Characterizing Printer Gamuts Using Tetrahedral Interpolation*

Ian E. Bell and William Cowan
University of Waterloo, Computer Graphics Laboratory, Waterloo, Ontario, Canada

Abstract

In digital color printing, printer gamuts are often modeled as functions from CMY space into a device independent color space such as CIE XYZ tristimulus values. To render large raster images across devices, these gamut functions must be evaluated and inverted very efficiently; such performance can be provided only if the gamut function is represented as a look-up table, and evaluated by interpolation. The most common interpolation method uses data on a rectilinear grid, sometimes based on division of the cells into tetrahedra. It is not always possible to use a rectilinear scheme: available gamut measurements may not lie on such a grid, and the inverse of a gamut function sampled on a rectilinear grid does not take this form. Based on ideas developed for the numerical solution of partial differential equations, this paper develops a general tetrahedral interpolation technique that works efficiently with nonuniform data. The technique is shown to extend easily into higher-dimensional spaces.

1. Introduction

To characterize a digital color printer, we *calibrate* it to conform to a mathematical model, and *characterize* it by adjusting model parameters to match the device's behavior. We shall model devices as functions from their input space, such as CMY or RGB, to a device-independent color space, such as CIE XYZ. The device gamut is then represented as the range of this function, which we will often call the *gamut function.* Rendering images across different devices requires both evaluation of the devices' gamut functions, and of their inverses. Although the gamut functions are only known to us through sampled measurements, they are typically quite smooth for any device having acceptable color reproduction.

Raster images for printers today are often as large as 3500 by 3500 pixels, and performance requirements limit per-pixel computation time to microseconds. It is difficult to use a sophisticated mathematical model to approximate the gamut function without sacrificing performance; instead, measured samples of the gamut function are stored in look-up tables, and interpolated linearly at print time. This method is simple enough to be implemented in printer firmware, a desirable goal for optimum performance. *Rectilinear interpolation,* wherein the sample points lie on a rectilinear grid in device input space, is very common; we will contrast this method with a more flexible technique called *tetrahedral interpolation,* based on dividing the sample space into tetrahedra. This latter method generalizes to higher dimensional spaces in a way that is potentially more efficient for gamut evaluation than rectilinear methods.

1.1 Example

Consider a CMY printer, whose behavior we wish to characterize in CIE tristimulus XYZ space. One common approach would be to choose a set of 512 colors (all combinations of eight values each of C, M, and Y), print them, and measure them with a colorimeter. This data now gives us a rough idea of the behavior of the printer over its entire gamut. The CMY input values we used, if graphed in CMY space, would appear as a rectilinear grid — we call this a *uniform* rectilinear grid if the values are evenly spaced along each axis so that all the volume elements are congruent. The XYZ values we measured, if graphed in XYZ space, would form an irregular cluster sampling the gamut of the printer. Imagine that there is a smooth gamut function $f : R^3 \rightarrow R^3$ which models the printer, taking CMY values to XYZ values. We wish to evaluate this function for various CMY values, and evaluate the inverse gamut function f^{-1} for various XYZ values. Typically, given a raster image, the gamut function or its inverse would be evaluated at each pixel, so efficiency is important.

If we have used evenly-spaced CMY values, one method of approximating f is apparent: for a given CMY value $\bar{x}$, we can locate the cell of our sample points containing it by dividing each component by the spacing interval and truncating, then use multilinear interpolation on the eight XYZ values for the cell vertices to find the result. We will call this approach *uniform rectilinear interpolation.* If we have used unevenly-spaced CMY values, locating the enclosing cell is harder: often a binary search is used along each of the C, M, and Y axes; again, once the cell is found, multilinear interpolation is used. This *is non-uniform rectilinear interpolation.*

It is also possible to use a slightly more sophisticated approach, and imagine each cell divided into tetrahedra: after locating the cell containing $\bar{x}$, we find the subtetrahedron containing $\bar{x}$, and then use barycentric interpolation on the four XYZ values for the tetrahedron vertices. We will call this *subtetrahedral interpolation.*

We could also use a more general technique: if we first *tetrahedrize* the CMY sample points, that is, find tetrahedra that partition their convex hull, we can use special searching techniques to find the tetrahedron containing $\bar{x}$, and use barycentric interpolation immediately on the four XYZ values at the tetrahedron's vertices. We will call this *tetrahedral interpolation.*

These methods allow us to approximate the gamut function, given a table of CMY versus XYZ values where the CMY values lie on a rectilinear grid. Consider now the problem of approximating the inverse gamut function, which takes XYZ values to CMY values. No longer do we have domain points which lie on such a grid. If we limit ourselves to the ideas above, we really only have one choice: tetrahedral interpolation. There are other methods to deal with this issue, including ad hoc searching, or the extrapolation of a rectilinear grid around the sample points, but we will not discuss them here. Instead, we will focus on efficient methods of performing tetrahedral interpolation, as it is an appealing general technique.

2. The General Problem

Instead of limiting ourselves to three-dimensional spaces, from now on we will assume the gamut function f maps R^m to R^n (formerly CMY-space to XYZ-space in our example). There are practical problems which require high-dimensional spaces: black printers add a fourth dimension, the black ink K; a fifth spot color may be used for special applications; high-dimensional linear reflectance spaces are of growing interest for illuminant-independence in gamut-mapping.[10, 9]

From printer calibration, we are given k sample points $\{ \bar{x}^1 \ldots \bar{x}^k \}$ in R^m (the CMY values printed), and corresponding function values $\{ \bar{y}^1 \ldots \bar{y}^k \}$ in R^n (the measured XYZ). Let the total number of sample points be $N = k^m$ ($512 = 8^3$). We wish to construct a piece-wise linear approximation of f, $f: R^m \rightarrow R^n$, and then evaluate it for many values $\bar{x}$ in R^m. In our example, as a preprocessing step, we set up data structures for the CMY samples (rectangular cells or tetrahedra), and then evaluated the gamut function for an image pixel-by-pixel.

Table 1 shows the general algorithm to interpolation, which we describe below:

```
ρ = partition ({ x̄^i } );
FOR EACH x̄ ∈ input image
        P = find (ρ , x̄ );
        IF P exists THEN
            c̄ = interp (P, x̄ );
            ŷ = f (M^P) c̄ ;
        ELSE
            ŷ = out-of-gamut (ρ , x̄ );
        END IF
        output ( ŷ );
    END FOR
```

Table 1.General interpolation algorithm

1. ρ = *partition* ($\{ \bar{x}^i \}$);
   Partition () determines a grid topology for the sample points, and finds a set of solid polytopes ρ which decompose the convex hull of the $\bar{x}^i$ such that[3]:
   • the $\bar{x}^i$ contain only the vertices of each polytope, and
   • any two polytopes intersect in a facet or not at all.

   The *partition* () function is the *preprocessing* step to characterize the printer; it can be carried out overnight on printer samples if it is slow — it is more important that the subsequent loop in the algorithm be efficient, as it will likely be performed for every pixel in an image at print time.

2. P = *find*(ρ , $\bar{x}$);
   Find () uses the grid topology to locate the unique polytope P containing $\bar{x}$, or return a null value if there is none.

3. $\bar{c}$ = *interp* (P, $\bar{x}$);
   Here we express $\bar{x}$ in terms of the polytope vertices, so that later we can express $\bar{y}$ in terms of its surrounding sample points in the range. *Interp* () solves for coefficients $\bar{c}$ so that $\bar{x} = M^P \bar{c}$, where M^P is the matrix of vertices of the polytope P, in columns. This system will always be under-constrained, however, since the polytope must have at least $m + 1$ points, and each is an m-vector. The particular method, such as rectilinear interpolation, will determine which extra constraints to add.

   Applying the gamut function f () to the equation $\bar{x} = M^P \bar{c}$, we arrive at our linear approximation $f(\bar{x}) \approx f(M^P) \bar{c}$.

4. $\hat{y} = f(M^P) \bar{c}$;
   This is straightforward matrix multiplication — we have completed the inner loop at this point.

5. $\hat{y}$ = *out-of-gamut* (ρ, x);
   When a point $\bar{x}$ is outside the gamut, that is, outside the convex hull of our sample points, the *find* () function fails to locate an enclosing polytope. Effectively, we are asking to print an unprintable color. We will assume that the function *out-of-gamut* () takes some appropriate action in this case, and not address here the difficulties of choosing a "closest" approximating color from the gamut.

3. Using Rectilinear Interpolation

Recall from our example how we chose a fixed set of values for C, M, and Y, and used their Cartesian product as a uniform rectangular sampling of the printer gamut. Measuring the tristimulus values of each color then gave us a table associating CMY and XYZ values. Consider how this approach fits into the general algorithm:

1. ρ = *partition* ($\{ \bar{x}^i \}$);
   The topology is that of a rectangular lattice. The structure is implicit: there is no need to store information about which cell is next to which. For uniform data, we store the starting point and spacing for each axis; to navigate non-uniform data, we create a multidimensional array indirectly referencing the sample values.

2. P = *find* (ρ , $\bar{x}$);
   For the uniform case, it only takes one division per coordinate, and truncation, to decide which cell a given point $\bar{x}$ lies in, thus m divisions; for the non-uniform case, we use binary search along each axis,

giving $O(m \log k) = O(\log N)$ comparisons. In early computer hardware, floating-point division was far more costly than comparison, but nowadays they are comparable — we will not distinguish them in subsequent complexity estimates.

3. $\bar{c} = interp\ (\mathrm{P}, \bar{x}\ )$;
For simplicity, consider the case where $m = 3$. Trilinear interpolation is used within each cell. Say $\bar{x}$ lies in the cell defined by

$$c^0 \le x_1 \le c^1 \quad (1)$$

$$m^0 \le x_2 \le m^1 \quad (2)$$

$$y^0 \le x_3 \le y^1 \quad (3)$$

Then if we let

$$\alpha_1 = (x_1 - c^0\ )/(c^1 - c^0) \quad (4)$$

$$\alpha_2 = (x_2 - m^0\ )/(m^1 - m^0) \quad (5)$$

$$\alpha_3 = (x_3 - y^0\ )/(y^1 - y^0) \quad (6)$$

multilinear interpolation implies that we have

$$\bar{x} = (1 - \alpha_1)V^0 + \alpha_1 V^1 \quad (7)$$

$$V^0 := (1 - \alpha_2)V^{00} + \alpha_2 V^{01} \quad (8)$$

$$V^1 := (1 - \alpha_2)V^{10} + \alpha_2 V^{11} \quad (9)$$

$$V^{00} := (1 - \alpha_3)V^{000} + \alpha_3 V^{001} \quad (10)$$

$$V^{01} := (1 - \alpha_3)V^{010} + \alpha_3 V^{011} \quad (11)$$

$$V^{10} := (1 - \alpha_3)V^{100} + \alpha_3 V^{101} \quad (12)$$

$$V^{11} := (1 - \alpha_3)V^{110} + \alpha_3 V^{111} \quad (13)$$

and where $V^{rst} := [c^r, m^s, y^t]$, the vertices of the cell. It is possible to write this in the form $\bar{x} = M\,\bar{c}$, where M is the 3×8 matrix of vertices V^{rst} (as column vectors), and $\bar{c}$ is the rather messy vector of α's obtained by expanding the equations above.

For $m > 3$, each of the m components of $\bar{c}$ must be computed as a product of α's and their complements, as indicated in the recursive equations. In this example, for each component, there are $2 + 4 + 8$ multiplications; in m dimensions, there will be

$$\Sigma_{i=1}^{m}\ 2^i = 2^{m+1} - 2$$

multiplications. Thus the complexity of *interp* () for all m components is $O\ (m2^{m+1}\ )$.

4. $\hat{y} = f\ (M^P)\,\bar{c}$;
The final step for the interpolation is to apply f to the above equation, to obtain $f\ (x) \approx f\ (\mathrm{M})\,\bar{c}$, where $f\ (M)$ is the matrix of function values at the vertices of the cell. For each of the n components in the result $\hat{y}$, a dot product is computed between a row of $f\ (M^P)$ and $\bar{\boldsymbol{c}}$, both 2^m-vectors. In total, there are $n2^m$ multiplications.

3.1 Cost of Rectangular Interpolation

The critical steps in the algorithm are those of the loop, which may be carried out for millions of pixels. For both uniform and non-uniform cases, *find* (), with m multiplications or $O(\log N)$ comparisons, respectively, is dominated by *interp* () and the evaluation, which require $0((2m+n)2^m)$ multiplications (see Table 2). The worst case costs are the same as the average costs for this method.

Table 2. Expected Cost of Rectilinear Interpolation

	Unif Rect	Non-Unif Rect
Find()	m	$O(\log N)$
Interp()	$O(m2^{m+1})$	$O(m2^{m+1})$
Evaluation	$n2^m$	$\mathrm{n}2^m$
Dominant Term	$O((2m + n)2^m)$	$O((2m + n)2^m$

Confining the data points to a uniform rectilinear grid is a severe restriction: printer gamuts are notorious for misbehaving in the dark regions, and so despite the slight-ly greater cost, non-uniform interpolation is preferred. It would be better still to eliminate the rectilinear restriction, for then, from arbitrary data sets, we could approximate the gamut function, or even its inverse. This is our goal with tetrahedral interpolation.

4. Properties of Tetrahedrizations

Before continuing with the description of subtetrahedral and tetrahedral interpolation, it is worth discussing some of the properties of *tetrahedrizations.*

4.1 Counting Tetrahedra

If we partition the convex hull of $\{\ \bar{x}^1 \ldots\ \bar{x}^N\}$ into t tetrahedra, we may discover that t is impractically large for our data structures, or some of the tetrahedra are thin and splintery, causing numerical difficulties. (Note that in m dimensions an $(m + 1)$-vertex polytope is usually called a *simplex* — we will usually refer to them as tetrahedra for consistency with the three-dimensional case.) The numerical problems are reduced by choosing the *Delaunay tetrahedrization* of the data, which guarantees that the circumsphere of each tetrahedron encloses no other data points.[2, 7, 3] This yields a worst-case value for t of $O\ (N^{\lceil m/2 \rceil})$, too large for storage; fortunately random data give an empirical space complexity of $O\ (N)$. [7] The expense of tetrahedrization is not crucial, being part of our *partition* () preprocessing step.

4.2 Barycentric Coordinates

In R^3, imagine joining $\bar{x}$ to the enclosing tetrahedron's vertices to form four small tetrahedra; the volumes of these as fractions of the large tetrahedron's volume are the *barycentric coordinates* of $\bar{x}$. Computing the volumes gives positive values if $\bar{x}$ is really inside the tetrahedron, but gives one or more negative values otherwise, a prop-erty we will exploit in *find* (). More commonly, barycentric coordinates are used for interpolation, as they express $\bar{x}$ in terms of the tetrahedron vertices. If $\{\ \bar{x}^1 \ldots\ \bar{x}^4\}$ are the four vertices of the tetrahedron, we compute the coordinates by solving

$$\begin{bmatrix} \bar{x}^1 & \bar{x}^2 & \bar{x}^3 & \bar{x}^4 \\ 1 & 1 & 1 & 1 \end{bmatrix} \bar{c} = \begin{bmatrix} \bar{x} \\ 1 \end{bmatrix}$$

This computation generalizes easily to m dimensions. Note that the LU-decomposition of the matrix can be precomputed for each tetrahedron, if there is storage space, making this only as expensive as backsolving the system: $O(m^2)$ multiplications.

4.3 Finding an Enclosing Tetrahedron

Given a Delaunay tetrahedrization of our data, the *find* () operation must locate an enclosing tetrahedron for a point $\bar{x}$. Searching all t tetrahedra is too expensive. We suggest two alternatives: the *walking* algorithm, and the *binary space-partitioning* (BSP) algorithm.

4.3.1 Walking Algorithm

If $\bar{x}$ is outside a tetrahedron S, at least one of its barycentric coordinates relative to S will be negative, showing that $\bar{x}$ lies away from the corresponding facet. The most negative coordinate can be used as likely direction to proceed to a tetrahedron containing $\bar{x}$. Starting from an arbitrary tetrahedron, we can compute barycentric coordinates, step over the facet with the most negative coordinate, and repeat; once all coordinates are positive, we have found the enclosing tetrahedron T. This technique is often used with tetrahedral meshes in scientific computing.[2, 7]

This *walk* will closely follow a straight line from the initial tetrahedron to T, and thus if the data points are evenly distributed in m-dimensional space, we expect, on average, to traverse $O(t^{1/m}) = O(N^{1/m}) = O(k)$ tetrahedra. In the worst case, however, if the data points fall close to a line, we could traverse all t tetrahedra — this is most unlikely for the gamut of any useful output device.

4.3.2 BSP Algorithm

The data structure known as a binary space-partitioning tree is usually used to determine a back-to-front drawing order for polygons relative to an observer.[4, 6] Each polygon is considered to lie in an oriented plane in space, so points lie before, on, or behind it. A binary tree is then built with a polygon at each node, so that all left children of a polygon lie in front of it, and all right children lie behind it; what is done with polygons in the same plane depends on the application. Some polygons may be split by planes intersecting them: each half of the polygon is then stored separately. It is possible with simple heuristics[5] to order the polygons so that the tree is fairly balanced, with a height of $O(\log p)$, where p is the number of polygons. Once the tree is built, the region of space containing the viewpoint can be determined by checking which half-space the viewpoint lies in at each node, down a path to a leaf.

We can use this structure with the facets of the tetrahedra as polygons; on the leaves of the tree we store identifiers for the tetrahedra. To find the tetrahedron T containing $\bar{x}$, we compute the dot product with facet normals at each node, and follow the appropriate branch down to T; the total path length is $O(\log t)$, giving, on average, $O(m \log N)$ multiplications. The BSP tree will be $O(t)$ in size, but its computation will be part of the *partition* () preprocessing step.

5. Subtetrahedral Interpolation

The main difference between rectilinear interpolation and subtetrahedral interpolation is that the former uses rectangular parallelepipeds as cells, and the latter further subdivides them into tetrahedra.[8] In three dimensions, one can color the vertices of a cube red and black so that those joined by an edge are different colors. The usual method of subdivision is into five tetrahedra: four have one black and three red vertices; the fifth is contained within the cube, and has all red vertices. Exchanging red for black and following the same approach gives another subdivision. Six tetrahedra are created by slicing the cube vertically through a diagonal of the top facet, and dividing each of the two resulting prisms into three tetrahedra.

In higher dimensions, there are also many ways to subdivide a hypercube into s simplices — computing the possible numbers of tetrahedra for each m is an open research problem, but we can give a lower bound. Every simplex uses up at most $m + 1$ of the hypercube vertices, so s is at least $2^m/(m + 1) = O(2^m)$ simplices. If we use Delaunay tetrahedrization on the hypercube vertices, we expect to have $O(2^m)$ simplices from our earlier arguments, so on average, $s = O(2^m)$. In the worst case, the Delaunay method could give $s = O((2^m)^{\lceil m/2 \rceil}) = O(2^{m^2})$ simplices, but this shouldn't occur with a regular structure like the hypercube—there may be better upper bounds for s.

The general algorithm proceeds as follows:

1. ρ = *partition* ($\{ \bar{x}^i \}$);
   Some extra information is needed to identify which of the two subdivision methods is being used.

2. P = *find* (ρ , $\bar{x}$);
   As in rectilinear interpolation, finding the cell takes m divisions (uniform sampling), or $O(\log N)$ comparisons (non-uniform sampling).

Table 3. Expected Cost of Subtetrahedral Interpolation

	Unif SubTet	Non-Unif SubTet
Find ()	m	$O(\log N)$
Interp ()	$O(m^2)$	$O(m^2)$
Evaluation	$O(nm)$	$O(nm)$
Dominant Term	$O(m(m + n))$	$O(m(m + n))$

Table 4. Worst-Case Cost of Subtetrahedral Interpolation

	Unif SubTet	Non-Unif SubTet
Find ()	$O(2^{m2})$	$O(2^{m2})$
Interp ()	$O(m^2)$	$O(m^2)$
Evaluation	$O(nm)$	$O(nm)$
Dominant Term	$O(2^{m2})$	$O(2^{m2})$

To find the enclosing subtetrahedron, a complicated algorithm like *BSP* would be overkill; using the walking method, we expect *find* () to take $O(s^{1/m}) = O((2^m)^{1/m}) = O(1)$, or constant time. The worst case imaginable, walking through all tetrahedra when their number is maximal, is $O(s) = O(2^{m^2})$, but is highly unlikely. We conclude that finding the cell dominates.

3. $\bar{c}$ = *interp* (P, $\bar{x}$);
We interpolate with barycentric coordinates, as described earlier, using $O(m^2)$ multiplications.

4. $\hat{y} = f(M^P)\,\bar{c}$;
For each of the n components in the result $\hat{y}$, there is a dot product of a row of $f(M^P)$ and $\bar{c}$, both $(m+1)$-vectors in this case. In total, there are $n(m+1) = O(nm)$ multiplications.

5.1 Cost of Subtetrahedral Interpolation

As with rectilinear interpolation, *interp* () and evaluation dominate, but the small number of vertices in a tetrahedron reduces the complexity to give $O(m^2)+O(nm) = O(m(m+n))$ multiplications on average (see Table 3 and Table 4). In the worst-case scenario, *find* () may dominate with $O(2^{m^2})$. We have not found a simple tetrahedrization of the hypercube which would reduce this estimate.

This idea avoids the expense of rectilinear *interp* (), but since characterization measurements are noisy, interpolating fewer data points may sacrifice accuracy. (One would prefer to smooth the data to conform more closely to the true gamut — this is the subject of our current research.[1]) For $m = 3$, it is not much more difficult to implement than rectangular interpolation, and yet runs faster, so this technique has become quite popular. The restriction of hav- ing rectilinear sample points remains.

6. Tetrahedral Interpolation

For tetrahedral interpolation, we require no particular topology of the samples, except non-degeneracy as it applies to the chosen tetrahedrization algorithm.[2,7]

1. ρ = *partition* ($\{\bar{x}^i\}$);
The *partition* () function uses the Delaunay tetrahedrization, and builds a data structure to determine the neighbors of a tetrahedron, plus LU-decompositions for the barycentric coordinates, and if using the BSP method, the BSP tree and facet normals.

Table 5. Expected Cost of Tetrahedral Interpolation

	TetWalk	TetBSP
Find ()	$O(N^{1/m})$	$0(m\log N)$
Interp ()	O	$O(m^2)$
Evaluation	$O(nm)$	$O(nm)$
Dominant Term	$O(N^{1/m})$	$O(m\log N)$

Table 6. Worst-Case Cost of Tetrahedral Interpolation

	TetWalk	TetBSP
Find ()	$O(N^{\lceil m/2 \rceil})$	$0(m\log N^{\lceil m/2 \rceil})$
Interp ()	O	$O(m^2)$
Evaluation	$O(nm)$	$O(nm)$
Dominant Term	$O(N^{\lceil m/2 \rceil})$	$O(m\log N^{\lceil m/2 \rceil})$

2. P = *find*(ρ , $\bar{x}$);
We walk, in $O(N^{1/m})$ multiplications, or use the BSP method for $O(M\log N)$ multiplications. In the worst case, we have maximal numbers of tetrahedra, $t = O(N^{\lceil m/2 \rceil})$: we may walk them all, $O(N^{\lceil m/2 \rceil})$; BSP gives $O(m\log N^{\lceil m/2 \rceil})$.

3. $\bar{c}$ = *interp*(P, $\bar{x}$);
Interp(), as we have shown before, requires finding the barycentric coordinates of $\bar{x}$ in P. With the walking algorithm, we have already performed this computation in *find*(); with BSP, $O(m^2)$ multiplications are needed.

4. $\hat{y} = f(M^P)\,\bar{c}$;
As in subtetrahedral interpolation, we require $O(nm)$ multiplications.

6.1 Cost of Tetrahedral Interpolation

For tetrahedral interpolation, *find* () dominates with the expected $O(N^{1/m})$ (walking), or $O(m \log N$ (BSP) multiplications; in the worst case it also dominates with $O(N^{\lceil m/2 \rceil})$(walking), or $O(m \log N^{\lceil m/2 \rceil})$. (*BSP*) (see Table 5 and Table 6).

Our analysis indicates that this method is usually more expensive than rectilinear or subtetrahedral interpolation, but competitive for large m. Given that it requires no special structure of the sample points, and can be used to approximate the gamut function and the inverse gamut function, we believe it has good potential for certain gamut-mapping applications.

Table 7. Evaluation of dominant terms for typical parameter values (rounded to integers).

N	m	n	Rect $(2m+n)2^m$	SubTet $m(m+n)$	TetWalk $N^{1/m}$	TetBSP $m \log N$
1000	3	3	72	18	10	21
1000	3	4	80	21	10	21
1000	3	5	88	24	10	21
1000	4	3	176	28	6	28
1000	4	4	192	32	6	28
1000	5	3	416	40	4	35
4000	3	3	72	18	16	25
4000	3	4	80	21	16	25
4000	3	5	88	24	16	25
4000	4	3	176	28	8	33
4000	4	4	192	32	8	33
4000	5	3	416	40	5	41

7. Comparison of All Methods

Table 7 evaluates the expected dominant term for typical values of *N, m* and *n*. This is not intended to represent execution times, as there are many other factors involved, but serves only to depict rough trends in the magnitudes.

We are currently implementing these methods to determine realistic empirical data. It is apparent, however, from this table, that we should not expect tetrahedral interpolation time to increase enormously with increases in dimension: the walking method actually decreases with increasing *m*. If we evaluate the worst-case complexities, this method and the subtetrahedral method grow excessively; only the rectilinear and tetrahedral-BSP algorithms appear practical. Experimentation will give the final verdict.

8. Conclusions

We are interested in high-dimensional linear representations of reflectance; this has led us to consider the problems inherent in high-dimensional interpolation of printer calibration data. Rectilinear and subtetrahedral methods restrict the form of the calibration data, and slow down with increasing dimensionality; however, in three dimensions, they are efficient and easy to implement.

Tetrahedral interpolation is extremely appealing because it does not require data on a rectilinear grid. Its disadvantages include the need for a tetrahedrization algorithm and a neighbor data structure, high preprocessing time, and the potential for expensive worst-case scenarios. The walking and BSP methods for *find* () make the algorithm competitive for typical sizes of gamut problems. Note, however, that we have not addressed the expense of the *out-of-gamut* () function: in real applications, the cost of this function is considerable, and has a great effect on the quality of image reproduction.

Calibration data is measured, and therefore noisy. Our related work now concentrates on smoothing out the noise using spline functions over both rectilinear and tetrahedral topologies. This will give compact and smooth gamut representations, and will probably prove even more effective than the interpolation techniques described here.

9. Acknowledgments

The authors thank Anna Lubiw for her combinatorial insight; Bruce Simpson, Adrian Bowyer, and Barry Joe for information and software related to *k*-dimensional tetrahedrizations.

References

1. Ian E. Bell. Algorithms for the creation of gamut mapping transformations based on reflective image representations. Ph.D. thesis (in preparation), University of Waterloo Computer Graphics Laboratory.
2. A. Bowyer. Computing dirichlet tessellations. *The Computer Journal,* **24**(2):162-166, 1981. dirichlet tessellations.
3. N. Edelsbrunner, F. P. Preparata, and D. B. West. Tetrahedrizing point sets in three dimensions. *J. Symbolic Computation,* **10**:335-347, 1990.
4. James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics Principles and Practice.* Addison-Wesley, 2nd edition, 1990.
5. H. Fuchs, G. D. Abram, and E. D. Grant. Near real-time shaded display of rigid objects. *Computer Graphics,* pages 65-72, 1983.
6. H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. *Computer Graphics,* pages 124-133, 1980.
7. Barry Joe. Construction of *k*-dimensional delaunay triangulations using local transformations (to-appear). *SIAM J. Sci. Comput.,* **14**, November ,1993.
8. Katsuhiro Kanamori, Hidehiko Kawakami, and Hiroaki Kotera. A novel color transformation algorithm and its applications. In *SPIE Image Processing Algorithms and Technique,* volume **1244**, pages 272-281, 1990.
9. Laurence T. Maloney and Brian A. Wandell. Color constancy: A method for recovering surface spectral reflectance. *Journal of the Optical Society of America A,* **3**(1):29-33, January, 1986.
10. Brian A. Wandell. The synthesis and analysis of color images. *IEEE Transactions of Pattern Analysis and Machine Intelligence,* **9**(1):2-13, January ,1987.

* This research is supported in part by the Natural Sciences and Engineering Research Council of Canada; Ian Bell thanks Xerox Corporation for financial support.