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

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 web-safe 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 mimics 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 diving 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:
Floyd-Steinberg dithering

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:

\[
\begin{array}{ccc}
X & 3 & 2 \\
3 & 2 & \\
\end{array}
\]

\[
(1/8)
\]

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:

Much more speckling than the legit Floyd-Steinberg algorithm – so don’t use this formula!

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:

\[
\begin{array}{ccccc}
X & 7 & 5 & 3 & 1 \\
3 & 5 & 7 & 5 & 3 \\
1 & 3 & 5 & 3 & 1 \\
\end{array}
\]

\[
(1/48)
\]

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.

Jarvis, Judice, Ninke dithering

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:

\[
\begin{array}{ccc}
X & 8 & 4 \\
3 & 5 & 7 \\
1 & 3 & 5 \\
\end{array}
\]

\[
(1/48)
\]
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, an 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.

Burkes Dithering

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

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:

Stucki dithering
Atkinson dithering
Burkes dithering
Sierra dithering

Image Dithering: Eleven Algorithms and Source Code - Tanner Holland (dot) Com
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:

Sierra (sometimes called Sierra-3)

Two-row Sierra

Sierra Lite

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 \( \frac{\max(R,G,B) + \min(R,G,B)}{2} \). Others use \( \frac{r+g+b}{3} \) 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 495. 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.

This site - and its many free downloads - are funded by donations from visitors like you. Please consider a small donation to fund server costs and to help me support my family. Even $1.00 helps. Thank you!

All source code in the above zip file(s) is released under a BSD License.
There are some interesting differences between my own implementation and others. I actually compared my results against GIMP and several other software implementations during the course of this article, and there are differences between them all.

All the criteria under “other dithering considerations” are possible explanations for the differences. Here’s another interesting one I didn’t include in the article – different JPEG libraries can also lead to subtle differences. For example, image software that loads JPEGs using GD2 will return a different unique color count than software that uses libjpeg. I don’t know if these small differences could explain the dithering differences above, but some combination of them all probably contributes.

I also use a slightly modified system for calculating the error:

```
errorVal = newL - 255
Else
ImageData(QuickVal, y) = highB
ImageData(QuickVal + 1, y) = highG
ImageData(QuickVal + 2, y) = highR
End If
```

This calculation is shown here because I find the output cleaner and clearer.

Finally, this image is particularly prone to bringing out differences in implementations, on account of the large patch of uniform pixels on the left. A more busy, “traditional” image would make any differences harder to pick out.

If you download PhotoDemon (which contains my implementation) and set the threshold to 255, you’ll see results very similar to your own Floyd-Steinberg image.

Good luck! Dithering is a neat topic, and I’m really glad you found this information useful.

Anyway, there are lots of fun ways to improve the output of the algorithm. Let me know how your project turns out.

---

I am a (retired) graphic artist and I have a keen interest in the Floyd-Steinberg dither pattern and your modifications of it. They very much resemble the pattern that the “koala” scanner made back in the ‘80s. Do you know of any scanning programs I can use to create this result? Koala has disappeared from graphics. I have used Photoshop’s “grain” pattern with great difficulty. I would appreciate hearing about any programs you know about.

Steve Shapiro

By Steve Shapiro / March 17, 2014, 4:34 pm / Reply to this comment
Hi Steve. I’m afraid I don’t know the “Koala Scanner” pattern you speak of. This article contains all the standard dithering algorithms I’ve seen implemented in other software. A quick check at Wikipedia shows some additional options, like simulated halftoning, but nothing that deviates much from the algorithms above.

Do you have any sample images from that particular scanner? I might be able to track something down if I had a visual reference for the pattern…

By Tanner | March 17, 2014, 9:25 pm | Reply to this comment

6. One optimization that should help with both 1D and 2D-dithering:
Direction Alternating Lines

odd-lines are performed left-to-right
even-lines are performed right-to-left (with the distribution pattern flipped, of course)

the main advantage is that it reduces the visible amount of "color-drifting".

By SkyCharger | July 16, 2014, 3:41 pm | Reply to this comment

7. one of the best articles about image processing I have ever seen, excellent tank you.

By safcomp | August 29, 2014, 10:18 am | Reply to this comment

8. I want to write an error diffusion algorithm into my own software, but I have a question that you didn’t cover in your article.

When I compute the error to the next pixel, should it be NextPixelNewValue = NextPixelOldValue + Error

and allow the new value to possibly go above 255, or go below 0, for the purpose of maintaining the exact pixel-error sum (and then clip the value to keep it in the 0 to 255 range just prior to displaying the image)?

Or should I calculate the next pixel’s new value like this:

NextPixelNewValue = Clip(NextPixelOldValue + Error)

where during the dither process, I always truncate the pixel-error sum to stay within the 0 to 255 value range?

In the first way, the error is always maintained so it the earliest error will effect the calculation of the last pixel. In the second way, the error data will be lost if it pixel-error sum exits the 0 to 255 range, so that the error calculated from the first pixel will NOT be propagated all the way to the last pixel.

Which of these 2 techniques is the correct way of doing error diffusion dithering?

By Ben Hutchinson | September 4, 2014, 1:58 am | Reply to this comment

As I mentioned in the other dithering considerations section:

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

When it comes to diffusion dithering (and really, most graphics programming topics), it’s less about “correct vs incorrect” approaches and more about “what is your desired output?” Said another way, there’s not a definitely correct answer to your question. Try each method and see which one produces the best output for your task.

In this dithering article, which I link above, that author presents his own opinion on these topics in his Special Considerations section. Maybe you’ll find his opinion helpful:

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 to appear across an image.

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 audio signal. 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’s 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.

By Tanner | September 4, 2014, 9:02 am | Reply to this comment

9. I’m doing some AI work, and I think dithering will help simplify my input, its just i need that, but I wish i could reverse the dithering so i could maintain some data integrity. of course i did a 3x3 blur kernel on a 3x3 floyd steinberg, and i think that is what i will do at first, but I wonder is there a better way to reverse dithering – with some mathemagic?

By Magnus | October 3, 2014, 6:36 pm | Reply to this comment

Hi Magnus. As far as I know, you’ve already discovered the best way to reconstruct dithered data. For performance reasons, you may find it easier to count to the number of black pixels in each window (3x3 or some other size), and set the reconstructed value that way, but aside from that, I’m afraid there’s not much more you can do.

If you know which dithering algorithm was used, I suppose you could construct a blur-like filter that weighted pixels using a reverse matrix of the one used for dithering. That might give you slightly better output, but it might also be a lot of work for minimal gain… no way to know for certain without trying. :)

Good luck either way!

By Tanner | October 3, 2014, 7:47 pm | Reply to this comment

I feel it an important thing to perseve, because I discovered ordered dithering is really awesome for temporal compression – because the dataset barely changes between results.

At some level of density, I’m sure you could bring back some precision for a scalar value definitely. (like you actually blur the data into a bit.) A Gaussian blur is perfect for reencoding the values, and then I found an erosion/median filter brought back some of the lost edges. Im still working on it, as binary neural networks are a good opportunity I dont want to not explore enough.

And I feel its important to bring up general database use and dithering, because its a good idea.

By MAGNUS | December 4, 2014, 8:00 am | Reply to this comment

10. I wrote a Floyd Steinberg implementation some years ago in Java and looking back it was one of the most satisfying little projects that I ever worked on. I even added an error diffusion of the alpha channel, which worked really well.

One thing that has been on my mind since then is that the beauty and high performance of the algorithm itself is overshadowed by the clumsiness and slow performance of finding the nearest color in the palette. Are you aware of any fancy data structures or algorithms that could improve performance for finding the nearest color? Currently I have to iterate through all colors in the palette (up to 256), calculate the error (err = dR * dR + dB * dB + dG * dG + dD * dD), and select the entry with the lowest error. Of course I can precalculate x*x for all values up to 256 and stop if err == 0, but that does not really do it.

Oh, and thanks for the really great article.

By Jan | December 4, 2014, 6:47 am | Reply to this comment

Thanks for the kind comment, Jan. Dithering the alpha channel is a great idea – and in fact, that very technique is used in a number of modern PNG compression tools (http://pngquant.org/).

For matching colors to the palette, your best bet is to sort the palette in advance by some criteria, typically distance from RGB(0,0,0). Then you can compare pixels to the palette using a binary search algorithm, so at worst, you’ll be making log(n) error comparisons.

By Tanner | December 4, 2014, 7:41 am | Reply to this comment

Hi Tanner, thanks for the prompt reply.

The idea for improving color matching is interesting, but I think you cannot implement a nearest neighbor search in RGB the way you described, because you really have to seach in three dimensions.
I had a closer look at possible algorithms and found k-d trees, which seem to be an interesting data structure for nearest neighbor search in >2 dimensions ([http://en.wikipedia.org/wiki/K-d_tree](http://en.wikipedia.org/wiki/K-d_tree)). But they seem to be a bit complicated, so I looked further.

Then I found this article by Geoffrey Probert:

What he does is quite simple:

**Preparation:**
1. Add the original palette index to each entry as a property, so it doesn’t get lost when order changes
2. Sort the palette entries by the red component.
3. Create an array of length 256 (redLookupMap) and set at each index the (new) palette index of the palette color that has the closest r distance to the redLookupMap index (if several colors have the same red distance, any of them will do).

**Search:**
1. Use the redLookupMap to find a palette index that has an equal or very close red difference to the sought color.
2. Calculate the Euclidian distance to the found palette color.
3. Move left and right in the palette array and search for the palette entry with the lowest Euclidian distance to the sought color. The trick is that you can stop moving to one direction if the Euclidian distance of only the red component already exceeds the current min distance. This reduces the number of required steps quite a bit.

I just implemented this algorithm for a Floyd Steinberg algorithm using a 256 color palette and voila, the whole procedure is now 3x faster!

*By Jan / December 4, 2014, 2:27 pm*