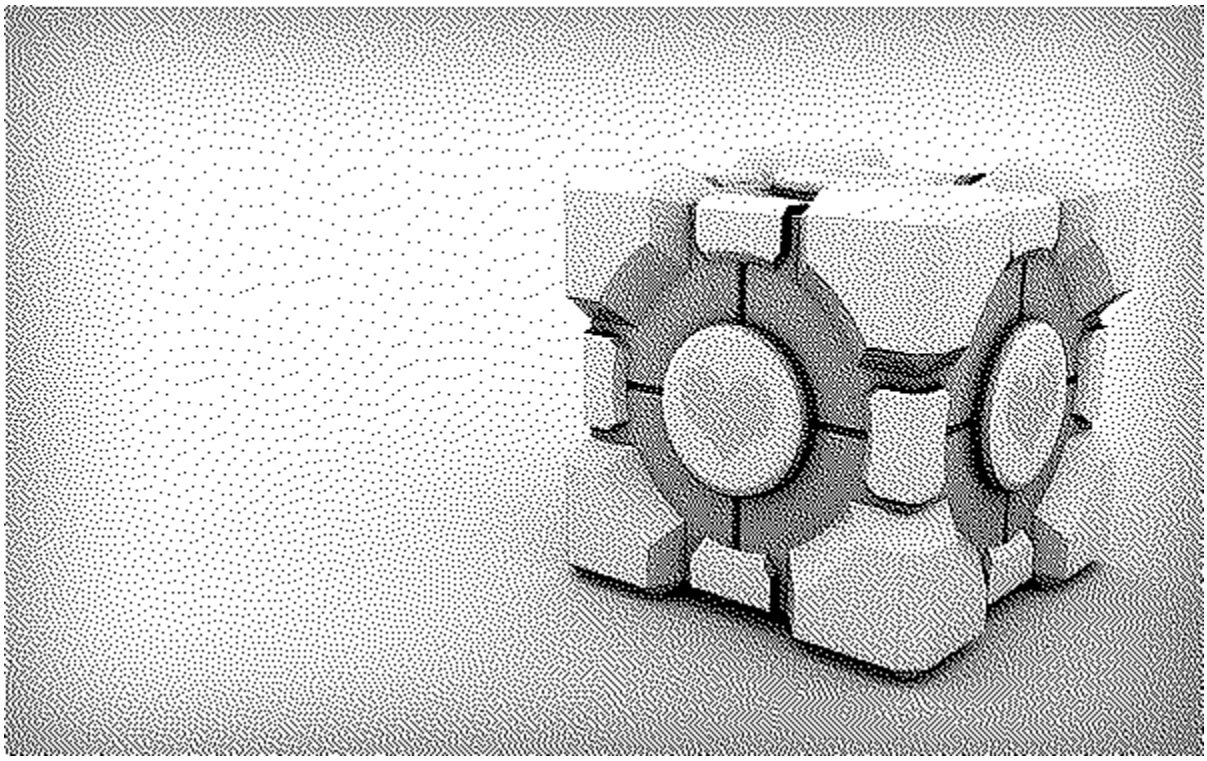
Stucki Dithering

Five years after Jarvis, Judice, and Ninke published their dithering formula, Peter Stucki published an adjusted version of it, with slight changes made to improve processing time:

```
          X   8   4
  2   4   8   4   2
  1   2   4   2   1

        (1/42)
```

The divisor of 42 is still not a power of two, but all the error propagation values are - so once the error is divided by 42, bit-shifting can be used to derive the specific values to propagate.

For most images, there will be minimal difference between the output of Stucki and JJN algorithms, so Stucki is often used because of its slight speed increase.

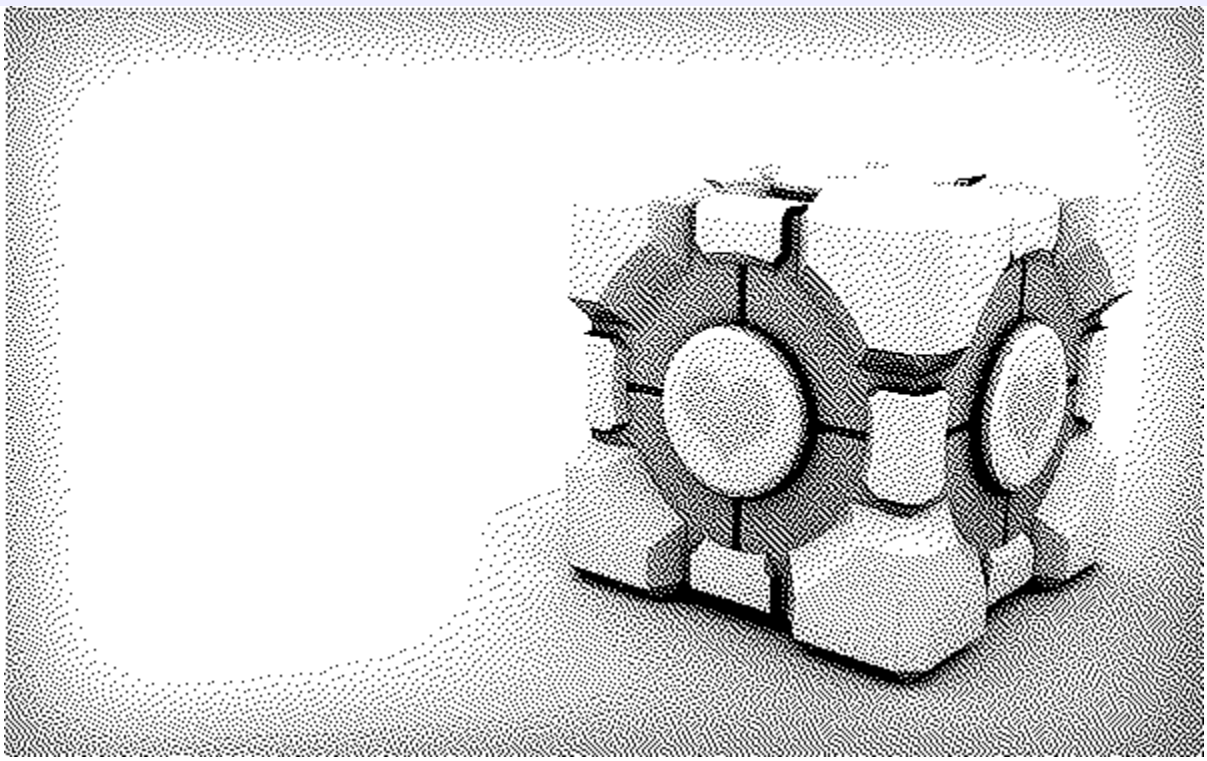


Atkinson Dithering

During the mid-1980's, dithering became increasingly popular as computer hardware advanced to support more powerful video drivers and displays. One of the best dithering algorithms from this era was developed by Bill Atkinson, a Apple employee who worked on everything from MacPaint (which he wrote from scratch for the original Macintosh) to HyperCard and QuickDraw.

Atkinson's formula is a bit different from others in this list, because it only propagates a fraction of the error instead of the full amount. This technique is sometimes offered by modern graphics applications as a "reduced color bleed" option. By only propagating part of the error, speckling is reduced, but contiguous dark or bright sections of an image may become washed out.

```
      X   1   1
  1   1   1
      1

    (1/8)
```

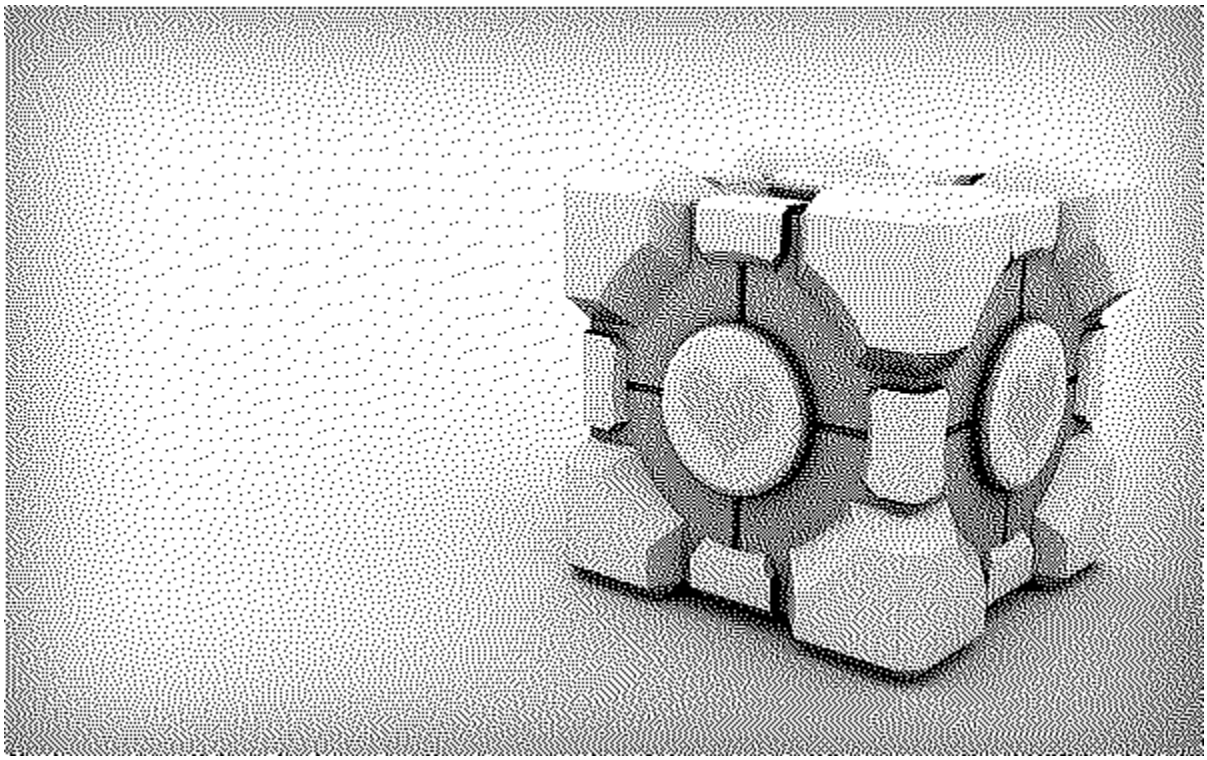


Burkes Dithering

Seven years after Stucki published his improvement to Jarvis, Judice, Ninke dithering, Daniel Burkes suggested a further improvement:

```
          X   8   4
  2   4   8   4   2

      (1/32)
```

Burkes's suggestion was to drop the bottom row of Stucki's matrix. Not only did this remove the need for two forward arrays, but it also resulted in a divisor that was once again a multiple of 2. This change meant that all math involved in the error calculation could be accomplished by simple bit-shifting, with only a minor hit to quality.

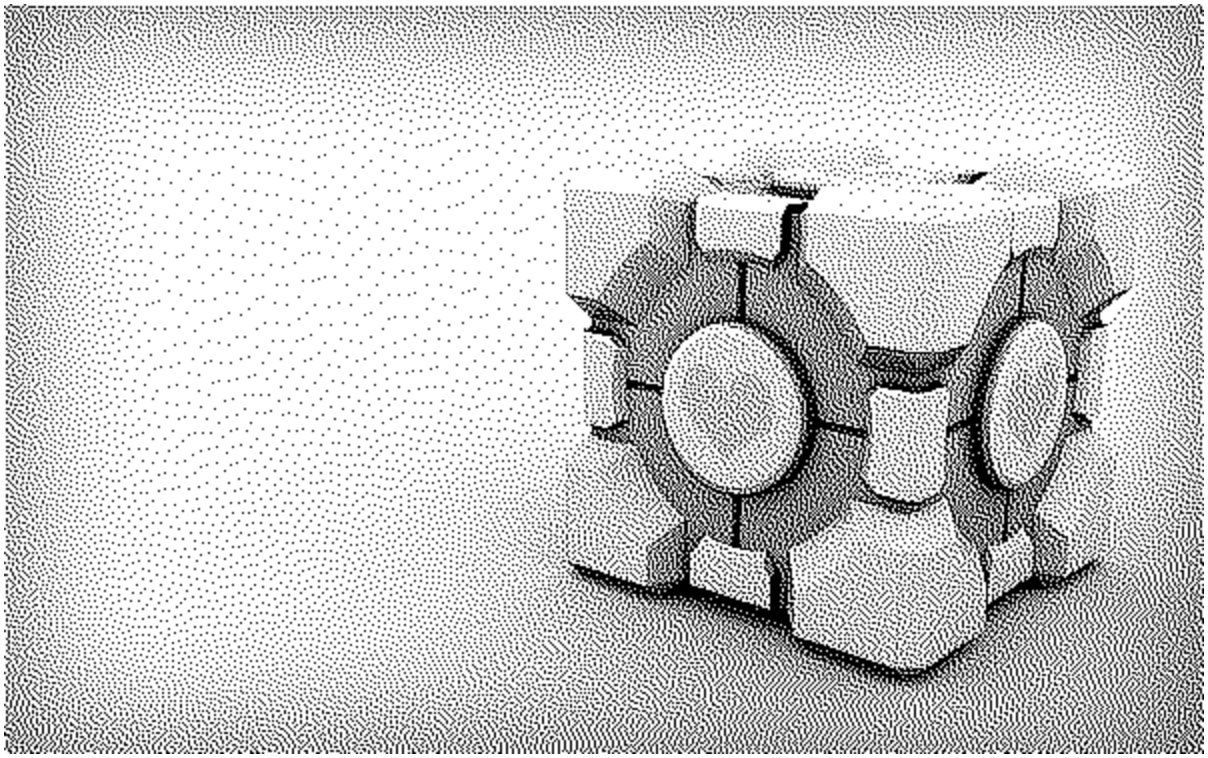


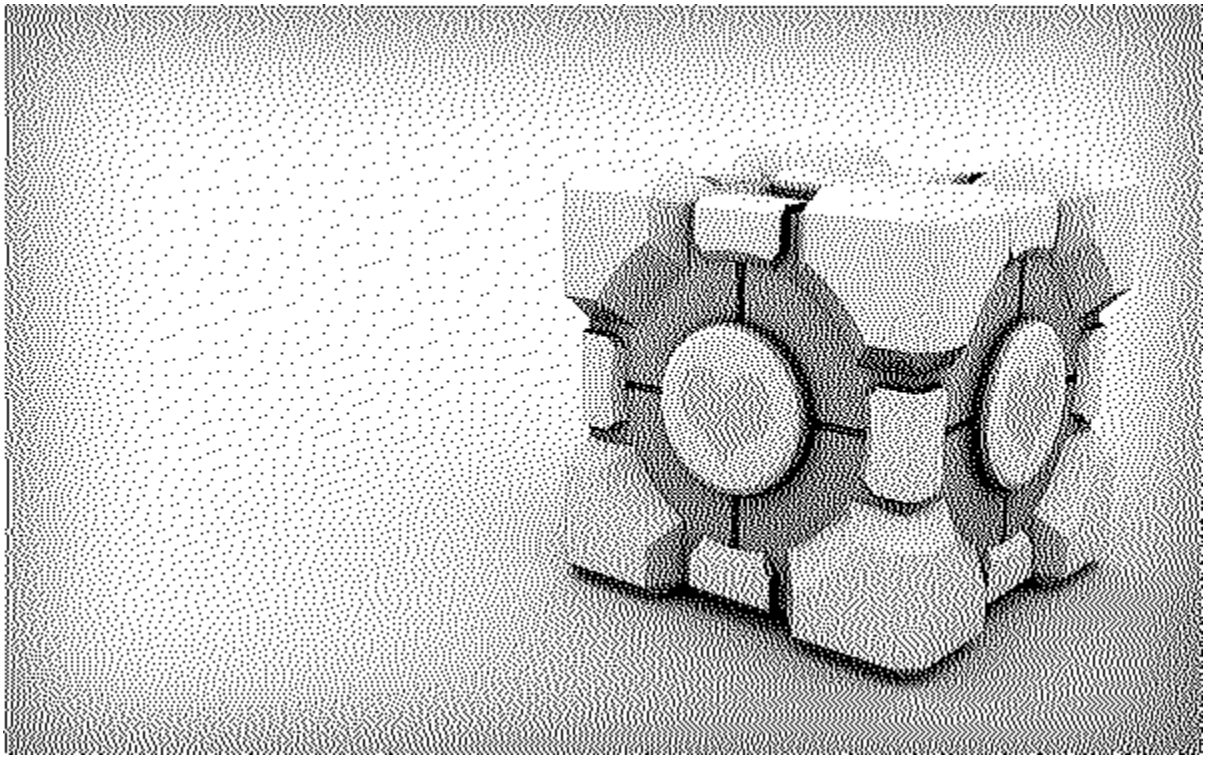
Sierra Dithering

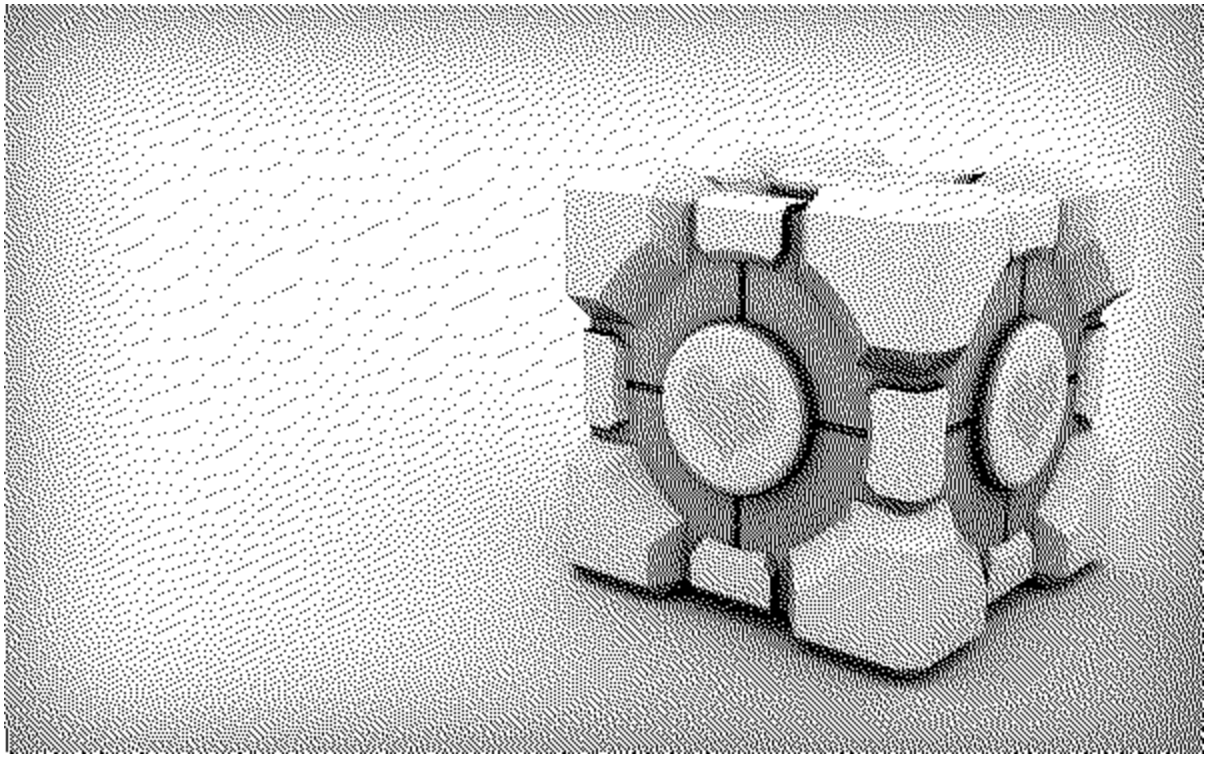
The final three dithering algorithms come from Frankie Sierra, who published the following matrices in 1989 and 1990:

```
          X   5   3
  2   4   5   4   2
      2   3   2
        (1/32)

          X   4   3
1   2   3   2   1
        (1/16)


          X   2
      1   1
        (1/4)
```

These three filters are commonly referred to as “Sierra”, “Two-Row Sierra”, and “Sierra Lite”. Their output on the sample cube image is as follows:







Other dithering considerations

If you compare the images above to the dithering results of another program, you may find slight differences. This is to be expected. There are a surprising number of variables that can affect the precise output of a dithering algorithm, including:

- Integer or floating point tracking of errors. Integer-only methods lose some resolution due to quantization errors.
- Color bleed reduction. Some software reduces the error by a set value - maybe 50% or 75% - to reduce the amount of “bleed” to neighboring pixels.
- The threshold cut-off for black or white. 127 or 128 are common, but on some images it may be helpful to use other values.
- For color images, how luminance is calculated can make a big difference. I use the HSL luminance formula ([max(R,G,B) + min(R,G,B)] / 2). Others use ([r+g+b] / 3) or one of the ITU formulas. YUV or CIELAB will offer even better results.

- Gamma correction or other pre-processing modifications. It is often beneficial to normalize an image before converting it to black and white, and whichever technique you use for this will obviously affect the output.
- Loop direction. I've discussed a standard "left-to-right, top-to-bottom" approach, but some clever dithering algorithms will follow a *serpentine* path, where left-to-right directionality is reversed each line. This can reduce spots of uniform speckling and give a more varied appearance, but it's more complicated to implement.

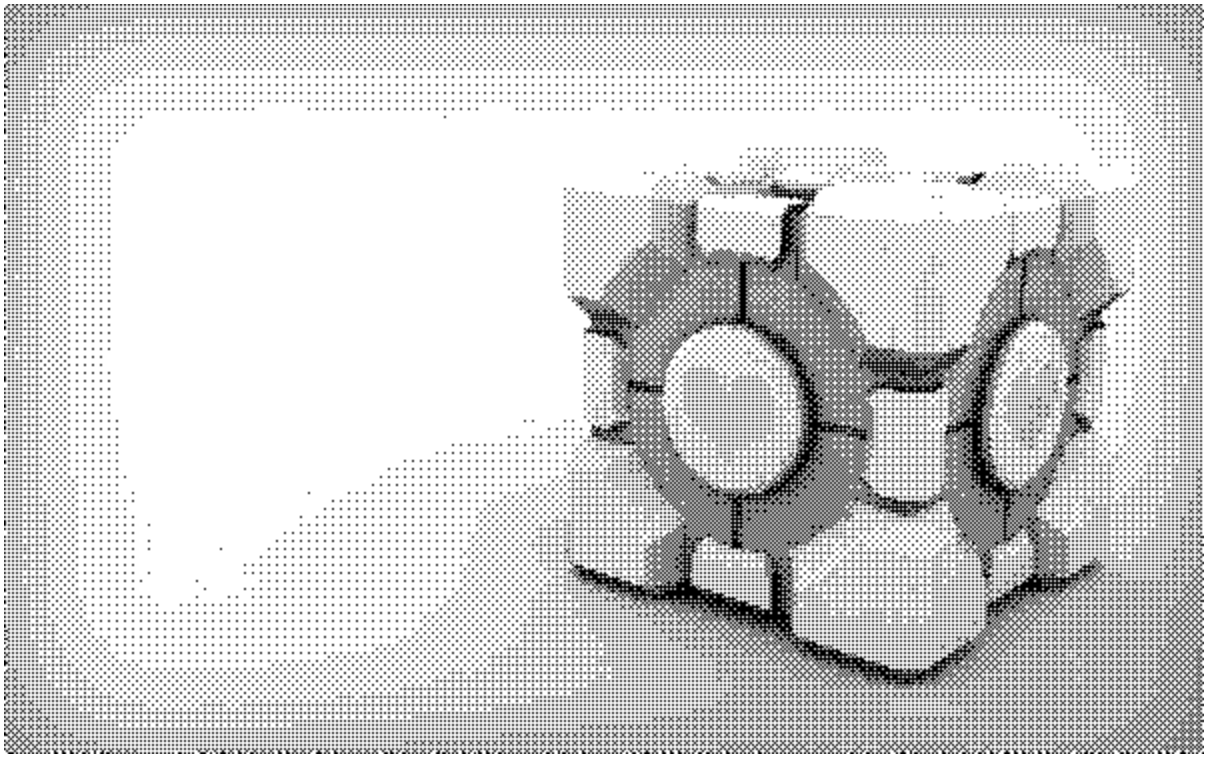
For the demonstration images in this article, I have not performed any pre-processing to the original image. All color matching is done in the RGB space with a cut-off of 127 (values <= 127 are set to 0). Loop direction is standard left-to-right, top-to-bottom.

Which specific techniques you may want to use will vary according to your programming language, processing constraints, and desired output.

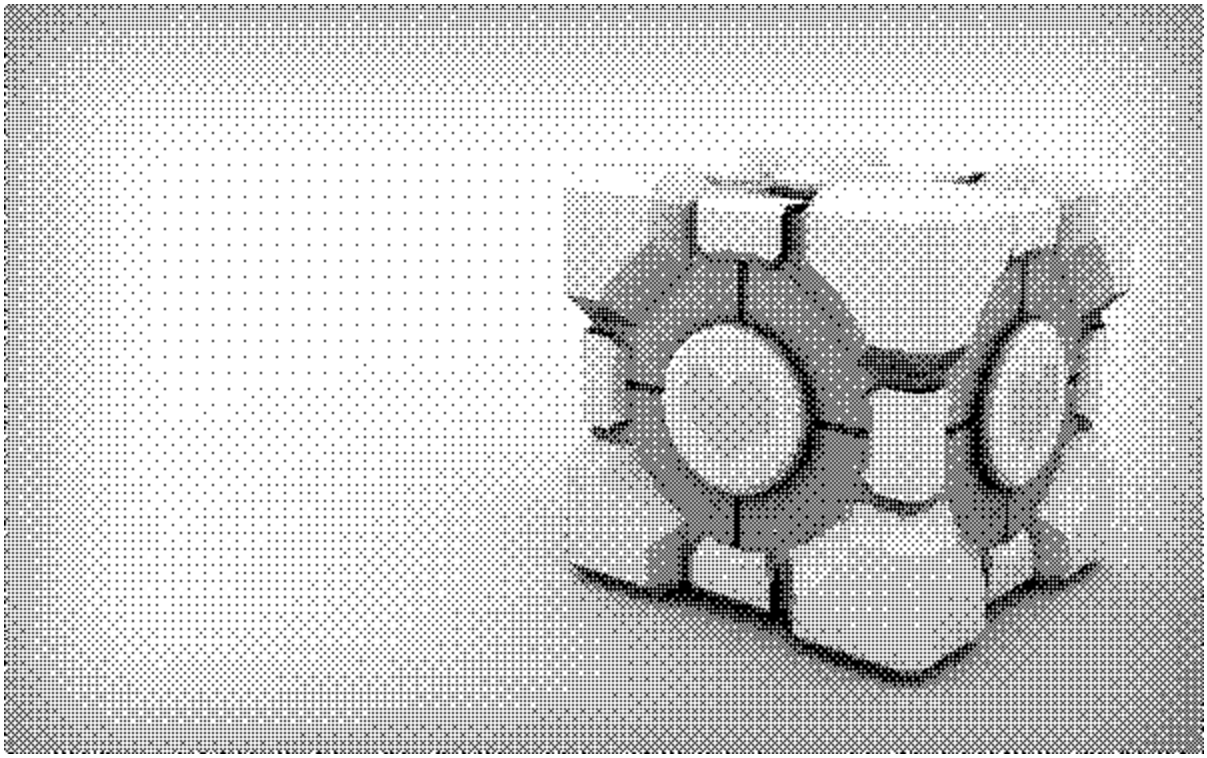
I count 9 algorithms, but you promised 11! Where are the other two?

So far I've focused purely on error-diffusion dithering, because it offers better results than static, non-diffusion dithering.

But for sake of completeness, here are demonstrations of two standard "ordered dither" techniques. Ordered dithering leads to far more speckling (and worse results) than error-diffusion dithering, but they require no forward arrays and are very fast to apply. For more information on ordered dithering, check out the relevant Wikipedia article.



Ordered dither using a 4x4 Bayer matrix



Ordered dither using an 8x8 Bayer matrix

With these, the article has now covered a total of 11 different dithering algorithms.

Writing your own general-purpose dithering algorithm

Earlier this year, I wrote a fully functional, general-purpose dithering engine for PhotoDemon (an open-source photo editor). Rather than post the entirety of the code here, let me refer you to the relevant page on GitHub. The black and white conversion engine starts at line 350. If you have any questions about the code - which covers all the algorithms described on this page - please let me know and I'll post additional explanations.

That engine works by allowing you to specify any dithering matrix in advance, just like the ones on this page. Then you hand that matrix over to the dithering engine and it takes care of the rest.

The engine is designed around monochrome conversion, but it could easily be modified to work on color palettes as well. The biggest difference with a color palette is that you must track separate errors for red, green, and blue, rather than a single luminance error. Otherwise, all the math is identical.

Seven grayscale conversion algorithms (with pseudocode and VB6 source code)

Oct 1, 2011 • by Tanner Helland 

I have uploaded a great many image processing demonstrations over the years, but today's project - grayscale conversion techniques - is actually the image processing technique that generates the most email queries for me. I'm glad to finally have a place to send those queries!

Despite many requests for a grayscale demonstration, I have held off coding anything until I could really present something *unique*. I don't like adding projects to this site that offer nothing novel or interesting, and there are already hundreds of downloads - in every programming language - that demonstrate standard color-to-grayscale conversions. So rather than add one more "here's a grayscale algorithm" article, I have spent the past week collecting every known grayscale conversion routine. To my knowledge, this is the only project on the Internet that presents seven unique grayscale conversion algorithms, and at least two of the algorithms - custom # of grayscale shades with and without dithering - were written from scratch for this very article.

So without further ado, here are seven unique ways to convert a full-color image to grayscale.

Grayscale - An Introduction

Black and white (or monochrome) photography dates back to the mid-19th century. Despite the eventual introduction of color photography, monochromatic photography

remains popular. If anything, the digital revolution has actually *increased* the popularity of monochromatic photography because any digital camera is capable of taking black-and-white photographs (whereas analog cameras required the use of special monochromatic film). Monochromatic photography is sometimes considered the "sculpture" variety of photographic art. It tends to abstract the subject, allowing the photographer to focus on form and interpretation instead of simply reproducing reality.

Because the terminology *black-and-white* is imprecise - black-and-white photography actually consists of many shades of gray - this article will refer to such images as *grayscale*.

Several other technical terms will be used throughout my explanations. The first is *color space*. A *color space* is a way to visualize a shape or object that represents all available colors. Different ways of representing color lead to different color spaces. The RGB color space is represented as a cube, HSL can be a cylinder, cone, or bicone, YIQ and YPbPr have more abstract shapes. This article will primarily reference the RGB and HSL color spaces.

I will also refer frequently to *color channels*. Most digital images are comprised of three separate color channels: a red channel, a green channel, and a blue channel. Layering these channels on top of each other creates a full-color image. Different color models have different channels (sometimes the channels are colors, sometimes they are other values like *lightness* or *saturation*), but this article will primarily focus on RGB channels.

How all grayscale algorithms fundamentally work

All grayscale algorithms use the same basic three-step process:

1. Get the red, green, and blue values of a pixel
2. Use fancy math to turn those numbers into a single gray value
3. Replace the original red, green, and blue values with the new gray value

When describing grayscale algorithms, I'm going to focus on step 2 - using math to turn color values into a grayscale value. So, when you see a formula like this:

```
Gray = (Red + Green + Blue) / 3
```

Recognize that the actual code to implement such an algorithm looks like:

```
For Each Pixel in Image {

  Red = Pixel.Red
  Green = Pixel.Green
  Blue = Pixel.Blue

  Gray = (Red + Green + Blue) / 3

  Pixel.Red = Gray
  Pixel.Green = Gray
  Pixel.Blue = Gray

}
```

On to the algorithms!

Sample Image:

This bright, colorful promo art for The Secret of Monkey Island: Special Edition will be used to demonstrate each of our seven unique grayscale algorithms.

Method 1 - Averaging (aka “quick and dirty”)

This method is the most boring, so let’s address it first. “Averaging” is the most common grayscale conversion routine, and it works like this:

Gray = (Red + Green + Blue) / 3

Fast, simple - no wonder this is the go-to grayscale algorithm for rookie programmers. This formula generates a reasonably nice grayscale equivalent, and its simplicity makes it easy to implement and optimize (look-up tables work quite well). However, this formula is not without shortcomings - while fast and simple, it does a poor job of representing shades of gray relative to the way humans perceive luminosity (brightness). For that, we need something a bit more complex.

Method 2 - Correcting for the human eye (sometimes called “luma” or “luminance,” though [such terminology isn’t really accurate]())

It’s hard to tell a difference between this image and the one above, so let me provide one more example. In the image below, method #1 or the “average method” covers the top half of the picture, while method #2 covers the bottom half:

If you look closely, you can see a horizontal line running across the center of the image. The top half (the average method) is more washed-out than the bottom half. This is especially visible in the middle-left segment of the image, beneath the cheekbone of the background skull.

The difference between the two methods is even more pronounced when flipping between them at full-size, as you can do in the provided source code. Now might be a good time to download my sample project (available at the bottom of this article) so you can compare the various algorithms side-by-side.

This second algorithm plays off the fact that cone density in the human eye is not uniform across colors. Humans perceive green more strongly than red, and red more strongly than blue. This makes sense from an evolutionary biology standpoint - much of the natural world appears in shades of green, so humans have evolved greater sensitivity to green light. *(Note: this is oversimplified, but accurate.)*

Because humans do not perceive all colors equally, the “average method” of grayscale conversion is inaccurate. Instead of treating red, green, and blue light equally, a good grayscale conversion will weight each color based on how the human eye perceives it. A common formula in image processors (Photoshop, GIMP) is:

Gray = (Red * 0.3 + Green * 0.59 + Blue * 0.11)

Surprising to see such a large difference between the red, green, and blue coefficients, isn’t it? This formula requires a bit of extra computation, but it results in a more dynamic grayscale image. Again, downloading the sample program is the best way to appreciate this, so I recommend grabbing the code, experimenting with it, then returning to this article.

It’s worth noting that there is disagreement on the best formula for this type of grayscale conversion. In my project, I have chosen to go with the original ITU-R recommendation (BT.709, specifically) which is the historical precedent. This formula, sometimes called Luma, looks like this:

Gray = (Red * 0.2126 + Green * 0.7152 + Blue * 0.0722)

Some modern digital image and video formats use a different recommendation (BT.601), which calls for slightly different coefficients:

Gray = (Red * 0.299 + Green * 0.587 + Blue * 0.114)

A full discussion of which formula is “better” is beyond the scope of this article. For further reading, I strongly suggest the work of Charles Poynton. For 99% of

programmers, the difference between these two formulas is irrelevant. Both are perceptually preferable to the “average method” discussed at the top of this article.

Method 3 - Desaturation

Next on our list of methods is *desaturation*.

There are various ways to describe the color of a pixel. Most programmers use the RGB color model, where each color is described by its red, green, and blue components. While this is a nice way for a machine to describe color, the RGB color space can be difficult for humans to visualize. If I tell you, “oh, I just bought a car. Its color is RGB(122, 0, 255),” you probably can’t picture the color I’m describing. If, however, I say, “I just bought a car. It is a bright, vivid shade of violet,” you can probably picture the color in question.

For this reason (among others), the HSL color space is sometimes used to describe colors. HSL stands for *hue*, *saturation*, *lightness*. *Hue* could be considered the name of the color - red, green, orange, yellow, etc. Mathematically, hue is described as an angular dimension on the color wheel (range [0,360]), where pure red occurs at 0°, pure green at 120°, pure blue at 240°, then back to pure red at 360°. Saturation describes how vivid a color is; a very vivid color has full saturation, while gray has no saturation. Lightness describes the brightness of a color; white has full lightness, while black has zero lightness.

Desaturating an image works by converting an RGB triplet to an HSL triplet, then forcing the saturation to zero. Basically, this takes a color and converts it to its *least-saturated variant*. The mathematics of this conversion are more complex than this article warrants, so I'll simply provide the shortcut calculation. A pixel can be desaturated by finding the midpoint between the maximum of (R, G, B) and the minimum of (R, G, B), like so:

```
Gray = ( Max(Red, Green, Blue) + Min(Red, Green, Blue) ) / 2
```

In terms of the RGB color space, desaturation forces each pixel to a point along the neutral axis running from (0, 0, 0) to (255, 255, 255). If that makes no sense, take a moment to read this wikipedia article about the RGB color space.

Desaturation results in a flatter, softer grayscale image. If you compare this desaturated sample to the human-eye-corrected sample (Method #2), you should notice a difference in the contrast of the image. Method #2 seems more like an Ansel Adams photograph, while desaturation looks like the kind of grayscale photo you might take with a cheap point-and-shoot camera. Of the three methods discussed thus far, desaturation results in the flattest (least contrast) and darkest overall image.

Method 4 - Decomposition (think of it as *de-composition*, *e.g.* not the biological process!)

Decomposition using maximum values

Decomposition using minimum values

Decomposing an image (sounds gross, doesn't it?) could be considered a simpler form of desaturation. To decompose an image, we force each pixel to the highest (maximum) or lowest (minimum) of its red, green, and blue values. Note that this is done on a per-pixel basis - so if we are performing a *maximum* decompose and pixel #1 is RGB(255, 0, 0) while pixel #2 is RGB(0, 0, 64), we will set pixel #1 to 255 and pixel #2 to 64. Decomposition only cares about which color value is highest or lowest - not which channel it comes from.

Maximum decomposition:

Gray = Max(Red, Green, Blue)

Minimum decomposition:

Gray = Min(Red, Green, Blue)

As you can imagine, a maximum decomposition provides a brighter grayscale image, while a minimum decomposition provides a darker one.

This method of grayscale reduction is typically used for artistic effect.

Method 5 - Single color channel

Grayscale generated using only red channel values

Grayscale generated using only green channel values

Grayscale generated using only blue channel values

Finally, we reach the fastest computational method for grayscale reduction - using data from a single color channel. Unlike all the methods mentioned so far, this method requires no calcuations. All it does is pick a single channel and make that the grayscale value, as in:

Gray = Red

```

```

...or:

Gray = Green

```

```

...or:

Gray = Blue

Believe it or not, this weak algorithm is the one most digital cameras use for taking "grayscale" photos. CCDs in digital cameras are comprised of a grid of red, green, and blue sensors, and rather than perform the necessary math to convert RGB values to gray ones, they simply grab a single channel (green, for the reasons mentioned in Method #2 - human eye correction) and call that the grayscale one. For this reason, most photographers recommend **against** using your camera's built-in grayscale option. Instead, shoot everything in color and then perform the grayscale conversion later, using whatever method leads to the best result.

It is difficult to predict the results of this method of grayscale conversion. As such, it is usually reserved for artistic effect.

Method 6 - Custom # of gray shades

Grayscale using only 4 shades - black, dark gray, light gray, and white

Now it's time for the fun algorithms. Method #6, which I wrote from scratch for this project, allows the user to specify how many shades of gray the resulting image will use. Any value between 2 and 256 is accepted; 2 results in a black-and-white image, while 256 gives you an image identical to Method #1 above. This project only uses 8-bit color channels, but for 16 or 24-bit grayscale images (and their resulting 65,536 and 16,777,216 maximums) this code would work just fine.

The algorithm works by selecting X # of gray values, equally spread (inclusively) between zero luminance - black - and full luminance - white. The above image uses four shades of gray. Here is another example, using sixteen shades of gray:

This grayscale algorithm is a bit more complex. It looks something like:

```
ConversionFactor = 255 / (NumberOfShades - 1)
AverageValue = (Red + Green + Blue) / 3
Gray = Integer((AverageValue / ConversionFactor) + 0.5) * ConversionFactor
```

Notes:
-NumberOfShades is a value between 2 and 256
-technically, any grayscale algorithm could be used to calculate AverageValue; it simply provides
an initial gray value estimate
-the "+ 0.5" addition is an optional parameter that imitates rounding the value of an integer conversion; YMMV depending on which programming language you use, as some round automatically

I enjoy the artistic possibilities of this algorithm. The attached source code renders all grayscale images in real-time, so for a better understanding of this algorithm, load up the sample code and rapidly scroll between different numbers of gray shades.

Method 7 - Custom # of gray shades with dithering (in this example, horizontal error-diffusion dithering)

This image also uses only four shades of gray (black, dark gray, light gray, white), but it distributes those shades using error-diffusion dithering.

Our final algorithm is perhaps the strangest one of all. Like the previous method, it allows the user to specify any value in the [2,256] range, and the algorithm will automatically calculate the best spread of grayscale values for that range. However, this algorithm also adds full dithering support.

What is dithering, you ask? In image processing, dithering uses optical illusions to make an image look more colorful than than it actually is. Dithering algorithms work by interspersing whatever colors are available into new patterns - ordered or random - that fool the human eye into perceiving more colors than are actually present. If that makes no sense, take a look at this gallery of dithered images.

There are many different dithering algorithms. The one I provide here is one of the simplest error-diffusion mechanisms: a one-dimensional diffusion that bleeds color conversion errors from left to right.

If you look at the image above, you'll notice that only four colors are present - black, dark gray, light gray, and white - but because these colors are mixed together, from a distance this image looks much sharper than the four-color non-dithered image under Method #6. Here is a side-by-side comparison:

When few colors are available, dithering preserves more nuances than a non-dithered image, but the trade-off is a “dirty,” speckled look. Some dithering algorithms are better than others; the one I’ve used falls somewhere in the middle, which is why I selected it.

As a final example, here is a 16-color grayscale image with full dithering, followed by a side-by-side comparison with the non-dithered version. As the number of shades of gray in an image increases, dithering artifacts become less and less noticeable. Can you tell which side of the second image is dithered and which is not?

Because the code for this algorithm is fairly complex, I'm going to refer you to the download for details. Simply open the `Grayscale.frm` file in your text editor of choice, then find the `drawGrayscaleCustomShadesDithered` sub. It has all the gory details, with comments.

Conclusion

If you're reading this from a slow Internet connection, I apologize for the image-heavy nature of this article. Unfortunately, the only way to really demonstrate all these grayscale techniques is by showing many examples!

The source code for this project, like all image processing code on this site, runs in real-time. The GUI is simple and streamlined, automatically hiding and displaying relevant user-adjustable options as you click through the various algorithms:

Each algorithm is provided as a stand-alone method, accepting a source and destination picturebox as parameters. I designed it this way so you can grab whatever algorithms interest you and drop them straight into an existing project, without need for modification.

Comments and suggestions are welcome. If you know of any interesting grayscale conversion algorithms I might have missed, please let me know.

(Fun fact: want to convert a grayscale image back to color? If so, check out my real-time image colorization project.)

Download the source code from GitHub