Color Space Binary Dither Interpolation

Steven F. Weed and Tomasz J. Cholewo
Lexmark International, Inc., Lexington, Kentucky
e-mail: weed@lexmark.com

Abstract

Binary Dither Interpolation (BDI) is a linear interpolation algorithm simpler and faster than other methods such as tetrahedral interpolation. Conversion methods employing a lookup table (LUT) process color values in three steps: divide inputs into a table indices and fractions to be interpolated, select LUT entries based on the indices, and weight the selected entries based on the fractions.

In contrast to other more geometrically-based methods, BDI generates a base index by truncation, then uses the truncated fractions to conduct a binary search among selected entries. In spite of having a very simple neighborhood dither implementation, the perceived quality of resulting images is comparable to that of more conventional methods, particularly with constraints on memory access. This paper describes some implementation costs and image quality trade-offs among trilinear, tetrahedral and binary dither interpolation for color space conversion of digital images.

1. Introduction

Interpolation tables are often employed for color space conversions.[1] Historically, printing has been the predominant application for interpolated color conversion, but some recent color video display technologies (such as LCDs) also require relatively complex color space manipulations for good color fidelity. At the same time, some color space representations allowing higher information coding density than RGB involve conversions that are relatively expensive to be performed in real time. Digital processing costs continue to decrease, but simpler and faster color conversion methods with good quality could accelerate the acceptance of advanced color devices and interchange representations.[2]

Precise relationships between perceived colors and control signals for devices such as printers typically defy terse analytical definition and also change over time. LUTs are applicable when the relationships can be usefully approximated as linear between adjacent table entries. For modern digital processing, the speed of interpolation can be constrained by the time to access LUT entries stored in random access memory. Faster memory will enable higher pixel conversion rates, but methods which require fewer accesses allow the use of slower and lower cost memory. Lowest cost embedded methods can avoid a requirement for general purpose or digital signal processor cores by employing only operations which can be economically implemented in ASIC and FPGA chips instead of general purpose processors. Algorithms based upon relatively few primitive logic functions, such as AND, OR, XOR, bit shifts, unsigned integer ADD and subtract, with minimal requirements for access to bulk random access memory (RAM), are preferable to those requiring frequent RAM access or more general arithmetic logic.

Recent progress in spectral-based and principal component image representations present additional challenges for geometrically-based color space interpolations. For example, n-linear interpolation among six components requires access to 64 LUT entries per conversion, while the six-dimensional simplex interpolation involves a somewhat cumbersome task of selecting the correct set of seven from 64 LUT entries.

Color information is often represented by three 8-bit bytes, one for each tristimulus channel. Since humans can visually discern roughly 100 increments along each tristimulus axis, quantizing to more than twice that precision supports the illusion of continuously varying colors. Converting from a 24 bit color space encoding can be accomplished by a lookup table (LUT) with 2^{24} entries. However, RAM storage for 50 megabyte tables is still considered expensive. Many color spaces are defined with explicit and separable algebraic conversions. However, these conversions often involve mathematical functions for which sufficiently fast and accurate digital implementations remain too costly for widespread deployment. A lookup table with integer interpolation remains a viable option.

Sparse tables have long been used for characterizing relatively complicated numeric relationships. Uniform spacing simplifies tables implementation, and table size is determined by the desired range and accuracy of results. Linear interpolation can be applied provided relationships are sufficiently linear among nearest table entries. Cubic splines could be applied to improve interpolation accuracy with sparser tables, but 32 LUT accesses are required for a 3-dimensional cubic spline interpolation, and the time for each LUT access becomes a limiting consideration in digital implementations. For a number of tristimulus color spaces, including sRGB and CIELab, $17 \times 17 \times 17$ color LUTs are common. Starting with 8 bits for each tristimulus value, an interpolation unit cube is selected by truncating each value to its 4 most significant bits.

Special precautions have to be taken in last input interval when using bytes for table values as well as indices. There are two basic methods to special case the final table intervals for byte indices and byte table values: (1) recognize final intervals as one unit shorter than others; (2) rescale final table entries. However, rescaling in general requires table values outside the range of possible byte values. In closed systems, integer interpolation may be employed without special treatment for the last interval by appropriate compensations in other processing, such as halftone threshold array values.

Implementing an interpolation using only non-negative integers is attractive for speed and cost considerations and seems reasonable, given inputs, outputs and LUT of constrained integer precision. At the same time, avoiding division by other than powers-of-two (which can be accomplished by shifts) also helps minimize cost/performance. Round-off errors are a consideration in integer processing. Given that in practice the difference between 0 and 1 is often more important than the difference between 254 and 255, it is generally appropriate to arrange LUTs so that small values are concentrated near the origin. For XYZ and RGB, this may involve inverting inputs when outputs will be in a subtractive color space.

2. Linear Interpolation Algorithms

Digital table interpolation proceeds by selecting a nearest set of table entries. For example, consider a 2-D table T representing a function of arguments X and Y. Values X and Y are converted to grid indices x, y and fractions 0 • a, b • 1, respectively.

Two-dimensional *n-linear interpolation* (bilinear interpolation) calculates

$$f(X, Y) = (1 - a)(1 - b)T_{x,y} + a(1 - b)T_{x+1,y} + b(1 - a)T_{x,y+1} + abT_{x+1,y+1}.$$

Two-dimensional *simplex interpolation* (triangular interpolation) uses only three vertices of the lookup table surrounding the interpolated point. If a > b:

$$f(X, Y) = (1 - a)T_{x,y} + (a - b)T_{x+1,y} + bT_{x+1,y+1},$$

else, for $a \bullet b$:

$$f(X, Y) = (1 - b)T_{x,y} + (b - a)T_{x,y+1} + aT_{x+1,y+1}.$$

While triangular interpolation considers numeric rank among fractions, BDI considers bit combinations of equal significance in fractions, assigning weights according to significance. A 4-bit two-dimensional BDI calculates:

$$f(X, Y) = (8 + 8T_{x+(8\&a)/8,\ y+(8\&b)/8} + 4T_{x+(4\&a)/4,\ y+(4\&b)/4} + 2T_{x+(2\&a)/2,\ y+(2\&b)/2} + T_{x+(1\&a),y+(1\&b)} + T_{x,y})/16$$

where '&' represents bit-wise logical AND.

The above equations can be generalized to higher dimensions. For this paper our focus is on three-dimensional interpolation for color conversion.

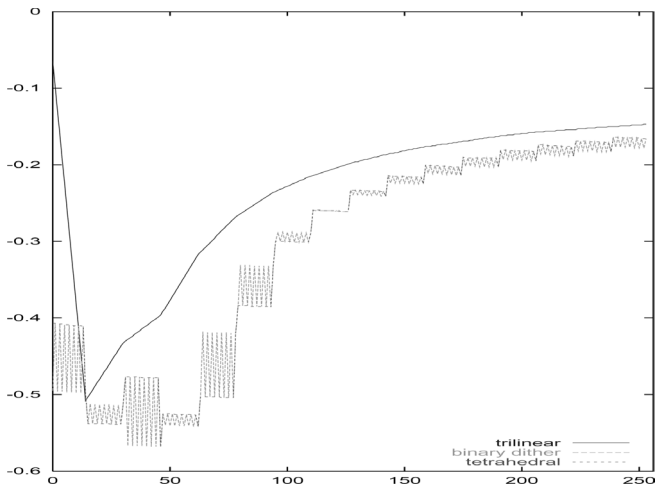


Figure 1. Noisy gray interpolation from 24-bit CIELab to floating point RGB (R - G plotted).

Ranking fractions for simplex interpolation becomes increasingly expensive for higher dimensions, while BDI masks fraction bits of equal significance from additional fractions to select which table entry is assigned a power-of-two weight. Ranking fractions will minimize the table entries accesses for dimensions less than the number of bits of precision in the fractions. Being fundamentally algebraic, albeit with some iterations for selection and ranking, tetrahedral interpolation is readily described by algebraic expression, as above. BDI is a substantially binary procedure, more readily described by pseudo-code. Comparable integer pseudo-codes for BDI and tetrahedral interpolation, both interpolating for the four least significant bits of 24-bit RGB, are shown in Figures 4 and 5.

Like simplex interpolation, BDI can access fewer LUT entries than n-linear interpolation. As with simplex, the number of LUT entries which BDI accesses is input data dependent. For example, truncated 4-bit tristimulus fractions 0100,0100,0100 will only access two of eight nearest table entries: [0,0,0] and [1,1,1]. Simplex interpolation and binary dither are less robust than n-linear to noise in sparse table entries. Specifically, as fewer table entries are used for each color space conversion, model-based LUTs are more likely to give smooth results than tables inverted from data measurements by numerical methods. LUT generation is relatively infrequent in comparison to LUT usage, so using more sophisticated processing to improve the quality of frequently used LUTs seems an easy trade-off.

Trilinear conversion is often used for tristimulus spaces, but requires access to eight lookup table (LUT) entries for each conversion. Tetrahedral and BDI are at a disadvantage when interpolating neutrals from a space such as CIELab, where neutrals lie along an axis (Figure 1). On

the other hand, Figure 2 shows errors of noisy gray interpolation from 24-bit RGB to floating point CIELab. Trilinear can generate anomalous results, for example when calculating near-neutral values along diagonals in RGB-like spaces.

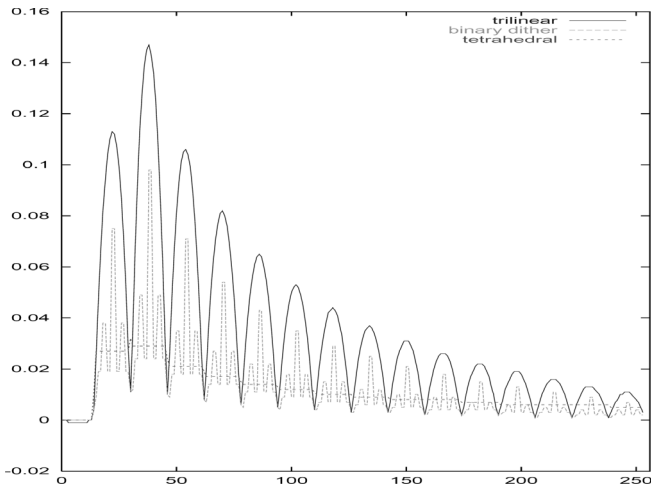


Figure 2. Noisy gray interpolation from 24-bit RGB to floating point CIELab (b. plotted).

Since trilinear interpolation appears to offer few if any advantages over tetrahedral for model-based LUTs and has dual disadvantages of consistently higher memory access requirements and inferior rendition of near-neutral colors from RGB,[3] we will focus on comparisons of more similar tetrahedral and binary dither interpolations.

Tetrahedral interpolation uses no more than 4 points and can yield better results along diagonals, but involves some complications in determining which tetrahedrons contain points to be interpolated.

Binary Dither Interpolation (BDI), as described here, performs another kind of linear interpolation. The number of BDI LUT accesses per conversion is related to the precision of interpolation, rather than data space dimensionality. For example, interpolation to 4-bit precision requires no more than 5 LUT entries for 3 or more dimensions. Two implementations, neighborhood mask dither and cached BDI, are constrained to no more than one LUT access for each conversion.

3. Example of Tetrahedral and BDI Calculations

Consider 24-bit RGB color {0xC8, 0x64, 0x96}. For a 17^3 lookup table T the high 4-bit nibbles of the input determine the sub-cube for this interpolation; in this case the origin is $T_{C,6,9}$ and the other end of the major axis is $T_{D,7,A}$ (Figure 3). The low 4-bit nibbles (8, 4, 6) are fractions within this sub-cube.

The resulting tetrahedral weights[1] are 4, 2, 2, and 8. The largest fraction is for red, so the axis is to $T_{D,6,9}$. The median fraction is for blue, so the minor diagonal is to $T_{D,6,A}$. Therefore the tetrahedral calculation is:

$$(4T_{D,7,A} + 2T_{D,6,A} + 2T_{D,6,9} + 8T_{C,6,9})/16.$$

The same color with binary fractions can be interpolated by BDI. Enumerated bit weights sum to 15/16 and origin is always included with a weight of 1/16. For each significant weight, the corresponding bit is added to the corresponding component of the origin. Thus the BDI calculation is:

$$(8T_{D,6,9} + 4T_{C,7,A} + 2T_{C,6,A} + T_{C,6,9} + T_{C,6,9})/16.$$

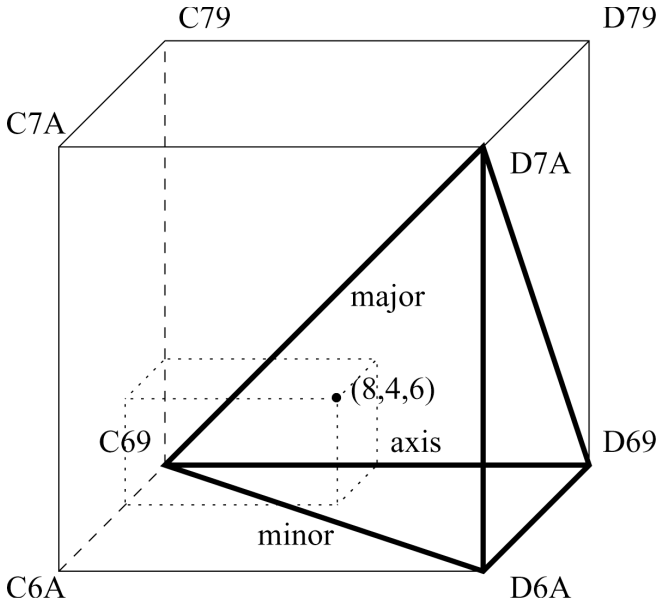


Figure 3. Example of tetrahedral interpolation.

The following table summarizes BDI processing for this example: BDI truncated BDI LUT weights inputs vertices

DBI weights	Truncated inputs 8	4	6	BDI LUT vertices
8	1	0	0	$T_{D,6,9}$
4	0	1	1	$T_{C,7,A}$
2	0	0	1	$T_{C,6,A}$
1	0	0	0	$T_{C,6,9}$

4. Single LUT Access Per Pixel

A well-known strategy for minimizing accesses to data in slow bulk storage is by use of a smaller high-speed cache. Cache implementation can be more complicated than the interpolation which it is intended to support. We have found a cache of 8 recently used LUT entries suffices to reduce artifacts. For a cache miss, it is desirable to first obtain the LUT entry which has the highest weight in the fully interpolated output. BDI weights are implicitly ordered, and

its highest weight is at least 8/16. Tetrahedral interpolation will require an additional sort of weights to prioritize LUT entries. Although tetrahedral interpolation has fewer cache misses on average than BDI, tetrahedral interpolation also has a higher worst case error when interpolating an identity 4 bit LUT, because the highest weight may be only 5/16. For single clock cycle execution, a simple hardware cache may need to employ a suboptimal strategy, such as preselecting a cache entry to be replaced before determining whether that entry may be used for the current pixel. Examples of color errors from LUT cache misses are shown in Figure 7. Since the relatively small color differences may not be apparent in print, we present difference images with differences multiplied by 16.

For this example, BDI reported 44 instances of LUT entries wanted but unavailable, since the number of LUT accesses is restricted to no more than one per pixel. Tetrahedral reported 25 cache misses. The largest color primary differences were 7/255 for BDI and 9/255 for tetrahedral, both in red. Largest of average absolute pixel color differences was 0.63/255 in red for BDI and 0.60/255 in blue for tetrahedral. Eliminating cache misses reduced BDI average absolute difference to 0.15/255 in blue.

BDI can also be implemented as a neighborhood dither, constrained to a single LUT access per pixel, by use of a following spatial bitmask:

$$\begin{pmatrix} 8 & 2 & 8 & 4 \\ 4 & 8 & 0 & 8 \\ 8 & 4 & 8 & 2 \\ 1 & 8 & 4 & 8 \end{pmatrix}$$

Neighborhood mask dither interpolation should be considered a spatial sampling process, to the extent that aliasing artifacts may be generated if used in conjunction with other sampling processes, such as halftoning by threshold array. Conversely, perturbation of colorant values by binary dither interpolation appreciably reduces the severity of "worm" artifacts generated by some error diffusion halftone algorithms. No dither occurs when truncated input bits are all zero, since conversion is exact. Results to date with neighborhood BDI followed by error diffusion have been very promising. At low resolution, dither of individual pixels is evident (Figure 6).

5. Conclusions

Mask dither color interpolation has been shown to work well in conjunction with error diffusion halftoning for inkjet printing at 600 dpi, causing no discernible artifact exacerbation and significantly reducing processing time for software based interpolation (about 30% reduction for a prototype filter to read, color convert, error diffuse and format an image for host-based inkjet printing). It also has applications in other control and data conversion processes that involve repetitive sampling when convergence to mean value is required only for local intervals.

Practical embedded applications include color conversions for error diffused printing and nonlinear color video displays. A neighborhood mask dither implemented in fewer than 100 gates for 3-dimensional 4-bit truncation will be eight times faster than uncached trilinear and four times faster than tetrahedral interpolation, where each is throttled by memory access speed for LUT entries. Cached implementations of tetrahedral interpolation can be competitive in speed but with considerably increased ASIC complexity and larger worst case errors.

Tetrahedral interpolation typically shows smaller average differences than BDI for RGB images converted with identity LUT. This may not generalize to nonidentity LUTs, and results with BDI are considered to have competitive quality on print samples evaluated to date. Note that neighborhood mask dither interpolated image of leaves Figure 8 may show moire patterns, depending upon how it is printed in the proceedings. Similar effects are to be expected with other dither interpolations.[4]

6. References

1. H. R. Kang. *Color technology for electronic imaging devices*. SPIE Optical Engineering Press, Bellingham, Wash., 1996.
2. R. Balasubramanian. Reducing the cost of lookup table based color transformations. *Journal of Imaging Science and Technology*, **44**(4):321–327, July/August 2000.
3. K. Kanamori. A study on interpolation errors and ripple artifacts of 3D lookup table method for nonlinear color conversion. In *Proceedings of SPIE*, volume **3648**, pages 167–178, San Jose, Calif., January 1999.
4. K. Spaulding. Method and apparatus employing mean preserving spatial modulation for transforming a digital color image signal. United States Patent No. 5,377,041, December 1994.

Biography

Steve Weed has worked for IBM, then Lexmark, since 1974. Since 1989, his primary research focus has been algorithms and architecture for image processing. Steve received his BSEE from Carlson College (now University) in 1969 and a Masters in Engineering from the University of Vermont in 1984.

```
for (c = 0; c < numcolors; c++)
    colorant[c] = 0;                                /* interpolate within unit cube of lut */

k = 16;                                             /* k is dither mask */
do {
    k = k / 2;                                      /* k = 0 on the 5th iteration */
    for (c = 0; c < 3; c++) {
        cmy[c] = 255 - rgb[c];
        if (k & cmy[c])
            cmy[c] = cmy[c]/16 + 1;
        else
            cmy[c] = cmy[c]/16;
    }
    if (k > 0)
        k1 = k;
    else
        k1 = 1;                                     /* non-zero weight for k */
    for (c = 0; c < numcolors; c++)
        colorant[c] += k1 * lut[cmy[0]][cmy[1]][cmy[2]][c]; /* lut access */
} while (k);

for (c = 0; c < numcolors; c++)
    colorant[c] = (colorant[c] + 8) / 16;
```

Figure 4: *Pseudo-code for BDI conversion of RGB to CMYK.*

```
/* rank the four LSBs of each tristimulus input component */
for (c = 0; c < 3; c++)
    cmy[c] = 0x0F & (255 - rgb[c]);

if (cmy[0] < cmy[1])
    axis = 1;
else
    axis = 0;
minor = 1 - axis;
if (cmy[axis] < cmy[2])
    axis = 2;
/* cmy[axis] is now >= other components */
major = 3 - (minor + axis);
if (cmy[minor] < cmy[major]) {
    minor = major;
    major = 3 - (minor + axis);
}
/* cmy[major] is now <= other components */
for (c = 0; c < 3; c++) {
    cmy[c] = (255 - rgb[c]) / 16;               /* tetrahedron origin by truncation */
    if (axis == c)
        cmyaxis[c] = cmy[c] + 1;                /* axis vertex */
    else
        cmyaxis[c] = cmy[c];
    if (major != c)
        cmyminor[c] = cmy[c] + 1;               /* minor diagonal vertex */
    else
        cmyminor[c] = cmy[c];
}
wmajor = cmy[major];                            /* major diagonal (neutral) weight */
wminor = cmy[minor] - cmy[major];               /* minor (rgb) diagonal weight */
waxis = cmy[axis] - cmy[minor];                 /* axis (cmy) weight */
worigin = 16 - cmy[axis];                       /* origin weight */
for (c = 0; c < numcolors; c++) {
    x =  wmajor  * lut[cmy[0] + 1 ][cmy[1] + 1 ][cmy[2] + 1 ][c];
    x += wminor  * lut[cmyminor[0]][cmyminor[1]][cmyminor[2]][c];
    x += waxis   * lut[cmyaxis[0] ][cmyaxis[1] ][cmyaxis[2] ][c];
    x += worigin * lut[cmy[0]     ][cmy[1]     ][cmy[2]     ][c];
    colorant[c] = (x + 8) / 16;
}
```

Figure 5. Pseudo-code for tetrahedral interpolation from RGB to CMYK.

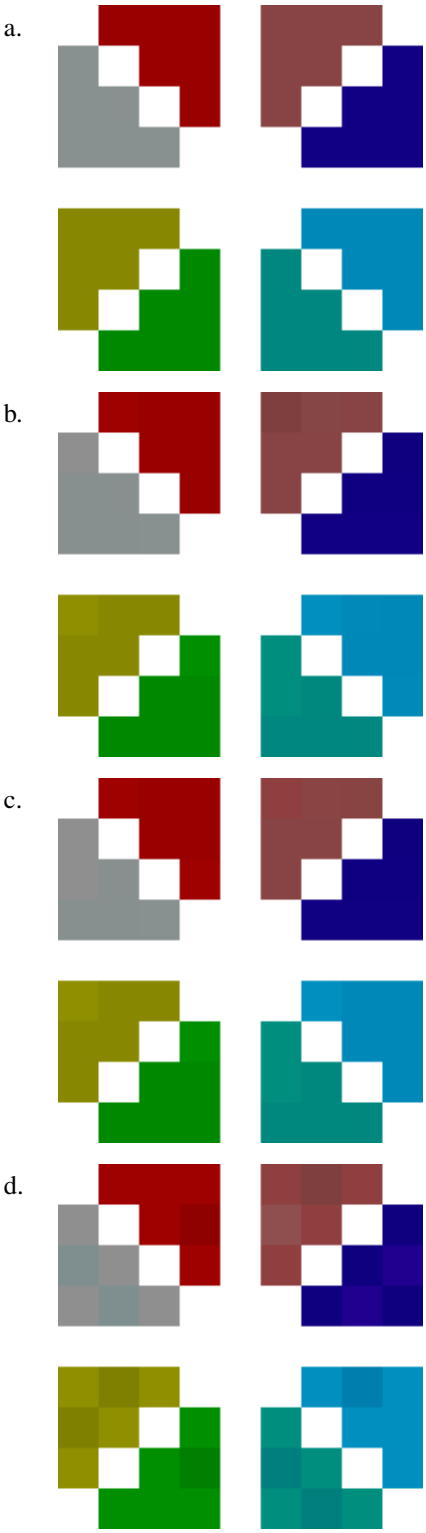


Figure 6. Effects of caching on interpolation using $17 \times 17 \times 17$ *identity 24-bit LUT: a. original* 9×9 *image; b. cached tetrahedral; c. cached binary dither; and d. neighborhood binary dither.*

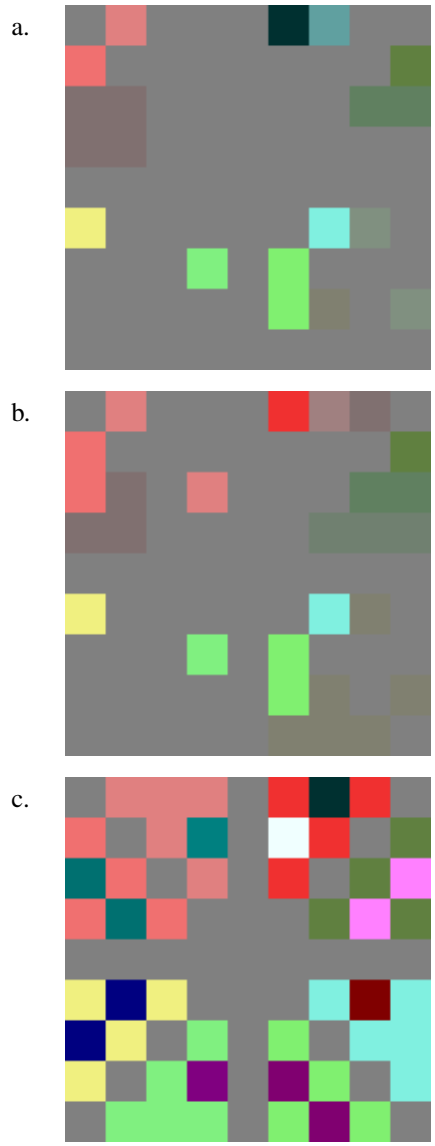


Figure 7. Effects of caching on interpolation using $17 \times 17 \times 17$ *identity 24-bit LUT: exaggerated difference images between the identity mapping and a. cached tetrahedral; b. cached binary dither; and c. neighborhood binary dither.*

a.

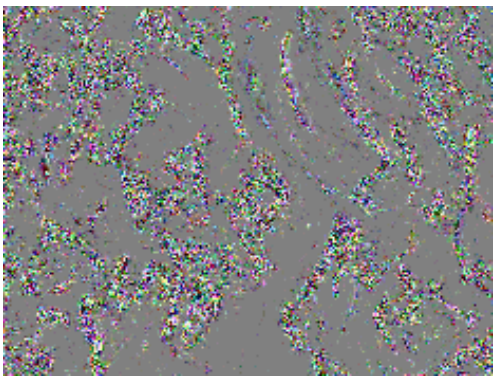
b.

c.

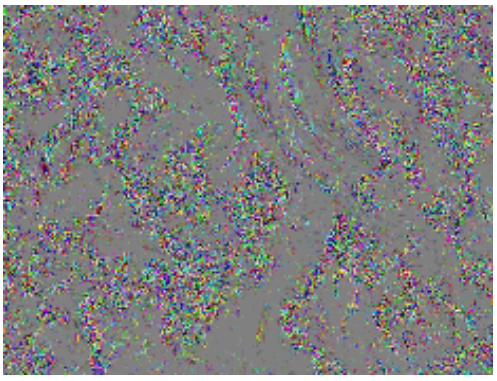
d.

Figure 8. Effects of caching on interpolation using $17 \times 17 \times 17$ *identity 24-bit LUT: a. original natural image; b. cached tetrahedral; c. cached binary dither; and d. neighborhood binary dither.*

a.



b.



c.

Figure 9. Effects of caching on interpolation using $17 \times 17 \times 17$ *identity 24-bit LUT: exaggerated difference images between the identity mapping and a. cached tetrahedral; b. cached binary dither; and c. neighborhood binary dither.*