An Edge-Preserving Inverse Halftoning Algorithm for Ordered Dithered Images

Paul Jensen

(Received on October 31, 2016)

Abstract

This paper provides the reader with an algorithm for removing or reducing noise caused by ordered dither patterns in ordered dithered images without a significant loss of detail. Dithering (or halftoning) uses controlled noise patterns to create the appearance of many colors using a limited color palette. Reducing an image from a larger palette to a smaller one introduces many undesirable artifacts, most notably hard edges between color transitions (called color banding). Color banding is the result of a quantization errors between the original colors in the input and the new ones in the output. Dithering attempts to reduce color banding by dispersing the error to surrounding pixels in a way that is subjectively pleasing to the eye.

However, the fact remains that dithering is itself noise, and noisy images do not lend themselves well to manipulation. Although algorithms such as Gaussian blurring are available for reducing the appearance of noise, applying these algorithms in a naïve way that fails to account for the characteristic dithering patterns present in the input can lead to overapplication of smoothing, resulting in unwanted loss of image detail. The algorithm in this paper uses a conditional smoothing technique based on pattern recognition to reduce dithering noise while also preserving high-contrast pixel boundaries to minimize detail loss.

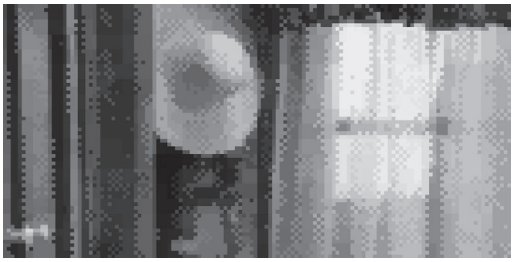
1. Introduction

The impetus for the algorithm outlined in this paper came from a personal software project of mine entitled *SCAT*. SCAT is a media decoding tool aimed at faithfully converting certain legacy media files from their original proprietary format into an open format so that they may be used in modern media editing software. The images contained in the files have a low resolution by current media standards, with a maximum size of around 320 by 240 pixels. They were also intended for older computer systems with limited color capability, and they make heavy use of ordered dither to compensate for that limitation.

When viewed on their original target hardware at their original resolutions, the

images looked reasonably pleasing, as the inherent blur and generally smaller size of older CRT displays tend to diminish the appearance of dither. However, the dither in the images becomes much more pronounced on a modern computer display. The most noticeable artifacts are checkerboard and dotted line patterns in flat-shaded areas and saw tooth-like patterns on vertical and horizontal edges, as illustrated by the images below. The images have been enlarged to better show the artifacts in physical versions of this paper, since the printing process itself involves dithering.

(a) Checkerboard patterns



(b) Saw tooth patterns

Figure 1

Notice the checkerboard patterns present in the input images, most noticeably in the darker areas of the upper subject's face, and the jagged-looking vertical edges on the curtains and door.

Although the original goal of writing SCAT was to produce converted files that were faithful to their original appearance, I wanted to incorporate an algorithm into the rendering code for SCAT to optionally smooth out distracting artifacts such as those seen above. After researching various smoothing methods, I finally settled on an algorithm of my own design, which works very well for images such as these. It could conceivably be used for smoothing out other ordered dithered images as well, so I present it here so others might use it in their own projects, and hopefully even improve upon it.

The main goals I had when creating this algorithm were (1) to reduce the checkerboard patterns and jagged vertical and horizontal edges produced by the dithering process and (2) preserving edges and other details in the images (3) in a consistent manner (4) without using an external image editing tool. Although dithered images have a certain look that many people find pleasing due to their unique aesthetic (Charlton, 2016), the checkerboard patterns and jagged edges of dithered images can also be quite distracting. I wanted to reduce those distracting patterns as much as possible. At the same time, I wanted to keep the rest of the image as close in appearance to the source image as I could, to avoid losing too much of the original "feel" of the image. I also wanted the output of the algorithm to be the identical each time it was run, meaning no use of random values and the like, which have been employed in other inverse halftoning algorithms[1] (Freitas, Farias, & de Araújo, 2011). Finally, it was especially important to me that the smoothing algorithm be integrated into my SCAT program itself, since I wanted to be able to automate the media conversion process to the furthest possible extent.

Digital image manipulation usually falls into the domain of software engineering or signal processing. In the interest of full disclosure, I feel it's worth mentioning that although I have a strong interest in these fields as a hobby, I have no formal education in them. I do however have formal education a theoretical linguist, and I believe the skill sets of modern linguists greatly overlap with those of engineers, particularly when it comes to finding patterns and creating algorithms to process complex data. In that sense, I think my training as a linguist prepared my brain to create algorithms such as the one outlined here. That said, I have made every effort to make this paper accessible to those without a background in any of the fields mentioned above, particularly in terms of formal notation.

In the next section I present the algorithm and explain each of its steps.

2. Algorithm

The algorithm works by a process of augmented discrete Gaussian blurring. The algorithm operates on a single pixel at a time, allowing multiple operations to be run in

1 I'm not attempting to speak pejoratively about algorithms such as these, as they are much more sophisticated than the one outlined in this paper. The choice to avoid the use of randomness in my algorithm was based on personal preference.

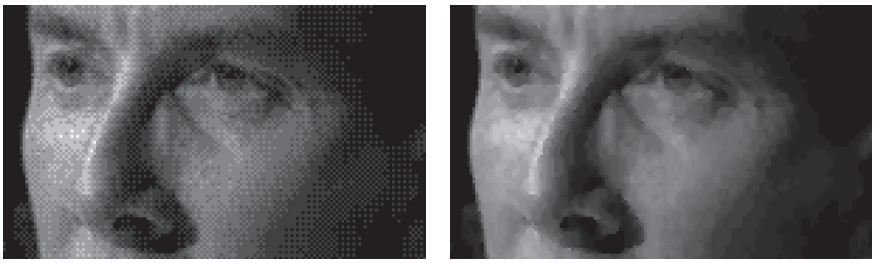
parallel on a multicore CPU or GPU. We might represent the algorithm in pseudocode as follows:

```
FOR each pixel p in the input image:
1) Get a 3x3 window of pixel brightness values from the input image, with
   coordinates ranging from (0,0) to (2,2); pixel p is at (1,1)
2) Create a 3x3 matrix of weight values; set (1,1) to a value of 1, and
   leave all other values at 0
3) IF |(1,0) – (1,2)| <= threshold_lo
   AND |p – (1,0)| <= threshold_hi,
   AND |p – (1,2)| <= threshold_hi
   THEN set weight matrix (1,0) and (1,2) to 0.5
4) IF |(0,1) – (2,1)| <= threshold_lo
   AND |p – (0,1)| <= threshold_hi,
   AND |p – (2,1)| <= threshold_hi
   THEN set weight matrix (0,1) and (2,1) to 0.5
5) IF |(2,0) – (0,2)| <= threshold_lo
   AND |p – (2,0)| <= threshold_hi
   AND |p – (0,2)| <= threshold_hi
   THEN set weight matrix (2,0) and (0,2) to 0.25
6) IF |(0,0) – (2,2)| <= threshold_lo
   AND |p – (0,0)| <= threshold_hi
   AND |p – (2,2)| <= threshold_hi
   THEN set weight matrix (0,0) and (2,2) to 0.25
7) Get a 3x3 window of RGB color values from the input image, with
   coordinates ranging from (0,0) to (2,2); pixel p is (1,1)
8) FOR each value of R, G, B in the window
   a) Multiply the value by its corresponding value in the weight matrix
   b) Add it to a total for that color
9) Divide the total for each color by the sum of the values in the weight
   matrix
10) Use the resulting RGB values as the RGB values of the output pixel.
```

First, the algorithm compares the brightness values of opposing neighbor pixels surrounding a pixel in the center of a 3x3 window. Opposing neighbor pixels are assigned an averaging weight based on (1) their similarity to each other and (2) their similarity to the center pixel. The algorithm considers pixels similar to each other if their absolute difference in brightness is equal to or below a specified threshold (the *low threshold*), and considers them similar to the center pixel if both of their respective absolute differences

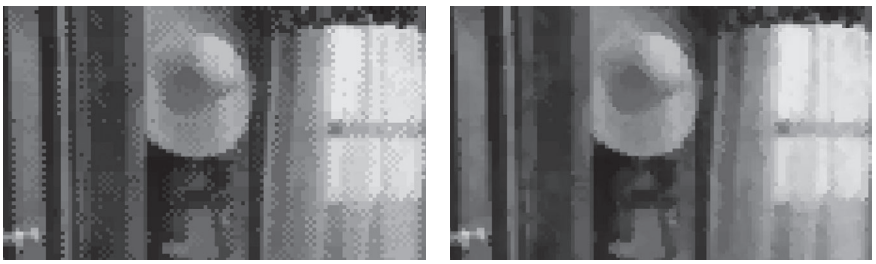
from the center pixel are equal to or below a second specified threshold (the *high threshold*). Similar neighbors at the cardinal poles (north, south, east, and west) are assigned a weight of 0.5; and similar neighbors at the corner poles (northeast, southeast, southwest, and northwest) are given a weight of 0.25. Neighbor pixels that are dissimilar to each other or to the center pixel are not averaged into the output. This condition avoids blurring across high-contrast boundaries. Finally, the algorithm calculates the weighted RGB (red, green, and blue) values of all the pixels in the window. These weighted values constitute the new pixel in the output image.

Below are comparisons of the example input images in Figure 1 with their corresponding outputs after being processed by the algorithm. Notice that the checkerboard patterns present in both (2a) and (3a) are absent from their respective output images. Most of the jagged vertical lines in (3a) have been smoothed into solid ones.



(a) Input **(b) Output**

Figure 2



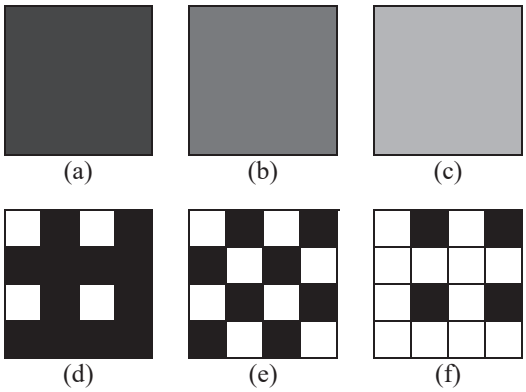
(a) Input **(b) Output**

Figure 3

In the following section, I explain the theory and observations behind the development of the algorithm.

3. Theoretical and observational basis

The algorithm works because ordered dithering creates very predictable arrangements of pixels that make it relatively easy to estimate the original color value of a given output pixel based on the arrangement of its neighbors. Consider how ordered dithering represents the following three levels of gray in flat-shaded areas:



From left to right, (a, d) 75% gray, (b, e) 50% gray, and (c, f) 25% gray

Figure 4

The shaded boxes in the top row of Figure 4 represent three levels of gray: 75%, 50%, and 25%. We would presume that if a color is 75% gray, it would have a mix of black and white at a ratio of 3:1. Likewise, for 50% gray, we would expect a black to white ratio of 1:1. We would also expect that 25% gray would have a black-white ratio of 1:3. Each of the dither patterns in the bottom row correctly represents our intuitions of how these colors should be represented. In the leftmost pattern (4d), there are three black pixels for every white pixel. The middle pattern in (4e) has a matching number of black and white pixels. Finally, the pattern in (4f) has one black pixel for every three white pixels. This means that each pixel in pattern (4d) should be represented as 75% gray, regardless of its representation in the output. Likewise, each pixel in (4e) should appear as 50% gray, and each pixel in (4f) should be 25% gray.

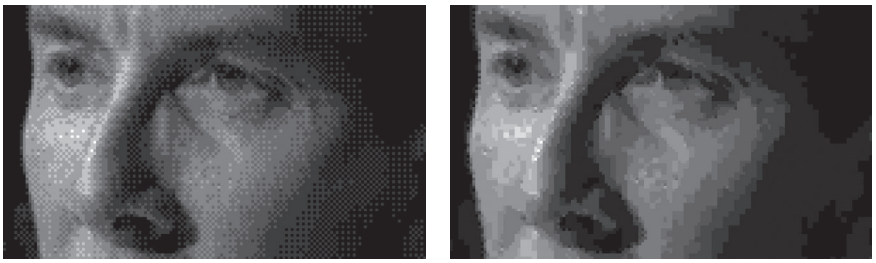
We can use these observations as our starting point in designing an algorithm to represent dithered patterns as solid colors. Given the patterns shown in (4d) though (4f), we might create an algorithm that searches for those three 4x4 pixel patterns in the

output and converts them to their corresponding flat colors in (4a) through (4c). We could even simplify the process if we observe that (4d) through (4f) contain repeating patterns, namely:

Simplified versions of the patterns in Figure 4d through 4f

Figure 5

An algorithm that searches for the patterns above would produce the following output for the example image in Figure 1a:



(a) Input **(b) Output**

Figure 6

The output image shown in Figure 6b represents flat shaded areas as intended, and preserves details well, since it only smooths areas of the image in which pixels exactly match the target patterns. However, the image is noticeably blocky in appearance, and the image also exhibits very prominent banding – that is, the transitions between different levels of gray are very abrupt, and considerably sharper than the simulated transitions in the source image.

Figure 4b shows the results of averaging color values based on matching a square, 2x2 pattern, which leads to a very “squarish”-looking output. So how can we represent the source image in a smoother way? The smoothest shape in nature is the circle, so perhaps we should smooth out our image using circles rather than squares. To do that, we can turn to the discrete Gaussian kernel.

The discrete Gaussian kernel allows us to approximate a circular shape using a matrix of discrete values. The matrix in Figure 7 shows the smallest, most basic form of such a

0.25	0.5	0.25
0.5	1	0.5
0.25	0.5	0.25

Discrete Gaussian kernel

Figure 7

matrix. The value of 1 in the center square represents the target pixel of the smoothing operation. The values in the remaining squares represent the amount of influence the surrounding pixels have on the center pixel. If we take a 3x3 window of brightness values for each pixel and its eight neighbors in the image from Figure 1a and average their values based on the above matrix, we would get the following output:

(a) Input **(b) Output**

Figure 8

The output image in Figure 8b is markedly smoother than the output image in Figure 6b, and displays much smoother transitions between different levels of gray. However, since the entire image has been smoothed indiscriminately, the details in some portions of the image have become perhaps too smooth.

We might at this point choose to aim for a middle ground between the sharp image seen in (6b) and the smother image seen in (8b). To achieve the output seen in (6b), we searched for a window of pixels that matched a pattern and then changed the value of those pixels accordingly. To achieve the output in (8b), we applied a "circular" smoothing filter to the entire image. So, it would seem clear that we would need to combine both approaches to find a middle ground solution.

First, if we want to take advantage of the smoothing capabilities of the discrete

Gaussian kernel, we need to search for patterns that have the same 3x3 pixel dimensions. At first we might reject this idea, since the patterns used in ordered dithering have even dimensions such as 2x2 and 4x4. However, if we observe the canonical patterns of ordered dithering more closely, we see that they fit quite nicely into a 3x3 matrix. Consider once again the pattern for 75% gray:

75% gray pattern

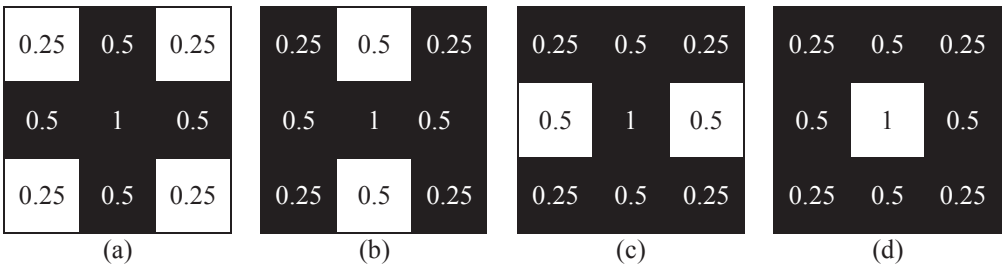
Figure 9

If we break the 4x4 pattern above into 3x3 windows, we get the following patterns:

75% gray pattern as 3x3 patterns

Figure 10

In the output, each pixel at the center of the patterns in Figure 10 should have a value of 75% gray (i.e. a black to white ratio of 3:1). This is exactly what we end up with if we apply the discrete Gaussian matrix in Figure 7 to the patterns in Figure 10:



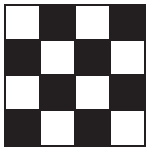
75% gray pattern as 3x3 patterns with Gaussian weights included

Figure 11

If we add up the values of the black pixels in (11a), we get a value of 3. The value of

the white pixels totals 1. Similarly, in (11b), the black pixels total 3, and the white pixels add up to 1. The same is true for (11c), which is a simple rotation of Figure (11b). The values in (11d) also show a total of 3 for black pixels and a total of 1 for white pixels. Each of these 3x3 windows gives us a 3:1 ratio of black to white, which is exactly what we want to have to represent a value of 75% gray.

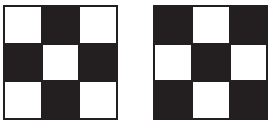
Now consider the pattern representing 50% gray:



50% gray pattern

Figure 12

If we break the 4x4 pattern for 50% gray into 3x3 windows, the following patterns emerge:



(a) (b)

50% gray pattern as 3x3 patterns

Figure 13

In the output, each pixel at the center of the patterns in Figure 13 should have a value of 50% gray (i.e. a black to white ratio of 2:1). Let's see what happens if we apply the discrete Gaussian matrix to these patterns:

0.25	0.5	0.25
0.5	1	0.5
0.25	0.5	0.25

(a)

0.25	0.5	0.25
0.5	1	0.5
0.25	0.5	0.25

(b)

50% gray pattern as 3x3 patterns with Gaussian weights included

Figure 14

The black pixels in (14a) total 2, as do the white pixels. Likewise, in Figure (14b), the black pixels total 2, and the white pixels also total 2. Both windows give us a 2:2 ratio of black to white, which is a value of 50% gray. Once again we have found the exact values we are looking for.

The observations above show why a naïve Gaussian smoothing algorithm would produce an image such as that in Figure 8b. The configuration of the discrete Gaussian kernel ensures that the value of a pixel appearing in the center of a 3x3 window of a pattern for 75% gray, 50% gray, and 25% gray (which is an inverted form of the 75% gray pattern) will be estimated correctly.

So now that we have established that we can create a sharp output by searching for specific patterns, and that we can create a smooth, value-accurate output using a Gaussian kernel, how can we combine the two approaches into a hybrid approach that reaches a "sweet spot" of both sharpness and smoothness?

Now we need to start thinking in terms of which patterns we want to accept as candidates for smoothing, and which patterns we want to reject. We want to accept the patterns seen in Figures 10 and 13, as we have seen that they can be represented in an accurate fashion. But what about patterns such as the following?

Questionable patterns

Figure 15

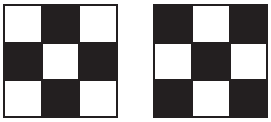
These patterns look like ones we would want to preserve. The pattern in (15a) looks like a hard, vertical edge, and blurring such an edge would most likely result in a loss of detail. Likewise, the pattern in (15b) appears to be a diagonal edge, which we would also want to preserve. (15c) looks like it might be a left to right gradient from black to white. If this is so, we want to preserve the integrity of the pure black and white vertical portions while also representing the center column as a 50% gray vertical line. The pattern in (15d) looks to be a corner of some sort, which we would want to keep. In (15e), we see what appears to be three horizontal lines, which is another detail we would presumably want to preserve.

We might at this point try to devise an algorithm that searches for the specific patterns seen in Figures 10, 13, and 15, as well as their various rotations and reflections, and apply weights to their pixels accordingly. However, there is a simpler approach that we can take based on some observations about the differences between the acceptable patterns and the questionable ones.

Let's first take another look at the patterns we want to accept as being candidates for smoothing (repeated here for convenience):

75% gray pattern as 3x3 patterns

Figure 10 (repeated)



50% gray pattern as 3x3 patterns

Figure 13 (repeated)

If we were to search for these patterns, we would have to compare each of these patterns against the current pixel and its surrounding neighbors. Each pattern could require up to 9 comparisons, for a total of 63 comparisons per pixel. This seems like an unnecessary use of processing power, particularly considering that a single image, even a low resolution one, can contain hundreds of thousands of pixels.

There is one shortcut we can take to avoid a strict pattern-matching solution. We might observe that all the patterns above are very symmetrical. Figures 10a, 10d, 13a, and 13b all have equal reflections and rotations. Figures 10b and 10c have equal reflections, and are indeed rotations of each other. Because they are all symmetrical, the pixels that oppose each other on either side of the center pixel are all the same. We can simply compare the opposing pixels. If all the opposing pixels match, then we can consider the entire pattern to be a match. This would only require a maximum of 4

comparisons per pixel, which is 16 times fewer than a rigid pattern matching algorithm.

So, what about the patterns in Figure 15? If we compare the opposing pixels in Figure 13a, we see that the pixels to the north and south of the center pixel are both black. However, none of the other opposing pixels match, so we would leave this pixel as is. In Figure 15, the opposing pixels in the northwest and southeast corners are identical, but none of the others are, so we would also leave this pixel as it is.

We run into a problem with the pattern in Figure 15c, however, because we would probably want to represent the middle column of pixels as 50% gray. By looking at the polar opposing pixels, we see that the north and south pixels are identical, but the others are not. We wouldn't want to reject it outright as we did with the previous patterns. If we were to blur this pattern in a north-south direction (i.e. a 90-degree angle), we would get the following:

North-south blur result

Figure 16

This is the output that we would want from this pattern. If the algorithm only operates on the center pixel, we would want patterns such as Figure 15c to produce pixels with a 50% gray value. We can achieve this if we apply the Gaussian kernel, but only include the neighbors that match:

0	0.5	0
0	1	0
0	0.5	0

North and south weights included

Figure 17

The black pixels in Figure 17 total 1, as do the white pixels. If we average the values and assign them to the center pixel, we get a value of 50% gray. It's worth mentioning at

this point that we could assign the default weight to each pixel and arrive at the same result:

0.25	0.5	0.25
0.5	1	0.5
0.25	0.5	0.25

All weights included

Figure 18

In Figure 18, both the black pixels and the white pixels have a total value of 2, or a ratio of 1:1. This is indeed the same ratio as we saw in Figure 17. We might wonder at this point if there is a need to selectively assign weights. But if we look back at the patterns in Figures 15a and 15b, we can see that assigning weights selectively gives us the output we want anyway. Consider the following matrices:

0	0.5	0
0	1	0
0	0.5	0

(=Figure 15a with selective weights)

Figure 19

0.25	0	0
0	1	0
0	0	0.25

(=Figure 15b with selective weights)

Figure 20

If we assign weights only to the neighbor pixels that match with their opposing neighbors, we get the values seen above. For Figure 19, the black pixels add up to 2, but that doesn't matter because the only color that receives any weight is black. In this case, the output pixel would be black, which is what we want, because it would preserve the thick, black edge seen in the pattern. In the same way, the black diagonal edge in Figure 20 would be preserved, since the only pixels with any weight are black.

Now what about the patterns in Figures 15d and 15e? These two patterns also pose a problem for our algorithm – particularly (15d). In (15d), all the opposing pixels are identical, so we would predict an output like the following:

0.25	0.5	0.25
0.5	1	0.5
0.25	0.5	0.25

(=Figure 15d with selective weights)

Figure 21

The weighted value of the black pixels adds up to 2, as do the white pixels. This would lead to an output of 50% gray, which is not what we would want to if we wish to preserve the horizontal lines in the pattern.

The pattern in (15e) presents a similar issue:

0	0	0.25
0	1	0
0.25	0	0

(=Figure 15e with selective weights)

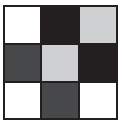
Figure 22

Here, the pixels in the northeast and southwest corners are a match. If we include their weights in the output, we would get a black to white ratio of 2:1, or 50% gray for the

center pixel. This would essentially round off the square corner in the pattern, which would be undesirable.

So how can we reconcile our simple, computationally cheap algorithm with cases such as the patterns in Figure 15d and 15e? One useful fact is that the two patterns occur very rarely in comparison with the other patterns, and so their impact on visual quality is lessened by nature of their rarity. Another observation about these two patterns is that they tend to appear in areas of high contrast. We can use this to our advantage in augmenting our algorithm to handle these cases. We can avoid giving weight to neighbor pixels if they differ too much from the center pixel. For this, we could set a threshold value. If the difference between either of the two neighbor pixels and the center pixel crosses that threshold, they would not be given any weight. By doing this, we can prevent blurring from happening in a direction that would cross a high-contrast boundary. This would double the number of comparisons we would have to make for each pixel, but it could be a useful strategy to avoid blurring straight lines or rounding off corners. In my tests, I have found that a value of 0.5 (i.e. a 50% difference in brightness) works well for defining high-contrast boundaries.

By the same token, what if the opposing neighbor values are not completely identical? Consider the following case:



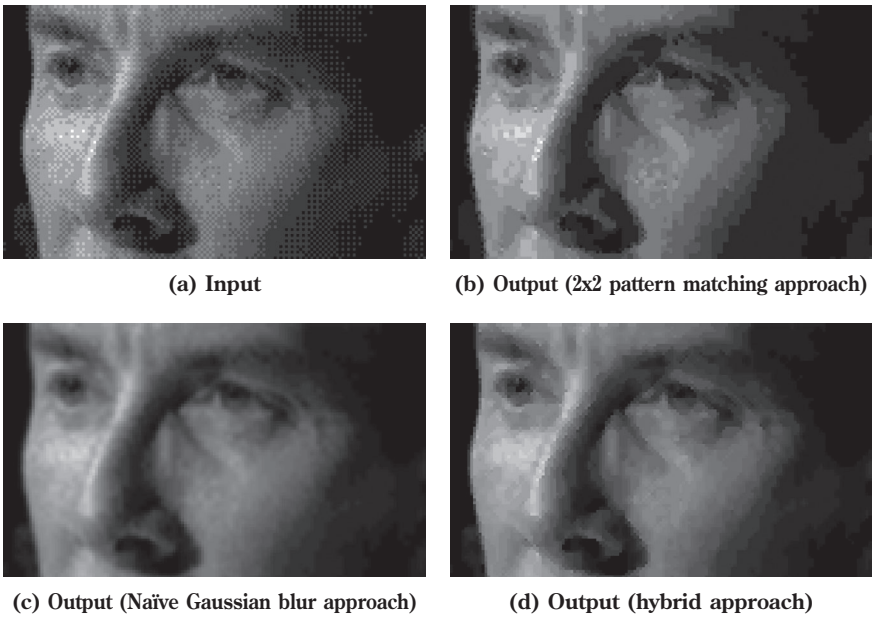
Near 50% gray pattern

Figure 23

This pattern looks arguably close enough to a true 50% gray pattern that we would want to assign a weight value to each of the neighbor pixels. If we want to accept these kinds of patterns, we would need to have a separate threshold value (a tolerance value) that would allow there to be some leeway when comparing opposing neighbor pixels. If the (absolute) difference between the pixels is below the threshold, then the pixels would be given weight. My tests have shown that threshold values ranging from 0.025 to 0.075 work well with most images.

Taking these observations into consideration is what led me to formulate the algorithm outlined in the previous section. Below is a comparison of the input image, the output of

the two candidate approaches considered earlier, and the output produced by the final algorithm:



(a) Input **(b) Output (2x2 pattern matching approach)**

(c) Output (Naïve Gaussian blur approach) **(d) Output (hybrid approach)**

Figure 24

The images above show that the output seen in Figure 24d, produced by the algorithm presented in this paper, manages to eliminate the dithering patterns seen in the input, while retaining detail in areas of higher visual complexity, particularly around the subject's eyes, mouth, and nose. Thus, the algorithm achieves our intended middle ground between the sharpness of (24b) and the smoothness of (24c).

4. Concluding remarks

In this paper, I have introduced a simple method for eliminating dithering patterns from certain images produced by ordered dithering. Dithering reduces the amount of color information in an image, resulting in permanent data loss, therefore it is impossible to perfectly restore a dithered image to its original state, however, I feel that this algorithm produces very high-quality output, especially considering its relative simplicity and computational economy.

I believe this method of smoothing could have applications in the preservation or

restoration of older media. For example, the algorithm could be used to enhance the appearance of image assets when porting a classic video game from an obsolete system to a modern one, particularly in cases where the original, non-dithered assets are no longer available. Although the algorithm can achieve near real-time speeds when implemented purely in software, it could easily be repurposed for real-time applications, such as emulation of older hardware systems, through a graphics processing unit (GPU), allowing for true real-time smoothing of dithered images.

Acknowledgements

This paper would not have been possible were it not for the following people, and to them I give my warm thanks. To my wife Mayu Watanabe and our daughter Lia for giving me the time to work on my academic pursuits. To my senior professor and mentor Seijiro Hiromitsu at Hiroshima Shudo University for always encouraging me to write, write, write. To TascoDLX and the users at the excellent Internet forum on SpritesMind.net for their aid in helping me crack some more challenging media encoding schemes used in the media files supported by SCAT. For Mike Melanson at MultimediaWiki for providing a space for media enthusiasts to document undocumented media encoding formats. Special thanks to the people at the now-defunct video game company Digital Pictures for producing video games such as *Night Trap* and *Prize Fighter* (otherwise SCAT would have no reason to exist). Thanks to the excellent Night Trap community on Facebook for providing feedback and encouragement for the development of SCAT. Extra special credit goes to legendary American boxing announcer Michael "Let's get ready to rumbu~~~le!" Buffer, whose face appears in the example images, which were extracted from the Sega CD game *Prize Fighter*.

References

Bayer, B. (1973). An optimum method for two-level rendition of continuous tone pictures. *IEEE International Conference on Communications*, *1*, pp. 11–15.

Charlton, A. (2016, June 26). *Dithering on the GPU*. Retrieved from alex-charlton.com: http://alex-charlton.com/posts/Dithering_on_the_GPU/

Crocker, L. D., Boulay, P., & Morra, M. (1991, June 20). *DHALF.TXT*. Retrieved from efg's Web Sites: http://www.efg2.com/Lab/Library/ImageProcessing/DHALF.TXT

Freitas, P., Farias, M., & de Araújo, A. (2011). Fast Inverse Halftone for Dispered-dot Ordered

Dithering. *2011 24th SIBGRAPI Conference on Graphics, Patterns, and Images*, (pp. 250–257).

Funkhouser, T. (2000). *Image Quantization, Halftoning, and Dithering.* Retrieved from Princeton University Department of Computer Science: https://www.cs.princeton.edu/courses/archive/fall00/cs426/lectures/dither/dither.pdf

Helland, T. (2012). *Image Dithering: Eleven Algorithms and Source Code.* Retrieved from www.tannerhelland.com: http://www.tannerhelland.com/4660/dithering-eleven-algorithms-source-code/

Jensen, P. (2016, October 28). *SGA Conversion and Analysis Tool.* Retrieved from SourceForge: https://sourceforge.net/projects/sgaconversionandanalysistool/

LaCivita, R., Welsh, K. (Producers), Kusmiak, G. (Writer), & Riley, J. (Director). (1992). *Night Trap* [Motion Picture].

Stein, R. (Director). (1994). *Prize Fighter* [Motion Picture].

Various contributors. (2014, September 23). *SGA*. (M. Melanson, Editor) Retrieved from MultimediaWiki: https://wiki.multimedia.cx/index.php?title=SGA

Wikipedia contributors. (2015, September 27). *Ordered dithering.* Retrieved from Wikipedia: http://en.wikipedia.org/w/index.php?title=Ordered_dithering&oldid=683021611

Ylilunoma, J. (2014, July 7). *Joel Yliluoma's arbitrary-palette positional dithering algorithm*. Retrieved from http://bisqwit.iki.fi/story/howto/dither/jy/