chapter eleven

Efficient color transformation implementation

Raja Balasubramanian
R. Victor Klassen
Xerox Innovation Group

Contents

11.1 Introduction
11.2 Interpolation on regular lattices
  11.2.1 Regular lattice structures
    11.2.1.1 Lattice dimensions: power of two or one greater?
    11.2.1.2 Anamorphic lattice structures
  11.2.2 2-D Interpolation geometries
  11.2.3 3-D Interpolation geometries
    11.2.3.1 Trilinear interpolation
    11.2.3.2 Prism interpolation
    11.2.3.3 Pyramidal interpolation
    11.2.3.4 Tetrahedral interpolation
11.3 Interpolation on irregular lattices
11.4 Acceleration techniques
  11.4.1 Caching node values
  11.4.2 Caching output values
  11.4.3 Hashing
  11.4.4 Precomputing fixed-point quantities
  11.4.5 Eliminating multiplications
  11.4.6 Eliminating tests
  11.4.7 Data Formats

© 2003 by CRC Press LLC
11.1 Introduction

In many instances along the imaging path, images need their colors transformed. For example, images received in a device-dependent color space must be transformed to a device-independent color space. In many cases, these transformations are complex nonlinear functions, thus making it impractical to process large images in real time. A simple solution would be to precompute the transform for all possible digital inputs and store the corresponding outputs in a lookup table (LUT). This approach results in a perfectly accurate transformation (within limits of numerical computation), and the computational processing is relatively simple (i.e., only multidimensional lookups). However, the storage requirements quickly become prohibitive. For example, 8-bit RGB input requires a LUT of size $2^{24} = 16$ M; 8-bit CMYK input requires a LUT size of $2^{32} = 4$ G entries. For CMYK output, each entry is at least 32 bits, or 4 bytes. If the price of this quantity of memory is not an issue, the time to access it may be, as it is unlikely that the portions of interest will remain in the processor cache.

An alternative is to build a LUT with a sparser node sampling and to use multidimensional interpolation for input points that do not coincide with the LUT nodes. Constructing such a LUT involves first defining a lattice of nodes that partition the input color space into a set of smaller subvolumes. The desired color transformation is evaluated at these nodes, and the resulting output values are stored at the node locations. Algorithms for LUT construction are described in Chapter 5, so we will not elaborate on that aspect here. We will focus instead on LUT interpolation techniques.
an input point for which the color transformation is to be evaluated, multi-dimensional interpolation involves the following steps:

1. Find the subvolume to which the input color belongs.
2. Retrieve the nodes that define the subvolume.
3. Interpolate among the output values at these nodes to obtain the final output color.

The data may need additional processing before and after these three steps. Examples will be given in the following sections.

LUT interpolation involves a three-way trade-off among (a) LUT approximation accuracy, (b) storage and memory requirements, and (c) computational complexity. In the following sections, a number of LUT interpolation techniques are presented and compared in terms of the trade-offs among these three variables. The best technique and trade-off depends on the requirements and constraints of a given application. In the following discussions, we will use two-dimensional and three-dimensional input spaces for illustrative purposes, bearing in mind that extension to more dimensions is usually straightforward.

Sections 11.2 through 11.4 describe the basic LUT transformation and variants thereof. Sections 11.5 through 11.9 describe algorithms for efficiently processing large amounts of image data through a LUT transformation.

### 11.2 Interpolation on regular lattices

#### 11.2.1 Regular lattice structures

Define \( l_i \) to be a set of real-valued levels along the \( i \)th color dimension. A regular lattice \( L^m \) in \( m \)-dimensional color space is defined as the set of all points \( x = [x_1, \ldots, x_m] \) whose \( i \)th component \( x_i \) belongs to the set \( l_i \). Mathematically, the lattice can be expressed as

\[
L^m \{ x \in R^m | x_i \in l_i, i = 1, \ldots, m \} \quad \text{or} \quad L^m = \prod_{i=1}^{m} l_i \quad (11.1)
\]

where the second expression is a Cartesian product. If \( s_i \) is the number of levels in \( l_i \), the size of the lattice is the product \( s_1s_2\ldots s_m \). A regular three-dimensional lattice is shown in Figure 11.1. It partitions the three-dimensional input space into rectangular subvolumes. In the simplest form, the lattice levels \( l_i \) are spaced uniformly along the \( i \)th dimension, and \( s_1 = s_2 = \ldots = s_m = s \), resulting in a lattice size of \( s^m \). However, the lattice size and spacing of levels along each dimension can be varied to suit the characteristics of the function being approximated. Typical tables have from \( 8 \times 8 \times 8 \) to \( 33 \times 33 \times 33 \) nodes, with \( s_i \) being either a power of 2 or one greater than a power of 2. Chapter 5 contains experimental data that compares LUT accuracy for different lattice sizes.
The main advantage of a regular lattice is that the step of finding the subvolume to which an input color belongs is very simple and accomplished as follows. Assume a color \([x, y, z]\) in a three-dimensional space represented by \(n\) bits, and thus values ranging from 0 to \(C_{\text{max}} = 2^n – 1\). This color is to be transformed using a LUT of size \(m \times m \times m\). First, a nearby LUT node is located with indices \([X][Y][Z]\), given by

\[
X = \lfloor x \sigma \rfloor, \quad Y = \lfloor y \sigma \rfloor, \quad Z = \lfloor z \sigma \rfloor, \quad \sigma = m / C_{\text{max}}
\]

(11.2)

where \(\lfloor \rfloor\) denotes the rounding down operation. (Throughout this section, C-style indices are used, i.e., the first index of the table runs from 0 to \(m - 1\).)

If the color \([x, y, z]\) coincides with node \([X][Y][Z]\), the output values stored at this node are simply looked up. Otherwise, the desired output is obtained by interpolating with the seven neighboring nodes, which have one or more of the LUT indices \([X][Y][Z]\) increased by one. These eight nodes form the rectangular subvolume enclosing the input point. The highest numbered nodes (index \(m - 1\)), when used, never require interpolation. One must either test to avoid interpolating with the next set, ensure that a set of dummy values (which will be multiplied by 0) are allocated beyond the last value, or risk reading beyond the end of allocated memory.

**Example.** Given an 8-bit, three-separation input space and a \(17 \times 17 \times 17\) uniform lattice, we have \(\sigma = 17/255 = 1/15\). An input color \([23, 36, 190]\) will locate LUT indices \([1][2][12]\). The output is obtained by interpolating among the nodes at \([1][2][12], [1][2][13], [1][3][12], [1][3][13], [2][2][12], [2][2][13], [2][3][12], \) and \([2][3][13]\). Any given input color not on a node \([X][Y][Z]\) is a fraction \(t, u,\) and \(v\) of the way to the next node in the \(x, y,\) and \(z\) directions, respectively. The fraction \(1/\sigma\) gives the distance between nodes. The fractional distance of \(x\) between nodes at \(X\) and \(X + 1/\sigma\) is \(t = x\sigma - X\).

11.2.1.1 Lattice dimensions: power of two or one greater?

There is a long-standing debate over whether the lattice dimensions should be a power of two or one greater. This is largely a question of efficiency. As we shall see, the difference is very small if both are implemented as efficiently as possible. For concreteness, we shall compare \(16^3\) with \(17^3\); however, any
such pairing follows the same argument — and one may leave the vicinity of a power of two altogether without paying the imagined penalty.

The key advantage of $16^3$ is in addressing the $(i, j, k)$ entry using the (C language) expression

\[
\text{base} + (((i << 4) + j) << 4) + k,
\]

rather than the expression needed for the $17^3$ case,

\[
\text{base} + (((i \times 17) + j) \times 17) + k,
\]

which involves two multiplications, normally slower than the shifts. Thus, once the values of $i$, $j$, and $k$ are known, indexing is faster using $16^3$. However, these quantities are derived from color separation values, and obtaining them and the interpolation parameters ($t$, $u$, and $v$) is more difficult for a power-of-two LUT.

From the geometry of the table, it clearly contains nodes at 0 and max, with others uniformly spaced in between. Interpolation is between pairs of adjacent nodes, and along any axis, the number of pairs of adjacent nodes is a power of two. If the table size is $17^3$, the index of the lower node is found in the high-order bits of the input color separations, and the interpolation parameters are the low-order bits of the same input color separations. Thus, to obtain $i$ and a fixed point value of $t$, one computes

\[
i = x >> 4
\]

\[
t = x & 0xf;
\]

whereas, for a table size $n$ that is not a power-of-two-plus-one,

\[
i = x / (n - 1),
\]

\[
t = x - (n - 1) i.
\]

So it appears that the debate is more about choosing where the multiplications and divisions occur than how many of them there are. As it happens, none of these calculations is necessarily optimal. The number of possible values of $x$ is small (somewhere in the 256 to 4096 range), and so the values of $i$ and $t$ may be obtained by a simple LUT from the value of $x$, eliminating the division and multiplication in the power-of-two-plus-one case. Note, however, that it is not really the value of $i$ that is needed, but the quantity $n \times m \times i$, which could just as easily be stored in the lookup table as $i$. The addressing arithmetic is equivalently

\[
\text{base} + i \times n \times m + j \times m + k,
\]

so the table indexed by $x$ can contain the appropriate values of $t$ as well as the values of $i \times n \times m$, while the table indexed by $y$ contains the values of $u$ and $j \times m$, with the table indexed by $z$ containing $k$ and $v$.

Thus, both table address calculation and parameter calculation are obtained through a single lookup operation (returning a two-element structure), independent of the table size. With this approach, there is no advantage
to using either a power of two or one greater; any size has the same computational cost. There is one potential advantage to having an odd-sized table, and that is for input color spaces with a neutral-axis aligned dimension (such as L*ab* or YCrCb), one may arrange to have the neutral axis exactly represented along one column of nodes while maintaining node symmetry about this axis.

### 11.2.1.2 Anamorphic lattice structures

Balasubramanian has proposed using an anamorphic lattice, a variation of the regular lattice structure that offers a more efficient sampling of nodes.\(^1\) The idea is based on the observation that interpolating from an \(m\)-dimensional input space to an \(n\)-dimensional output space consists of \(n\) independent interpolations from \(m\)-dimensional input to one-dimensional output. Consider, for example, a LUT transform from display RGB to printer CMY. The output cyan value is calculated by interpolating the cyan values at the surrounding nodes. Similarly, the magenta and yellow outputs are independently calculated by interpolating the nearby magenta and yellow node values, respectively. This being the case, there is no reason to use the same lattice to interpolate the C, M, and Y. For example, one might expect that C has a strong dependence on its complementary input R and a weak dependence on G and B. Likewise, M may show stronger dependence on input G than on R and B. Hence, a suitable lattice for interpolating C would have a dense sampling along R and coarse sampling along G and B. Equivalently, the M signal requires a finer sampling along G than along R and B. The idea is demonstrated in Figure 11.2 for a 2-D transform from R-G to C-M. Experimental results have shown that the average \(\Delta E^*_{ab}\) LUT error is reduced from 2.1 using a standard lattice to 1.1 using an anamorphic lattice of the same size. Thus, an anamorphic lattice improves the size-accuracy trade-off. Note, however, that the anamorphic lattice requires finding the enclosing subvolume for each output separation, thus incurring a small computational overhead.

### 11.2.2 Two-dimensional interpolation geometries

Given a rectangular subvolume and its vertex nodes that surround the input point, the interpolation geometry specifies which subset of those nodes is used to calculate the final color. The two-dimensional geometries of bilinear and triangular interpolation will be described first, as these are used as building blocks for the three-dimensional geometries commonly used in color transformations.

Bilinear interpolation is the natural extension of linear interpolation to two dimensions. It is based on the assumption that the function being approximated is “close enough” to linear between two nodes. That is, it has constant derivative between those nodes. If the values at nodes \(n_0\) and \(n_1\) are \(p_0\) and \(p_1\), respectively, then the value at a location that is a fraction \(t\) of the way from \(n_0\) to \(n_1\) is given by \(p_0 + (p_1 - p_0)t\). To compute an interpolant a fraction \(t\) of the
way along a first dimension of a square and a fraction \( u \) of the way along a second dimension of the square, begin by computing two intermediate points, a fraction \( t \) along the edges that traverse the first dimension. Then compute the interpolant a fraction \( u \) of the way from the first to the second of those intermediate points. Either dimension may be interpolated first with identical results, as shown below.

### \( t \) first:

\[
\begin{align*}
 p_0 &= p_{00} + t(p_{01} - p_{00}) \\
 p_1 &= p_{10} + t(p_{11} - p_{10}) \\
 p &= p_0 + u(p_1 - p_0) = p_{00} + t(p_{01} - p_{00}) + u(p_{10} + t(p_{11} - p_{10}) - (p_{00} + t(p_{01} - p_{00}))) \\
 &= p_{00} + t(p_{01} - p_{00}) + u(p_{10} - p_{00}) + tu(p_{11} - p_{10} - p_{01} + p_{00})
\end{align*}
\]

### \( u \) first:

\[
\begin{align*}
 p'_{0} &= p_{00} + t(p_{10} - p_{00}) \\
 p'_{1} &= p_{01} + u(p_{11} - p_{01}) \\
 p &= p'_{0} + u(p'_{1} - p'_{0}) = p_{00} + u(p_{10} - p_{00}) + t(p_{01} + u(p_{11} - p_{01}) - (p_{00} + u(p_{10} - p_{00}))) \\
 &= p_{00} + t(p_{01} - p_{00}) + u(p_{10} - p_{00}) + tu(p_{11} - p_{10} - p_{01} + p_{00})
\end{align*}
\]

Figure 11.2 shows this operation for the case where the \( t \) interpolation is done first.

Triangular interpolation, applied to right triangles with two axis-aligned edges, employs the additional assumption that the function being
approximated has zero cross derivatives. That is, the derivative with respect to \( u \) is the same irrespective of the value of \( t \). With reference to Figure 11.4, given nodes \( n_{00}, n_{01} \) and \( n_{11} \) with values \( p_{00}, p_{01} \) and \( p_{11} \) corresponding to \( t = u = 0; t = 0, u = 1; \) and \( t = u = 1 \), respectively, begin with \( p_{00} \), add \( (p_{01} - p_{00})u \), to find a value corresponding to \( u \) of the way from \( p_{00} \) to \( p_{01} \). Then add \( (p_{11} - p_{01})t \) to find a value corresponding to \( t \) of the way from there to the line containing \( p_{11} \). If both of the assumptions — linear function with zero cross derivatives — hold, the result of triangular interpolation is the same as that of bilinear.

\[
\begin{align*}
p_0 &= p_{00} + t(p_{01} - p_{00}) = p_{00} + t(p_{11} - p_{01}) \\
p_1 &= p_{10} + t(p_{11} - p_{01}) \\
p &= p_0 + u(p_1 - p_0) = p_{00} + t(p_{11} - p_{01}) + u(p_{10} + t(p_{11} - p_{01}) - (p_{00} + t(p_{11} - p_{01}))) \\
&= p_{00} + t(p_{11} - p_{01}) + u(p_{01} - p_{00})
\end{align*}
\]

### 11.2.3 Three-dimensional interpolation geometries

Three-dimensional interpolation geometries include trilinear (all nodes), prism, pyramidal, and tetrahedral, in order of decreasing numbers of nodes used. The vertices of the interpolation solids for the various geometries are given in Table 11.1, with the labeling convention used in Figure 11.5. In all of these geometries, the facets of the solids within which interpolation is performed are either squares or triangles, hence the reliance on bilinear and triangular interpolation. From a computational standpoint, interpolating within a square is more expensive than interpolating within a triangle; this
is one of the motivations of the various geometries. At the same time, the fewer cutting planes subdivide the cube, the fewer tests are required to select the interpolation solid. From a qualitative standpoint, geometries that use more nodes will perform stronger local averaging of the function being approximated, while geometries with fewer nodes will more closely follow the curvature of the function. The relative performance of the geometries will thus depend on the nature of the function and noise characteristics. The two most commonly used geometries are trilinear and tetrahedral — the
three-dimensional analogies of bilinear and triangular — representing the two extremes. Each of the geometries is described in detail below.

11.2.3.1 Trilinear interpolation

Trilinear interpolation uses all eight nodes and is the natural extension of linear interpolation to three dimensions. This is shown in Figure 11.6. To compute the interpolant at fractions \( t, u, \) and \( v \) along three respective dimensions, first compute four intermediate points \( (n_{000}, n_{001}, n_{010}, n_{100}) \) that are \( t \) of the way along the edges varying in the first dimension. Then use those four points to define a square and perform bilinear interpolation as described in the previous section. From the square, two more points are computed \( (n_0 \text{ and } n_1) \); these are used to compute the final point \( n \).

Using the notation \( L^3(x_1...x_n) \) to refer to linear interpolation in \( n \) dimensions, \( L^3(t, u, v) \) then denotes trilinear interpolation. It may be expressed recursively in terms of lower-order interpolation as:

![Figure 11.6 Trilinear interpolation.](image-url)
\[ L^3(t, u, v) = L^2_0(t, u) + (L^2(t, u) - L^2_0(t, u)) v \]
\[ = L^1_{00}(t) + (L^1_{01}(t) - L^1_{00}(t))u + \{L^1_{10}(t) + (L^1_{11}(t) - L^1_{10}(t))u \}
- \{L^1_{00}(t) + (L^1_{01}(t) - L^1_{00}(t))u \} v \]
\[ = p_{000} + (p_{001} - p_{000}) t + (p_{010} + (p_{011} - p_{010}) t - (p_{000} + \(p_{001} - p_{000}) t) u + (p_{100} + (p_{101} - p_{100}) t + (p_{110} + (p_{111} - p_{110}) t) u - [p_{000} + (p_{001} - p_{000}) t) u + (p_{010} + (p_{011} - p_{010}) t - (p_{000} + (p_{001} - p_{000}) t) u] v \] (11.3)

Computationally,
\[ L^3(t, u, v) = p_{000} + \Delta^0_{00} t + \delta^0_{00} u + (\Delta^0 + \Delta^0 v) t + \delta^0_{01} u v \]
\[ \Delta^0_{00} = p_{001} - p_{000}; \quad \Delta^0_{10} = p_{101} - p_{100}; \quad \Delta^0_{11} = p_{111} - p_{110}; \quad \Delta^0 = p_{111} - p_{110}; \]
\[ \Delta^v_{00} = p_{010} - p_{000}; \quad \Delta^v_{10} = p_{110} - p_{100}; \quad \Delta^v = p_{100} - p_{100}; \quad \Delta^v = p_{111} - p_{110}; \]
\[ \Delta^u_{00} = \Delta^u_{10} - \Delta^u_{00}; \quad \Delta^u = \Delta^u_{10} - \Delta^u_{00}; \quad \Delta^uv = \Delta^v - \Delta^u v; \quad \Delta^uv = \Delta^v - \Delta^u v. \]
\[ \delta^u_{00} = \Delta^u_{00} + \Delta^u_{0} t. \quad \delta^u_{01} = \Delta^u_{00} + \Delta^u_{0} t. \] (11.4)

This requires seven multiplications and seven additions, with twelve pre-computed coefficients stored at each node.

### 11.2.3.2 Prism interpolation

Prism interpolation splits the cube in half along a face-diagonal, as shown in Figure 11.7. When interpolating in RGB or like color spaces, the cutting plane should be chosen to contain the neutral \((R = G = B)\) axis so that

**Figure 11.7** Prism interpolation.
interpolation along this axis can be carefully controlled. An initial test is required to determine in which prism the point lies, and then interpolation is performed using only six, rather than all eight, points of the cube. As shown in Figure 11.7, the point is first projected onto each of the two triangular faces, and triangular interpolation is performed within these faces using fractional distances \( t \) and \( u \). The resulting two points are interpolated linearly using the fractional distance \( v \). These two steps can be combined as

\[
P^3_{t,u}(t, u, v) = p_{000} + (p_{011} - p_{001}) u + (p_{001} - p_{000}) t + \left[ p_{100} + (p_{111} - p_{101}) u + (p_{101} - p_{100}) t - (p_{000} + (p_{011} - p_{001}) u + (p_{001} - p_{000}) t) \right] v
\]

(11.5)

Computationally,

\[
P^3_{t,u}(t, u, v) = s_5 + \left[ p_{100} + s_2 + s_4 - s_3 \right] v
\]

(11.6)

where

\[
\begin{align*}
  s_1 &= (p_{011} - p_{001}) u = \Delta_1 u \\
  s_2 &= (p_{111} - p_{101}) u = \Delta_2 u \\
  s_3 &= (p_{001} - p_{000}) t = \Delta_3 t \\
  s_4 &= (p_{101} - p_{100}) t = \Delta_4 t \\
  s_5 &= p_{000} + s_1 + s_3
\end{align*}
\]

Prism interpolation requires five multiplications and six additions with nine coefficients stored at each node. The reduced multiplications come at the expense of a single test.

11.2.3.3 Pyramid interpolation

Figure 11.8 shows the geometry of pyramid interpolation. It uses one corner (which for RGB-like spaces should be either the node \([X][Y][Z]\) or \([X+1][Y+1][Z+1]\)) as the apex of three pyramids, the bases being the three faces that share the diagonally opposite vertex. Two tests are required to determine in which pyramid the point lies, and then interpolation is performed using five points of the cube.

\[
P^3_{t,u,v}(t, u, v) = p_{000} + s_1 u + s_2 t + (s_3 + s_4 u) v
\]

(11.7)

where

\[
\begin{align*}
  s_1 &= p_{010} - p_{000} \\
  s_2 &= p_{110} - p_{100} \\
  s_3 &= p_{001} - p_{000} \\
  s_4 &= p_{011} - p_{100} - p_{001} + p_{000}
\end{align*}
\]

\[
P^3_{u,v,t}(t, u, v) = p_{000} + s_1 v + s_2 u + (s_3 + s_4 v) t
\]

(11.8)

where

\[
\begin{align*}
  s_1 &= p_{100} - p_{100} \\
  s_2 &= p_{111} - p_{101} \\
  s_3 &= p_{001} - p_{000} \\
  s_4 &= p_{101} - p_{001} - p_{100} + p_{000}
\end{align*}
\]

\[
P^3_{u,v,t}(t, u, v) = p_{000} + s_1 t + s_2 v + (s_3 + s_4 u)
\]

(11.9)

where

\[
\begin{align*}
  s_1 &= p_{001} - p_{000} \\
  s_2 &= p_{111} - p_{011} \\
  s_3 &= p_{010} - p_{000} \\
  s_4 &= p_{011} - p_{010} - p_{001} + p_{000}
\end{align*}
\]
Pyramidal interpolation requires four multiplications, with five values stored at each node.

11.2.3.4 Tetrahedral interpolation
Due to its computational simplicity, tetrahedral interpolation is a very popular choice for three-dimensional color transformations. The most common form divides the two prisms of prism interpolation into three tetrahedra each, as shown in Figure 11.9. All six tetrahedra share one common edge: the main diagonal of the cube. In the case of RGB spaces, the diagonal axis should coincide with the neutral (R = G = B) axis. The remaining edges of the tetrahedra join the neutral vertices to vertices diagonally opposite them on shared faces. Tetrahedral interpolation requires, on average, 2.5 comparison tests (i.e., 2 tests or 3 tests, both with 50% probability) to find the enclosing tetrahedron. The code structure is given below:

\[
\begin{align*}
&\text{If } t < u \; // \; \text{lower front prism} \\
&\quad \text{If } t > v \\
&\quad \quad \text{Interpolate within tetrahedron 1} \\
&\quad \text{Else} \\
&\quad \quad \text{If } u < v \\
&\quad \quad \quad \text{Interpolate within tetrahedron 2} \\
&\quad \quad \text{Else} \\
&\quad \quad \quad \text{Interpolate within tetrahedron 3}
\end{align*}
\]
Else // upper, back prism
    If \( t < v \)
        Interpolate within tetrahedron 4
    Else
        If \( u < v \)
            Interpolate within tetrahedron 5
        Else
            Interpolate within tetrahedron 6.
\}

The case where \( t < u, t > v \) (i.e., tetrahedron 1) is shown in Figure 11.10. Interpolation among the four points within the enclosing tetrahedron is given by

\[
P_0(t, u, v) = p_{000} + s_t t + s_u u + s_v v \tag{11.10}
\]

The interpolation weights \( s_t, s_u, s_v \) depend on which tetrahedron contains the input point, and they are determined by comparing the magnitudes of \( t, u \), and \( v \), as shown above and in Table 11.2. Table 11.3 gives the differences \( \Delta_t \) used in Table 11.2 (and hence the values of \( s_t, s_u, \) and \( s_v \)). The 12 differences correspond to the 12 edges of the cube. Assuming the differences are all precomputed, tetrahedral interpolation requires 3 additions and 3 multiplications, with 13 values stored at each node. It is essentially three triangular interpolations: one per dimension. On most processors, memory fetch times dominate in tetrahedral interpolation.

Other variants of tetrahedral interpolation exist; for example, it is possible to partition the rectangular volume into only five tetrahedra. Details are given Kang’s book.\(^3\)
11.3 Interpolation on irregular lattices

In the most general case where the lattice comprises an arbitrary set of points in three-dimensional space, it is possible to construct a tetrahedral partition of the space with these points using the DeLaunay tetrahedralization algorithm.

This generalizes for arbitrary dimensions as well. The advantage of this approach is the flexibility with which the lattice nodes can be positioned.
in color space. However, the step of retrieving the enclosing tetrahedron becomes far more complex than in the case of regular lattices. Furthermore, the interpolation step is also more complex, involving tetrahedral inversion (see Chapter 5, Section 5.4.5). For this reason, the Delaunay partition is rarely used in applications that require efficient real-time mapping through LUTs, and it will not be discussed further.

A special case of an irregular lattice is the sequential lattice, a structure that offers more flexibility than a regular lattice but at a modest increase in computational cost. The general sequential interpolation architecture is described in detail in Chapter 5 and will not be presented again here. We instead focus on the particular example of sequential linear interpolation (SLI). The SLI lattice is shown in two dimensions in Figure 11.11. Along the first dimension $c_1$ are node levels $c_{1,1}, \ldots, c_{1,N}$. At each such node level, a set of nodes is positioned along $c_2$. The interesting feature of SLI is that the node separation along $c_2$ can vary depending on the location along $c_1$. With reference to the diagram on the right in Figure 11.11, interpolation for an arbitrary input point $P$ is performed as follows:

- Project $P$ onto the $c_1$ axis and locate the two neighboring levels $c_{1,i}, c_{1,i+1}$.
- Project $P$ onto the line $c_{1,i}$ and perform linear interpolation between points $D_1$ and $D_2$, resulting in intermediate value $D_a$.
- Project $P$ onto $c_{1,i+1}$ and perform linear interpolation between $D_3$ and $D_4$, resulting in $D_a$.
- Perform 1-D interpolation between $D_a$ and $D_b$ to obtain the final output.

Note that this process is very similar to bilinear interpolation, described in Section 11.2.2. Indeed, if the node placements along $c_2$ are independent of $c_1$, the SLI lattice reduces to the standard regular lattice, and the algorithm reduces to bilinear interpolation. The more general and interesting case

![Sequential linear interpolation in two dimensions.](image-url)
where interpolation along $c_2$ depends on both $c_2$ and $c_1$ enables a more efficient placement of nodes in those regions where the function being approximated exhibits greatest curvature. Optimal design of an SLI lattice to suit the characteristics of the underlying transformation is described in detail by Chang et al.\textsuperscript{6} and Agar et al.\textsuperscript{7}

The algorithm just described extends readily to an arbitrary number of dimensions, with node level placement along the $i$th dimension depending on the input values along the previous $i-1$ dimensions. In the common case of three dimensions, SLI requires seven multiplications and seven additions and is hence computationally equivalent to trilinear interpolation. However, SLI requires prestoring more fractional quantities. This is because two of the quantities $t, u, v$ in Equations 11.3 and 11.4 vary depending on the position of the input point in the 3-D table. Normally, these are precomputed as described in Section 11.4. The chief advantage of SLI lies in its more efficient use of table space, allowing a table of a given number of nodes to more accurately follow the curvature of the function being approximated.

11.4 Acceleration techniques

Lookup table interpolation is much faster than inverting measured data or evaluating a printer model. However, applying the techniques below may speed it up further. Not all acceleration techniques are applicable to all applications; some are architecture dependent, while the advantage of others depends on the nature of the input. Each should be evaluated in the context of the intended application.

11.4.1 Caching node values

For some machines, three-dimensional lookup is expensive, and if 13 values are stored at a node, pulling in these values from memory only adds to the expense. In such cases, a small cache of node values may be beneficial. Typical images contain long runs (tens or even hundreds of pixels) of colors similar enough to be within a node of each other. A cache containing seven nodes is generally sufficient for such purposes. In such an implementation, the cache is checked before the full three-dimensional lookup is performed. On most modern architectures, these values will be cached in the hardware cache. Artificially checking a cache may result in reduced performance in the presence of a hardware cache. One might consider reducing the precision of the values stored at the nodes if the hardware cache is too small to hold seven nodes.

11.4.2 Caching output values

For most machines, the computations, even for tetrahedral interpolation, are more expensive than simple caching schemes. The challenge lies in making the amortized cost of cached interpolation less expensive than reinterpol-
tion. Some number $n_c$ of input–output pairs is stored in a table, with a simple hashing scheme determining the location in the table at which the color pair containing input color $c_i$ is stored.

To convert a new color, apply the hash function $h(n_c, c_i)$ to obtain an address in the table. If the color pair containing $c_i$ is in that location, return the output color of that pair. Otherwise, perform the normal interpolation function, and store the input and output colors in a color pair at the computed location.

The value of caching output values depends on colors repeating before they are overwritten in the table. Only large/high-resolution images are important to the discussion of efficiency. These images typically have many regions at least 0.1% of their size containing nearly constant colors. Even highly textured regions typically vary only along a single line in color space. In a 300-dpi scan of a $4 \times 5$ inch image, a region of 0.1% of the image is 1800 pixels. Due to film grain, scanner noise, and other sources of noise, even a region of apparently constant color will not have exactly constant colors throughout such a region. However, it is not uncommon to find regions of this size containing fewer than 300 distinct colors.

The size of the table used for caching should typically be at least large enough to hold a scanline or two of the typical image. The chance of reusing a color increases substantially if the reuse may happen anywhere in a multi-scanline region rather than in a single scanline. Very large tables have two disadvantages: besides requiring a large quantity of memory, successively needed values are unlikely to be in the hardware cache because, by design, similar values are scattered throughout the table.

### 11.4.3 Hashing

Hashing is discussed at length in Aho, Sethi, and Ullman.\textsuperscript{8} Conventionally, collisions (when a second value maps to an already-used location) are handled using linked lists or secondary hash schemes. For caching color conversions, it is more efficient simply to replace the old value with the newly computed one, as described above. An ideal hash function will map all the distinct colors found in a few scanlines of an image to different locations in the table. Designing such a perfect function requires knowledge of the colors in the image. However, a few guiding principles can be used to design good enough hash functions.

- Use only additions, shifts, and logical operations.
- Assume that the few high-order bits of each separation will be constant (hence, they contribute no new information).
- Assume that variation is primarily in luminance (RGB variations are correlated; L*\textsuperscript{a}b* variations decorrelated).

A simple hash function based on a mapping $hashpjw$ given in Aho et al.\textsuperscript{8} for 8-bit RGB color is $h(n_c, c_i) = (((r_i << 4) + g_i) << 4) + b_i) \mod n_c$. 
11.4.4 Precomputing fixed-point quantities

To compute any of Equations 11.2 through 11.10 requires computing values of \( t, u, \) and \( v \), normally in floating point, and multiplying them by the appropriate differences. For example,

\[
X = \lfloor x\sigma \rfloor \quad t = x\sigma - X \tag{11.11}
\]

The value of \( \sigma \) is a constant for a given lookup table size; a table of \( C_{\text{max}} \) entries will give the values of \( X \) and \( t \) in Equation 11.11. This sacrifices no precision, as the subsequent operations may still use floating point.

The basic operation of linear interpolation involves multiplying a difference by a parameter in the \([0, 1)\) range and then adding it to a base value. Essentially the same operations appear throughout Equations 11.2 through 11.10.

\[
\text{Lerp}(a, b, t) = a + (b - a) t \tag{11.12}
\]

Values \( a \) and \( b \) may be viewed as being fixed-point numbers with \( n \) bits fraction. In this representation, they range from 0 to \( 1 - 1/C_{\text{max}} \). Converting Equation 11.12 from floating point to fixed point with \( Z \) fraction bits,

\[
\text{Lerp}(a, b, T) = a + ((D T + (1 << (Z - 1))) >> Z) \tag{11.13}
\]

with \( D = (b - a), \ T = [t(1<<Z) + 0.5] \).

All of the variables in Equation 11.12 are originally retrieved from lookup tables. Hence, replacing the lookup table entries with the corresponding values in Equation 11.13 eliminates all of the floating to fixed point conversions from the interpolation. On typical machines, conversions to and from floating point are expensive, so this is a significant savings. All of the fixed point values in Equation 11.12 have known ranges: \( 0 \leq a < 1, -1 + 1/C_{\text{max}} \leq b - a \leq 1 - 1/C_{\text{max}}, 0 \leq t < 1 \). With \( C_{\text{max}} = 2^8 \), a value of \( Z = 23 \) will not cause overflow on a 32-bit machine in the expression \((D T) >> Z\).

11.4.5 Eliminating multiplications

It is possible to eliminate all of the multiplications at the expense of an extra table lookup. Whether this speeds up the interpolation will depend on the machine architecture. On a typical RISC machine, multiplications are implemented as multiple instruction microcode subroutines, making them more expensive than table lookups. On a machine with relatively slow memory and a fast multiply instruction, leaving in the multiplications may be faster.

In Equation 11.13, \( D \) takes one of \( 2C_{\text{max}} - 1 \) possible values. It is multiplied by \( T \), which takes on one of \( C_{\text{max}} \) values. \( T \) itself and \( D \) are retrieved from lookup tables; there is no reason why the value in the table from which \( T \) was retrieved should not, instead, contain a pointer to a multiplication table,
or an index into a table of multiplication tables. Assuming \( n = 8 \) bits, there would be 256 tables of 511 entries, for a total of just under 128k entries. The entries themselves could be either short words (to save space) or full words (for faster computation).

The expression then becomes

\[
\text{Lerp}(a, D, T) = a + ((M[D, T] + (1 << (Z – 1))) >> Z)
\]

with \( M[j, k] = j \cdot k \), and with the precomputed values of \( D \) offset by \( C_{\text{max}} \) so that the table indices are always positive. The rounding and final shift can also be folded into the table; however, for tetrahedral interpolation, there are three differences multiplied by three parameters and added to a node value. In that case, \( (1 << (Z – 1)) \) should be added only once, as in

\[
\]

with \( M'[j, k] = M[j, k] + (1 << (Z – 1)). \)

### 11.4.6 Eliminating tests

On modern machines, frequent branches are expensive because they require the instruction pipeline to be flushed. Some machines have a large enough instruction cache so that instructions may be fetched for both sides of a branch, up to two or even three branches, in which case the penalty is negligible until the number of branches without intervening computation reaches three or four. Thus, it is sometimes, but not always, advisable to eliminate tests. For tetrahedral interpolation, recall from Section 11.2.3.4 that tests are required to determine one of six enclosing tetrahedra.

In all six cases, the expression to be evaluated is the same; only the operands \( s_t, s_u, \) and \( s_v \) differ. These are taken from a set of 12, depending on the case. If a node is defined as containing the value of \( p_{00} \) and an array containing the 12 edge-differences, the case analysis selects which of the 12 array indices are used to obtain \( s_t, s_u, \) and \( s_v \). Each of the tests in the case analysis is decided on the basis of the sign of the difference between two of \( t, u, \) and \( v. \) Therefore, the case analysis may be replaced by

\[
D_{tu} = t - u; \quad D_{tv} = t - v; \quad D_{uv} = u - v
\]

where \( s \) is the number of bits in the machine word holding \( D_{tu}, D_{tv}, \) and \( D_{uv}. \) An eight-entry lookup table of three-tuples will give the indices of \( s_t, s_u, \) and \( s_v. \) This completely replaces the tests with straight-line code, making the code smaller and possibly faster. Whether the code is faster depends on whether the extra shifts and ors of values likely to be in registers is more or
less expensive than the average 2.5 tests. The shift and mask combination can also be incorporated into a LUT, which may help as well. This depends on the trade-off between two ALU operations and a memory fetch. Fully optimized, the version with 2.5 tests ran in approximately 71 ns per color on a 500-MHz PowerPC G3; the best version without tests took about 79 ns, 10% slower.

11.4.7 Data Formats

The format of the data in the table may make as much difference as many of the other acceleration techniques. Most of the processing involved in transforming large images is memory-bound. That is, the time required on current architectures tends to be dominated by the number of memory accesses. Typical processors have 32- or 64-bit memory buses, which means that a 32-bit (or 64-bit) memory fetch costs no more than a single-byte fetch. With this in mind, an image stored as packed pixels will generally be more efficient to process than one with separately stored channels. Similarly, a multidimensional LUT containing all of the data for a color at each node enables more efficient processing than one table per output separation.

Many processors penalize conversion to and from floating point. Lookup tables are typically computed in floating point, but color images are most commonly stored as arrays of fixed-point integers. Output devices invariably require integer-valued pixels. Hence, the LUT should be converted to fixed point once and then all pixel operations carried out in fixed point.

Besides memory access time, packed pixels provide acceleration opportunities, as multiple additions or averaging operations may be carried out in parallel across separations. Specialized media processing instructions (VIS instructions on the SPARC processor, MMX on Intel, AltiVec on PowerPC) often permit the operations for a linear interpolate (LERP, above) to be computed in parallel on two or four sets of operands (with the same value of the parameter) in the time normally required to do one such operation, provided the operands are formatted appropriately.

11.5 Color transforming palettized images

A palettized image is one in which pixel colors are limited to a relatively small palette of colors. A common palette size is 256, used in 8-bit displays. The process of designing the optimal palette and assigning image pixels to the most appropriate palette color is commonly referred to as color quantization and is covered in detail in another chapter. The most efficient way to apply a color transformation to a palettized image is to apply the transformation a priori to the palette colors and store the palette in the new transformed color coordinates. The color transformation problem is thus translated to one of color quantization.
11.6 Subsampled color correction

11.6.1 Introduction

Some applications use luminance/chrominance-aligned color spaces. For example, YCrCb allows color/brightness/saturation adjustments at the user interface, and the color fax standard uses L*a*b*, which is also luminance/chrominance based. To avoid ambiguity with yellow, assume that the input is a luminance/chrominance space, denoted LCrCb, a surrogate for other luminance/chrominance spaces.

It is well known that the visual system’s spatial resolution for chrominance information is less than its resolution for luminance information. This fact is exploited both in TV transmission (where Cr and Cb are transmitted with half the bandwidth of L) and in JPEG/ADCT compression. Good quality color conversion may be had, therefore, by selecting one pixel from every 2\times2 block for full conversion through the 3-D LUT, and attempting to approximately preserve only the luminance channel for the remaining pixels.

In the case where the output space is RGB (or CMY), this consists of the following steps:

- From the input block \{(L00, Cr00, Cb00), (L01, Cr01, Cb01), (L10, Cr10, Cb10), (L11, Cr11, Cb11)\}, select one pixel, \((L_m, Cr_m, Cb_m)\) as the master pixel.
- Convert this pixel using full 3-D LUT interpolation to \((R_m, G_m, B_m)\).
- For each pixel \(P_{ij}\) in the block, assign

\[
R_{ij} = \frac{R_m}{L_m}L_{ij}; \quad G_{ij} = \frac{G_m}{L_m}L_{ij}; \quad B_{ij} = \frac{B_m}{L_m}L_{ij}. \quad (11.17)
\]

For efficiency, the loop need only be iterated over the remaining three pixels; however, there may be advantages in hardware to performing it in parallel on all four.

Note that the ratios in Equation 11.17 are invariant over pixel and need only be computed once per block. Alternatively, the ratio \(L_{ij}/L_m\) is invariant over separation and need only be computed once per pixel. The multiplications and divisions in Equation 11.17 can be sped up by computing in log space.

\[
R_{ij} = a^{\log_a R_m - \log_a L_m + \log_a L_{ij}} \quad (11.18)
\]

The difference \(\log_a R_m - \log_a L_m\) is invariant within the block and should be computed only once. Both the exponentiation and logarithms can be performed by one-dimensional table lookup. The loss of precision that results from computing in eight-bit log space should not be a problem because the
space is density-like and mimics the logarithmic behavior of the visual system.

The simplest and probably most reliable choice of the master pixel is the one with maximum luminance. Then, all of the ratios in Equation 11.17 are between 0 and 1. One can conceive of an analogous rule for converting to CMYK. However, four-colorant rendering typically involves some type of undercolor removal (UCR) and gray component replacement (GCR) strategy, establishing the relative proportions of CMY and K used to produce a given color. UCR/GCR can be very different from printer to printer; hence, a simple application of the above method may not always work. For the case where there is very little CMY subtraction, the following may be used:

- Convert one pixel from \((L_m, Cr_m, Cb_m)\) to \((C_m, M_m, Y_m, K_m)\) using the full three-dimensional lookup table approach.
- For the remaining pixels, \(P_{ij}\), set
  \[
  C_{ij} = 1 - ((1 - C_m)/L_m)L_{ij}
  \]
  (11.19)
  with analogous expressions for \(M_{ij}\) and \(Y_{ij}\).
- The same rule should not be used for \(K_{ij}\) because this would result in a situation where \(K\) is introduced at an edge between two colors, neither of which contains \(K\). Instead one might simply assign \(K_{ij} = K_m\).

Because \(L_m\) is the maximum \(L\), the ratio is always in the 0, 1 range, so all of the colors are between 0 and their original values.

At the expense of six more additions per block (and two shifts in software), \(Cr_m\) and \(Cb_m\) for the master pixel can be computed as the mean values of \(C_r\) and \(C_b\), providing substantial improvements in color image quality and reducing the tendency of the method to erode color against white edges. In the JPEG architecture, the chrominance channels are subsampled after filtering; hence, \(Cr\) and \(Cb\) are already averaged, and this step is not needed.

### 11.6.2 Results

Subsampled color correction was compared to conventional color correction, both in terms of computational cost and image quality. For the simulation, subsampled color conversion was performed from LCrCb to CMY using the six previously described steps, with no chroma averaging. This was followed by UCR/GCR and tone reproduction curve (TRC) correction applied to the full resolution image. The standard used was conventional color conversion from LCrCb to CMYK using three-dimensional lookup and tetrahedral interpolation. Table 11.4 summarizes the computational requirements per pixel. Clearly, subsampled color correction yields substantial savings in multiplications, additions, and shifts, with moderate savings in lookups — and an extra comparison operation. When full resolution and subsampled color correction are compared on 400-dpi prints, the two images are visually almost indistinguishable.
Many high-end printing systems store and/or transmit compressed images to save bandwidth, memory, and disk space. Prior to printing, the system will decompress and then color correct the image to compensate for the characteristics of a particular printer. In the conventional architecture, the decompression step converts the image from some encoded format (e.g., JPEG with Huffman encoding) into the spatial domain. In this domain, the color image may be stored in some standard device-independent color space such as IJG/CCIR YCrCb, SMPTE RGB, or Kodak Photo Ycc. The ensuing color correction transforms colors from this space to device coordinates. It is more efficient, however, to perform the color space conversion in the compressed domain. As with subsampled color correction, this approach involves parsing color correction into two processes, applying an expensive process to a low-resolution version of the image before decompression and an inexpensive operation to the entire image after decompression. The ratio of resolutions controls the trade-off between quality and speed.

In this section, we focus on color transforming JPEG compressed images. JPEG compression is described in detail in Chapter 8 and therefore is not presented here.

### 11.7 Correcting for RGB devices

RGB devices include monitors (cathode-ray tube based and liquid crystal display based), film recorders, and some printers that present an RGB interface. To perform fast color correction on compressed images, separate the color correction problem into two phases as shown in Figure 11.12a. First, account for any mismatch between the image and device primaries, and interactions among the device primaries. Second, linearize the individual primaries to some measured standard. (In Chapter 5, these two phases are denoted *characterization* and *calibration*, respectively.) The first phase converts device-independent data to device-dependent linearized data, and it requires at least a $3 \times 3$ matrix multiplication, and possibly a full three-dimensional table lookup with interpolation. The second phase involves using tone reproduction curves (TRCs) to linearize the device. The TRCs are implemented as one-dimensional table lookups and are therefore cheaper than the first phase. Normally, if the first phase involves a three-dimensional table lookup

<table>
<thead>
<tr>
<th>Method</th>
<th>Multiples</th>
<th>Adds</th>
<th>Compares</th>
<th>1-D table lookups</th>
<th>Binary shifts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conventional</td>
<td>12</td>
<td>14</td>
<td>2.5</td>
<td>19</td>
<td>2</td>
</tr>
<tr>
<td>Subsampled</td>
<td>2.25</td>
<td>9.75</td>
<td>3.63</td>
<td>17.1</td>
<td>0.5</td>
</tr>
<tr>
<td>Percent savings</td>
<td>81</td>
<td>30</td>
<td>-45</td>
<td>11</td>
<td>75</td>
</tr>
</tbody>
</table>

**Table 11.4** Computational Savings from Subsampled Color Correction
with interpolation, the second phase can be combined into the same table lookup, yielding the entire result. When converting compressed images, it is advantageous to keep the two phases separate.

For reasons that will become apparent shortly, it is important that the first phase preserves the sense of the color space. That is, if the input image data is in a device-independent RGB space (e.g., sRGB), this will be converted to a device-dependent RGB space. Similarly, if the input is in a device-independent luminance-chrominance form (e.g., YCbCr), it will be converted to a device-dependent luminance chrominance space, $Y'C'_1' C'_2'$. Note that the relation between YCbCr and RGB need not be kept between $Y'C'_1' C'_2'$ and the device-dependent RGB space (i.e., $R'G'B'$). The advantage of choosing a suitable luminance-chrominance space $Y'C'_1' C'_2'$ is to reduce operations when converting data to $R'G'B'$, as compared to the standard conversion from YCbCr to RGB. The disadvantage is the possibility of having data in different color spaces for the same DCT block (to be discussed later). The device luminance-chrominance space is defined by a simple transformation from device RGB (or CMY),

$$
Y' = 3 \frac{R'}{8} + \frac{G'}{2} + \frac{B'}{8}
$$

$$
C'_1 = B' - Y'
$$

$$
C'_2 = R' - Y'
$$

The inverse transform from device $Y'C'_1' C'_2'$ to device RGB is simply

$$
R' = C'_2 + Y
$$

$$
G' = Y - 3 \frac{C'_2}{4} - \frac{C'_1}{4}
$$

$$
= Y - C'_2 + \frac{(C'_2 - C'_1)}{4}
$$

$$
B' = C'_1 + Y
$$

Figure 11.12 The two phases of color correction for (a) RGB and (b) CMYK devices.
11.7.2 Correcting for CMYK Devices

For four-colorant printers, color correction comprises a transformation from a device-independent three-tuple to printer CMYK. Again, the transformation can be divided into two phases. The first phase converts the device-independent three-tuple (e.g., colorimetric RGB) into a device-dependent three-tuple with the same color coordinate orientation (e.g., printer RGB). This transformation is invariably implemented by a three-dimensional LUT and is the more computationally intensive phase. As before, the second phase comprises one-dimensional linearization TRCs. Additionally, four-colorant printers require undercolor removal (UCR) and gray component replacement (GCR), converting printer RGB to printer CMYK. (See Chapter 5 for details on UCR and GCR.) Figure 11.12b summarizes the two phases. Often, both phases are combined into a single LUT that maps the device-independent three-tuple directly to CMYK.

11.7.3 Color correction in the JPEG compressed domain

In conventional applications, the image is first decompressed, and then the color correction transform is applied, as shown in Figure 11.13. It is often faster to combine the decompression and color correction. We will describe the CMYK case; the RGB is simpler and easy to derive from the description of the CMYK case. We assume that images are stored in colorimetric (device-independent) RGB space and JPEG compressed format. We exploit the fact that the human visual system is most sensitive to errors at low spatial frequencies. (This is especially true of chromatic errors.) We thus propose an efficient color correction scheme that performs the more expensive (Phase I) correction on only a sub-image corresponding to the first few low-order DCT coefficients. The simpler (Phase II) color correction is then performed on the entire image. The steps are summarized below:

- Obtain the $N \times N$ sub-block of low-order DCT components, $N < 8$.
- Perform the $N \times N$ inverse DCT, and obtain a reduced block of an RGB image of reduced resolution.
- Apply Phase I color correction to the pixels of the $N \times N$ reduced block by three-dimensional lookup and interpolation, yielding an image in printer RGB coordinates.
- Perform a forward $N \times N$ DCT on the reduced block, and place the resulting DCT components into their respective positions in the original $8 \times 8$ DCT block.

Figure 11.13 Standard approach of color correcting JPEG compressed images.
• Perform an $8 \times 8$ inverse DCT to the resulting block to reconstruct the image at full resolution.
• Apply Phase II color correction to all image pixels of the resulting image.

These steps are summarized in Figure 11.14. Note that because the $N \times N$ sub-block of partially corrected pixels is reinserted into the $8 \times 8$ block in the fourth step, it is important that Phase I correction preserve the orientation of color space (i.e., colorimetric RGB to device RGB, colorimetric $Y_C, C_1$ to device $Y_C, C_2$, etc.).

In the luminance/chrominance case, we use colorimetric $Y_CbCr$ and device $Y_C, C_2$. As $Y_C, C_2$ is just a simplification of $Y_CbCr$, we expect the two color orientations to be very close. The DCT block right before inverse DCT contains low-frequency components from device $Y_C, C_2$ but high-frequency components from colorimetric $Y_CbCr$. As the difference between colorimetric and device $Y_C, C_2$ (or $Y_CbCr$) is generally more significant than the differences between colorimetric $Y_C, C_2$ and colorimetric $Y_CbCr$, the slightly different high-frequency components (whenever they are present) do not affect the quality of the color reproduction. Because the lowest frequency term (DC) is always converted using the full process, every block has the correct color on average.

11.7.4 Results

Results indicate significant computational savings over the conventional approach (i.e., full decompression, followed by RGB to CMYK by three-dimensional LUT and interpolation for all image pixels), with little loss in image quality. The quality/cost trade-off clearly depends on the number of low-order DCT coefficients (i.e., $N$) to which Phase I color correction is applied. Experiments were carried out for $N = 1, 2, 3$. When $N = 1$, the computational savings are a factor of 64, and the image quality may be acceptable for some applications. When $N = 3$, there is some computation savings, and the image quality is very close to that achieved with conventional decompression and color correction. The computational savings for different values of $N$ are summarized in Table 11.5.

11.8 Color transforming multiresolution images

The previous section exploited the reduced sensitivity of the human visual system in high spatial frequencies to develop an efficient color transforma-

![Figure 11.14](image-url)
The technique presented in this section is similarly motivated but applied to images represented in a multiresolution format. In many applications, images undergo a multiresolution decomposition (e.g., discrete wavelet transform) for the purpose of image compression or other processing. After the image is reconstructed from its multiresolution representation, it typically must be color corrected for rendering to a specific device. As in the previous section, the color correction process is divided into two phases: (1) a complex multidimensional transform (Phase 1), and (2) a series of essentially one-dimensional transforms (Phase 2). Phase 1 correction is then moved within the multiresolution reconstruction process in such a way that a small subset of the image samples undergoes the multidimensional correction. Phase 2 correction is then applied to all image samples after the image is reconstructed to its full resolution.

To illustrate the idea, a particular example is considered wherein a discrete wavelet transform (DWT) is used to represent the image in a multiresolution format for an application such as compression or inverse...
halftoning. A four-colorant (CMYK) printer is used as the output device to which the final image is rendered. This technique uses the multiresolution image representation to perform accurate color correction on a low-resolution image and then to perform a partial correction to the full resolution image prior to printing. The prime advantage is reduced computational cost required by the color correction. Another potential advantage is to reduce the effect of high-frequency noise in the color correction. This technique differs from that described in the previous section in that it is not restricted to the JPEG framework, but it addresses a more general architecture wherein multiple resolutions of an image (equivalently a frequency band decomposition of the image) are available at some point in the image processing path.

11.8.1 Wavelet representation

Numerous methods exist to decompose images into multiple resolutions. We use the wavelet transform as an example of a multiresolution technique, as this is a powerful and general framework that has shown considerable promise in several applications such as compression. The wavelet transform, assumed to operate independently on each of the color separations, is applied using a cascade of two-channel filter banks. These filter banks are composed of decimators (i.e., downsampling operators) and lowpass and highpass filters. At a given level, the image is split into four sub-bands (lowpass and highpass in each direction). The wavelet transform is obtained by repeating, at each level, the downsampling and filtering process over the lowpass sub-band of the previous level. This is shown in a pictorial example in Figure 11.15 for four levels. The image is then processed or analyzed in the wavelet domain for the given application and is finally submitted to an inverse wavelet transform so as to reconstruct the spatial domain representation. Note that since each downsampling or upsampling operation is accompanied by a filtering operation, the multiresolution representation can also be conceptualized as a decomposition of the image into a series of frequency bands, i.e., a sub-band decomposition. Such a decomposition trades frequency resolution for spatial resolution. The reader is referred to Chapter 8 for further details on the wavelet transform. The DWT is assumed to already be a part of the system for a given application such as compression (see Figure 11.16); we simply take advantage of its availability for efficient color correction.

11.8.2 Combining multiresolution analysis and color correction

The technique is implemented by moving Phase 1 color correction inside the inverse wavelet (i.e., image reconstruction) transform as shown in Figure 11.17. The image is partly reconstructed in the device-independent color space. Phase 1 color correction is then applied to convert the subsampled image to a device color space, and the remaining wavelet reconstruction is
applied in this device space. Finally, Phase 2 correction is applied to the fully reconstructed (i.e., full resolution) image.

As with the JPEG case described above, the input color space to the Phase 1 transform must have the same sense and orientation as the output color space (e.g., colorimetric RGB to device R’G’B’). This is because the frequency bands that have gone through Phase 1 color correction are now combined with the remaining frequency bands that have not been color corrected, and such combinations make sense only if the image data in all the bands are of the same color sense.

Note that, when performing \( N \) stages on the forward transform over an image of \( P \times P \) pixels and only \( N - M \) stages in the inverse transform, the result is a subsampled image with dimensions \( P' \times P' \), where \( P = 2^M P' \). Hence, Phase 1 color correction is applied to only a small fraction \( (1/2^M) \) of the samples, thus drastically reducing the computational cost. If \( C_1 \) and \( C_2 \) represent the computational cost per pixel of Phase 1 and Phase 2 color corrections, respectively, the overall cost for the proposed system is

\[
C = C_1 2^{-2M} + C_2 \tag{11.22}
\]

This is to be compared with the complexity of the standard color correction, which is on the order of \( 4C_1/3 \) (since three-dimensional interpolation has to be performed for each of \( C, M, Y, K \) as opposed to three color separations in the proposed method). The wavelet transform is not included in the cost because it is assumed to be needed for other reasons.

Table 11.6 shows the computational cost per sample necessary to color correct the image using the low-cost and conventional approaches for \( M = \ldots \)
1, 2, 3 (i.e., Phase 1 applied to 1/4, 1/16, and 1/64 of the samples, respectively). The savings provided by the proposed method is significant, particularly for multiplications, additions, and table lookups.

The parameter $M$ offers the trade-off between computational cost and image quality. Higher values of $M$ result in more computational savings but greater image degradation. Since the degradation is introduced at high spatial frequencies, where the HVS has reduced sensitivity, an acceptable quality/cost trade-off should be achievable with this technique.

Furthermore, because the color correction process itself is derived from measurements of targets of uniform patches, it is based on a low-frequency characterization of the device. Hence, there seems to be some consistency in applying the color correction to a low-frequency (i.e., locally averaged) version of the image. The high-frequency information must somehow be added back to the image to preserve texture and edges. However, because there is no explicit color characterization of the device at these frequencies, a simple approximation can be used to add the information in these bands, exploiting again the frequency response of the HVS.

### 11.8.3 Results

Balasubramanian et al.\textsuperscript{10} have examined the cost/benefit trade-off for wavelet-based color correction. For this experiment, no application-dependent processing (e.g., compression, segmentation, etc.) in the wavelet domain was included. That is, the wavelet transform was computed up to a given stage $M$ (where $M = 0$ represents the original image at full resolution); Phase 1 color correction was performed on the lowpass channel at that stage; the inverse wavelet transform was performed; and finally, Phase 2 color correction was applied on the entire image. A 16-tap QMF (Johnston) filter bank\textsuperscript{11} was used for the wavelet analysis. The images were processed through the standard and wavelet-based techniques and reproduced on a Xerox 5760 CMYK xerographic printer at 400 dpi.

For $M = 1$ or $M = 2$, pictorial images corrected with the proposed algorithm result in quality that is very close to, if not indistinguishable from,
that of the standard approach. For $M = 3$, artifacts start to appear in the form of \textit{beating or oscillations} in reasonably flat areas of the image. These artifacts are mainly noticeable in the luminance component of the image, so similar effects might also appear if one would apply the same concept for TRC adjustments of monochrome images. They result from the printing process’s highly nonlinear nature and from mixing corrected and uncorrected frequency bands.

The algorithm’s resulting image quality was also measured quantitatively. To provide a visually meaningful representation, the CMYK images were transformed into CIELAB space using a printer characterization function. To maintain consistency throughout the system, the same characterization function was used to derive the color correction transforms. However, while the colors are now in a perceptually meaningful representation, the CIELAB coordinate system suffers from the drawback that it is intended only for comparison of large, uniform patches and does not account for the sensitivity of the HVS to different spatial frequencies. This limits the utility of the CIELAB space for evaluating differences between reproductions of natural scenes. Since the algorithm exploits frequency characteristics of the HVS, an error metric should also be spatial frequency dependent. To this end, the spatial CIELAB (sCIELAB) model\textsuperscript{14} was used. This model uses the luminance and chrominance spatial contrast sensitivity functions, as well as the traditional CIELAB color space, to derive a color difference metric that accounts for both spatial frequency characteristics and color difference sensitivity of the HVS. For the color difference term, the original sCIELAB model uses CIELAB 1976 $\Delta E$. In this work, we used the CIE 1994 color difference equations as a superior metric for perceived color difference. The final sCIELAB error metric is in the form of an error image that represents the perceived difference between two images. An aggregate error criterion was selected as

$$
\Delta E_{\text{rms}} = \sqrt{\frac{1}{S} \sum_{ij} \Delta E_{ij}^2}
$$

(11.23)

where $S$ is the number of image pixels, and $\Delta E_{ij}$ are the individual sCIELAB $\Delta E$ errors for each pixel. Tests were carried out for $M = 1$, $M = 2$, and $M = 3$, for three images with varying pictorial content. A viewing distance of 14 in. and a 400 dpi resolution were assumed. The average $\Delta E$ across the three images is shown as a function of $M$ in Table 11.7. In general, $\Delta E < 1$ is considered to be below the visual detectability threshold. Noise, printer instability, and other imperfections in the system can easily give rise to reproduction errors between $\Delta E = 1$ and $\Delta E = 3$. Given this margin, and the qualitative observations, $M = 2$ appears to provide substantial savings with acceptable image quality. We remark, finally, that to make the evaluation more realistic, the effect of wavelet processing for the given application (e.g., compression, descreening, etc.) must also be considered.
11.9 Color transformations using multilevel chrominance halftoning

11.9.1 Introduction

As with the previous two approaches, this technique is based on the observation that the HVS is less sensitive to errors in chrominance than errors in luminance at high spatial frequencies. This fact may be exploited for efficient LUT interpolation in the case where the input to the LUT is a luminance-chrominance color space. Consider the example of a printer color correction LUT from L*a*b* to CMYK. The aforementioned property of the HVS suggests that it is possible to introduce high-frequency distortions in the a* and b* channels in a manner that is not objectionable to a human observer. One such distortion is multilevel halftoning along each of the a* and b* axes. Multilevel halftoning in three-dimensional LUTs has been described by Spaulding et al. and Love et al. In both cases, the authors apply halftoning to all three dimensions of the input signal, eliminating interpolation altogether. Here, we consider the special case of luminance-chrominance input and apply halftoning only to chrominance. By introducing high-frequency distortions only in the chrominance channels and preserving high accuracy in luminance, this technique offers image quality superior to that achieved by three-dimensional halftoning, at a modest computational cost.

Numerous multilevel halftoning techniques exist, e.g., screening, scalar error diffusion, vector error diffusion, etc. (These are covered in detail in Chapter 6.) Of these, multilevel screening is the least computationally intensive. The cost savings is achieved by choosing the halftone levels to coincide with the LUT node locations along a* and b*, as shown in Figure 11.18a. In this figure, the gray circle represents an input point in CIELAB space. First, multilevel halftoning is performed along the a* dimension by comparing the input a* value to a threshold in a spatial, periodically repeating halftone screen. If the input value exceeds the screen threshold, the former is mapped to the next larger level a* in the LUT; otherwise, it is mapped to the next smaller level a*. This process is repeated for the b* dimension. Thus, after the chrominance halftoning step, the input color maps to one of the four neighboring LUT nodes in the a*-b* plane (denoted by black circles in Figure 11.18a).
11.18a). Calculating the output CMYK now requires only a one-dimensional interpolation along the $L^*$ dimension between the neighboring levels $L^*_1$ and $L^*_2$, as shown in Figure 11.18b.

The choice of halftone screen is an important consideration. For Balasubramanian's experiment, an 8 × 8 Bayer screen was used whose threshold values are shown in Table 11.8. (The threshold values as well as the input values are normalized to the range [0–1] prior to halftoning.) The consider-

![Figure 11.18](image)

**Figure 11.18** (a) Two-dimensional multilevel halftoning in the chrominance plane, followed by (b) 1-D interpolation in luminance.

<table>
<thead>
<tr>
<th>Table 11.8</th>
<th>Threshold Values in Bayer Dot Screen Used for Multilevel Chrominance Halftoning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

*Note: Each of the values is divided by 31 before comparing with the input signal, assumed to be normalized to the range [0, 1].*
able variation in threshold values between adjacent halftone cells gives the Bayesian dot its ability to effectively contain the halftone signature within high spatial frequencies. Other screens, e.g., stochastic screens, can also be used. The same screen was used at the same spatial orientation for both a* and b* channels. It is conceivable that an improvement in image quality could be gained by using different screens or screen orientations for the a* and b* channels.

11.9.2 Results

To evaluate the quality of this technique, four test images of natural scenes were processed. These images, shown in grayscale in Figure 11.19, represent a variety of image content, including several smooth gradations, which are particularly challenging to multilevel halftoning. These images, specified in CIELAB, were transformed to CMYK with a three-dimensional LUT using the standard and halftone-based interpolation techniques. The LUT was constructed for a Xerox 5795 laser printer and comprised 16 uniformly spaced nodes along each of the three axes. Table 11.9 provides a comparison of the computational complexity of the novel chrominance halftoning technique with standard trilinear and tetrahedral interpolation. Recall that the latter is the fastest known three-dimensional interpolation algorithm. The table breaks down the computational cost in terms of multiplication, addition, comparison, and shift operations per pixel. The last column in the table shows actual execution times on a Sparc20 workstation. To account for variations from trial to trial, the timing data was derived by processing each of the four images through 10 trials and averaging the results across all the trials. Also, to make the comparison realistic, standard speed-up techniques such as caching were used with the trilinear and tetrahedral interpolation for the timing tests (these were not included in the computational cost measures). The data in Table 11.9 show that the cost savings are significant.

As in the previous sections, sCIELAB was used in conjunction with the CIE 1994 color difference metric to assess image quality objectively. The experiment was performed as follows. A three-dimensional LUT from L*a*b* to offset press CMYK was constructed using a standard printer

<table>
<thead>
<tr>
<th>Interpolation algorithm</th>
<th>M</th>
<th>A</th>
<th>C</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Trilinear</td>
<td>7N</td>
<td>7N + 2</td>
<td>0</td>
<td>2</td>
<td>18.0</td>
</tr>
<tr>
<td>2. Tetrahedral</td>
<td>3N</td>
<td>3N + 2</td>
<td>2.5</td>
<td>2</td>
<td>12.0</td>
</tr>
<tr>
<td>3. Chrominance halftoning + luminance interpolation</td>
<td>N</td>
<td>N + 4</td>
<td>0</td>
<td>2</td>
<td>5.9</td>
</tr>
<tr>
<td>4. Percent savings from (2) to (3)</td>
<td>67</td>
<td>43</td>
<td>100</td>
<td>0</td>
<td>51</td>
</tr>
</tbody>
</table>

Note: M, A, C, and S denote multiplications, additions, comparisons, and shift operations, respectively. T denotes the real execution time in µs to perform the interpolation for each pixel on a Sparc20 workstation.
characterization technique (see Chapter 5 for exemplary methods). The input L*a*b* image was transformed to printer CMYK with this three-dimensional LUT, using both the new technique and tetrahedral interpolation. The two CMYK images were then converted back to CIELAB using a printer characterization function for the offset printer and a visual difference image computed using sCIELAB. The mean and 95th percentile values from the difference image are reported in Table 11.10 for the four test images. Recall as a point of reference that, for an image consisting of a single color.

Figure 11.19 Images used in evaluating multilevel chrominance halftoning (clockwise from top left): Jill, Fruit, Carousel, Painted girl. “Painted girl” was obtained from the Kodak Photo Sampler (photographer: Steve Kelly).
Table 11.10  Average and 95th Percentile sCIELAB Errors for Images in Figure 11.19, Error Calculated between Reproductions Produced by Tetrahedral Interpolation vs. Chrominance Halftoning

<table>
<thead>
<tr>
<th>Images</th>
<th>Average</th>
<th>95th percentile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jill</td>
<td>2.59</td>
<td>4.4</td>
</tr>
<tr>
<td>Painted girl</td>
<td>3.38</td>
<td>6.0</td>
</tr>
<tr>
<td>Fruit</td>
<td>3.48</td>
<td>6.4</td>
</tr>
<tr>
<td>Carousel</td>
<td>4.0</td>
<td>8.1</td>
</tr>
</tbody>
</table>

(i.e., only the zero frequency component), the sCIELAB error reduces to CIE ΔE94. In our experience, the errors shown in Table 11.10 are within typical page-to-page and day-to-day print variations, which lie between 3 and 5 ΔE units for many printers. If LUT size is not a critical issue, a denser node sampling along a* and b* will further reduce the visibility of the chrominance halftoning.

Visually, it is usually difficult to distinguish the difference between chrominance halftoning and standard tetrahedral interpolation at normal viewing distances. Close examination sometimes reveals minor artifacts arising from the chrominance halftoning, particularly in smoothly varying image regions. On the whole, experimentation shows that the new technique works well for a wide variety of imagery, including computer generated graphics content. For printers with low spatial resolution, e.g., low-cost inkjet printers, the minor artifacts introduced by the chrominance halftoning are often masked by the final binary halftoning applied to each of the C, M, Y, K separations.

11.10 Conclusions

In this chapter, a number of techniques have been introduced to facilitate efficient color transformations. Since multidimensional LUTs pose the greatest challenges in terms of computational and storage costs, these have been the primary focus. The chapter began with an introduction to basic three-dimensional LUT interpolation geometries on regular lattices, including trilinear, prism, pyramidal, and tetrahedral. This was followed by a discussion of interpolation on irregular lattices, a special case being sequential interpolation. A set of practical techniques to accelerate interpolation operations was then described. Finally, various techniques were presented for reducing the computational cost of processing large images through three-dimensional LUTs. These techniques all exploit the human visual system’s reduced sensitivity to luminance errors at high spatial frequencies. The most suitable technique is determined by the trade-offs between cost and accuracy relevant to a given application. High-speed printing systems, for example, may call
for techniques that reduce computation, while low-cost devices with limited built-in memory are likely to benefit from techniques that compress LUT size with minimal impact on quality. Hardware assistance can also be used to complement the techniques proposed in this chapter. Finally, conceivably, these techniques can be combined with each other to form even more effective approaches.

Acknowledgments

The authors would like to thank Dean Harrington for his assistance in preparing many of the figures for this chapter.

References

