Radial and Pruned Tetrahedral Interpolation Techniques

Gary L. Vondran
Hewlett-Packard Laboratories - Cambridge

HPL-98-95

radial, tetrahedral, pruned tetrahedral, interpolation, color mapping, color space conversion, cube subdivision, image processing, color processing

Presented are two new interpolation approaches based on Cube Subdivision. The first, Radial Interpolation, exploits the observation that the subcube in each iteration can be generated by averaging the vertex of the cube indexed by the slice across the input values with all other vertices of the cube. This leads to further optimizations, resulting in Radial Interpolation requiring only *n+1* table accesses and *n+1* additions to generate each pixel output component, when interpolating using *n* bits. The second, Pruned Tetrahedral Interpolation, takes advantage of pruning techniques to implement a tetrahedral interpolation in 2^n additions for each pixel output component. Also presented is an implementation that can perform both Radial and Pruned Tetrahedral Interpolations. Finally, Non-Symmetric Radial and Pruned Tetrahedral Interpolation approaches are presented that allow for interpolation with a non-symmetric color table.

1. Background

When working with color data, it is a common desire to convert the data from one color space representation into another. A scanner may use the additive color space RGB (Red, Green, Blue) while printers use the subtractive color space CMY (Cyan, Magenta, Yellow). Other spaces include CMYK (Cyan, Magenta, Yellow, and Black), CieLab, YC_bC_r, LUV, LHS, and others. To allow these devices to interact, a method is needed to convert the data from the color space of one device to the color space of another.

1.1 Color Space Conversion Process

Ideally a mathematical formula can be used for color space conversion. Some conversions can be performed with a linear matrix multiply while others have complex non-linear relationships. In addition, translations to expand the color gamut and correct for imperfections in the reading, displaying, and printing devices are also performed on the color data. It is generally desired to combine these translations into the color conversion process in order to reduce computations. It is also desired that the implementation be flexible such that it may be used to translate from any color space to any other color space, including device corrections.

The most direct approach that satisfies all these wants is a lookup table. The advantages of a lookup table are that it is simple and can be used for any translation. The translation performed is determined by the table used. The drawback, however, is that these tables can be enormous. A conversion from a 24 bit RGB color space to a 24 bit CMY color space requires a

$$2^{24\text{ bit RGB}} \text{ Table Entries} * 24 \text{ bit CMY / Entry} * 1 \text{ Byte} / 8 \text{ bit} = 48 \text{ MByte Table.}$$

A 48 MByte table is clearly unreasonable and makes the direct table lookup approach unacceptable.

The solution, presented in Pugsley [10] and Pugsley [11], is to reduce the table by storing only a coarse lattice of points of the color space and using linear interpolation to compute the values between these points. In this approach, only values for every 2^n points are stored in each of the input dimensions. The last point of each dimension is also stored to allow for interpolation between the last point that is a multiple of 2^n and the final value in the color space. Using this approach, the lookup table is $(2^{b-n}+1)^d$ entries, given d input dimensions each with b input bits. The value n should be chosen such that the behavior between lattice points is approximately linear. If n is chosen too high, the linear interpolation will not adequately approximate the translation. Choosing n to be too small results in a table that is larger then needed. Given a 24 bit RGB input (d=3 and b=8) to be converted to 24 bit CMY with n=4, the resultant table size is

$$(2^{8-4}+1)^3 \text{ Table Entries} * 24 \text{ bit CMY / Entry} * 1 \text{ Byte} / 8 \text{ bit} = 14.4 \text{ KByte Table.}$$

Figure 1 shows an overview of the color space conversion process. In it, the input value (a,b,c), representing a color in a cylindrical color space, is translated into coordinates (x,y,z), representing the same color in an output cartesian color space.

In Figure 1, the input value's upper bits, (a_u,b_u,c_u), index the color table, retrieving the coarse lattice cube that contains the input value (a,b,c). The lower bits, (a_l,b_l,c_l), are then used to interpolate the output value within this cube.

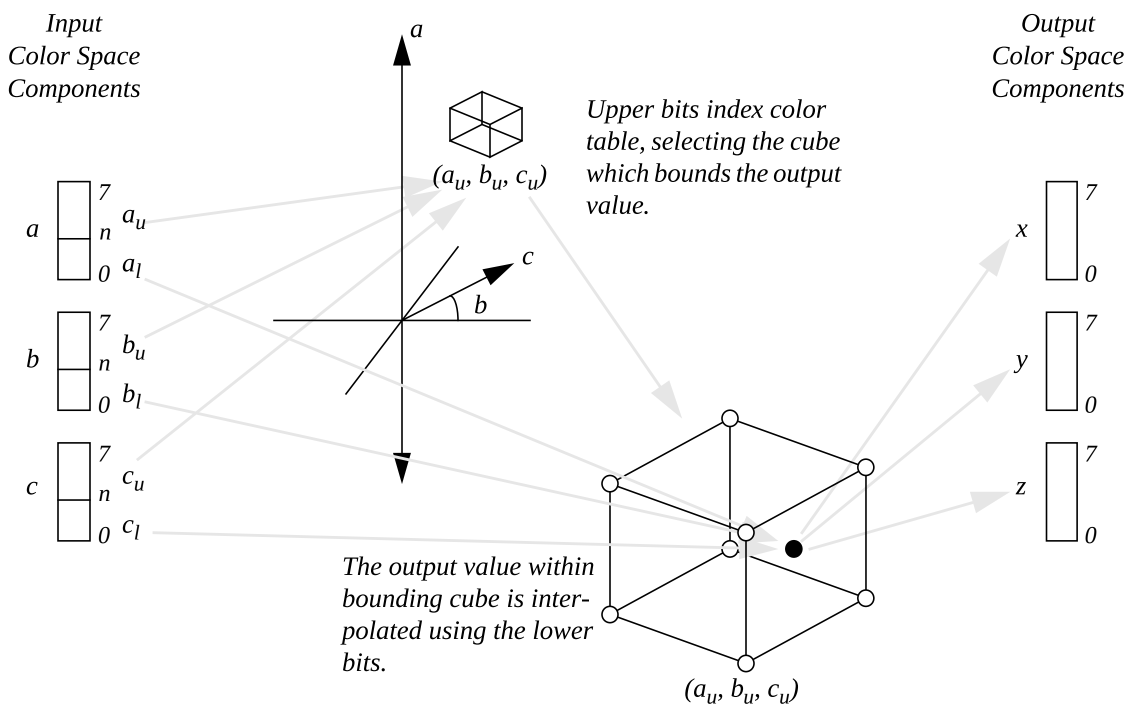

Input Color Space Components
a
7
n
0
a_u
a_l
b
7
n
0
b_u
b_l
c
7
n
0
c_u
c_l
a
c
b
(a_u, b_u, c_u)
Upper bits index color table, selecting the cube which bounds the output value.
Output Color Space Components
x
7
0
y
7
0
z
7
0
The output value within bounding cube is interpolated using the lower bits.
(a_u, b_u, c_u)

Figure 1. Color Space Conversion Process

A cube is returned from the color table lookup because the color table only stores a coarse lattice of points, spaced 2^n values apart. Thus, each component of the input value (*a*,*b*,*c*) may fall between these lattice points, as shown below.

$$2^n a_u \le a < 2^n(a_u + 1) \qquad \text{where } a_u = \lfloor a/2^n \rfloor = a >> n = a[b:n]$$

$$2^n b_u \le b < 2^n(b_u + 1) \qquad \text{where } b_u = \lfloor b/2^n \rfloor = b >> n = b[b:n]$$

$$2^n c_u \le c < 2^n(c_u + 1) \qquad \text{where } c_u = \lfloor c/2^n \rfloor = c >> n = c[b:n]$$

To be able to interpolate a value that does not fall on a lattice point, all the nearest entries to the input point within the coarse lattice are accessed. In a three dimensional lattice, the value (*a*,*b*,*c*) is between eight lattice entries: (a_u,b_u,c_u), (a_u,b_u,c_u+1), (a_u,b_u+1,c_u), (a_u,b_u+1,c_u+1), (a_u+1,b_u,c_u), (a_u+1,b_u,c_u+1), (a_u+1,b_u+1,c_u), (a_u+1,b_u+1,c_u+1). This is shown in figure 2. For a one dimensional space, a line is returned. For a two dimensional space, a square is returned. For a four dimensional space a hyper-cube is returned. For all cases, the number of points returned is 2^d, given d input dimensions.

Once the cube is returned from the color table, the location within the cube is interpolated using the lower bits of the input value, i.e. (a_l,b_l,c_l). It is the purpose of this paper to present the cube subdivision interpolation methods developed.

1.2 Cube Subdivision Interpolation

Jay Gondek [2] developed the concept of Cube Subdivision Interpolation. Cube Subdivision uses

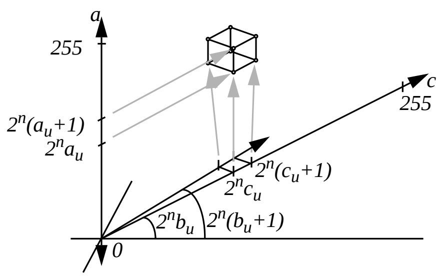


Figure 2. Eight Nearest Lattice Points to (a,b,c)

the process of dividing the cube into subcubes and selecting the subcube that contains the value (*a,b,c*). The selected subcube then becomes the cube for the next subdivision iteration. The subdivision process is performed n times, at which point the location (*a,b,c*) will be origin of the final cube and its value will be the output (*x,y,z*).

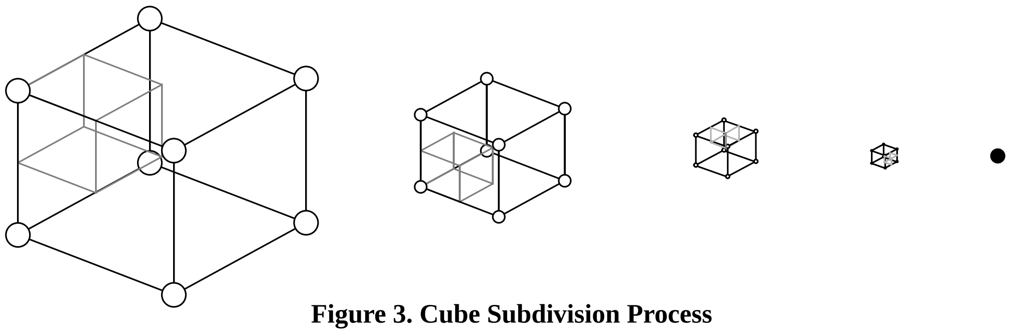


Figure 3. Cube Subdivision Process

In this example, the initial cube is divided into eight subcubes. The first iteration determines point (*a,b,c*) to be in the top-left subcube. This subcube then becomes the cube for the next iteration, which determines the point (*a,b,c*) to be in the bottom-left subcube. The following iteration then selects the top-back subcube and the final iteration selects the bottom-right subcube. The point (*a,b,c*) is the origin of this subcube, and its value is the output (*x,y,z*).

Note, the above process is for a three dimensional input space. The same approach can be used for any dimensional input space. In a one dimensional space, two values will be returned from the table lookup, forming a line. For each iteration the line is divided into two segments, and the segment containing the point is selected. This is effectively a binary search. For the two dimensional space, four points are returned from the lookup, forming a square. This square is divided into four sub-squares and the sub-square which contains the point is then used for the next iteration. A four dimensional space will work with hyper-cubes, in which each iteration will subdivide the hypercube into sixteen sub-hyper-cubes. It should be noted that the number of sub-components that can be selected in each iteration is a function of the number of dimensions, d, and is equal to 2^d.

1.2.1 Subcube Selection

In the Cube Subdivision process, the ability to determine which subcube the point (*a,b,c*) is in is needed. If one considers that Cube Subdivision is in effect a multi-dimensional binary search, this task is quite simple. Cube Subdivision works by splitting the cube into two halves in each dimension and selecting the half which contains the point (*a*,*b*,*c*). The selection of the half for the first iteration is performed by examining the most-significant-bit (msb), i.e. bit n-1, of the lower bits (a_l,b_l,c_l) in each input dimension. If the msb is a “0”, then the desired point is in the lower half. Else the msb is a “1” and the desired point is in the upper half. Figure 4 shows the selection of the subcube given lower bits (a_l,b_l,c_l) = (1XXX, 1XXX, 0XXX). The selected subcube then becomes the cube to be subdivided in the next iteration and the next lower significant bit of (a_l,b_l,c_l), i.e. bit n-2, is used for the next subcube selection.

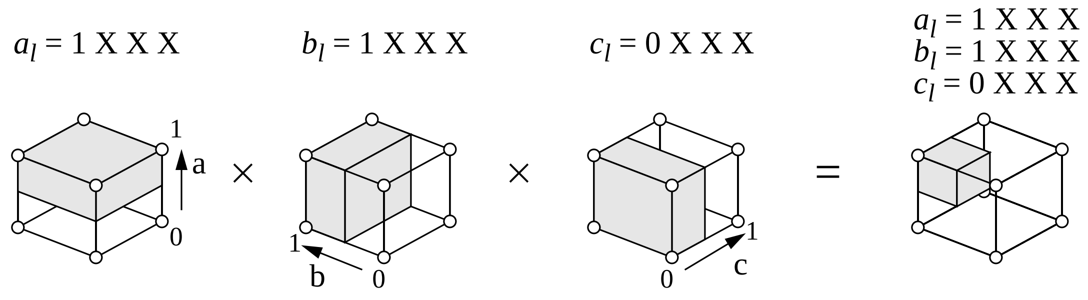
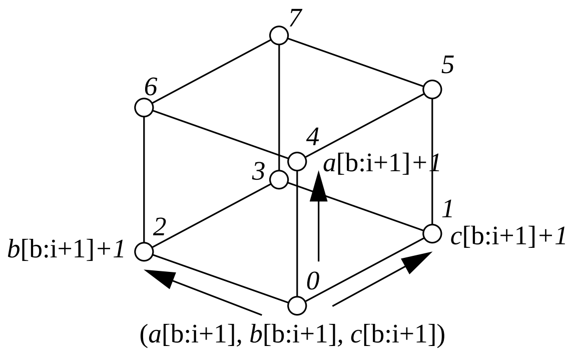


Figure 4. Subcube Selection

It is observed that by numbering the vertices of the cube using the following method simplifies the subcube selection. As shown in Figure 5., the vertices of the cube are numbered such that each dimension is assigned a bit position in a binary encoding. The value of the bit for each position is the offset from the origin of the cube at (*a*[b:i+1],*b*[b:i+1],*c*[b:i+1]) in that dimension. The value b is the number of bits input in each dimension and i is the bit position being interpolated. The value i = n-1 for the first iteration and is decremented after each subdivision iteration.



Vertex Number	0	1	2	3	4	5	6	7
Binary Encoding	000	001	010	011	100	101	110	111
Offset from (*a*[b:i+1],*b*[b:i+1],*c*[b:i+1])	(0,0,0)	(0,0,1)	(0,1,0)	(0,1,1)	(1,0,0)	(1,0,1)	(1,1,0)	(1,1,1)

Figure 5. Cube Vertex Numbering

With this cube numbering, the cube vertex that is closest to (*a*,*b*,*c*) is simply the value of the slice (*a*[i],*b*[i],*c*[i]). Once the closest vertex is known, the subcube which contains the desired point is the subcube that has as one of its vertices, the closest vertex. Figure 6 shows the selected vertex and subcube, for all possible slices across the lower bits.

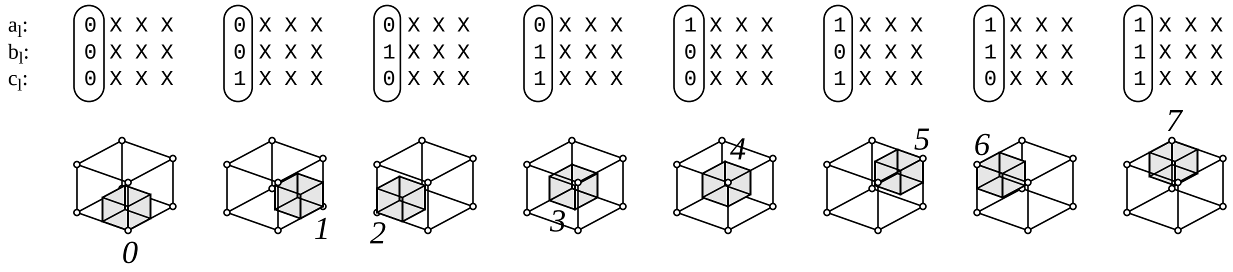


Figure 6. Closest Vertex and Subcube Containing Point (*a,b,c*)

1.2.2 Subcube Generation

Once it is known which subcube contains point (*a,b,c*), the subcube must be generated. Several methods exist for subcube generation, as shown in Figure 7. The differences between the methods are how the points in the center of the faces and center of the cube are computed.

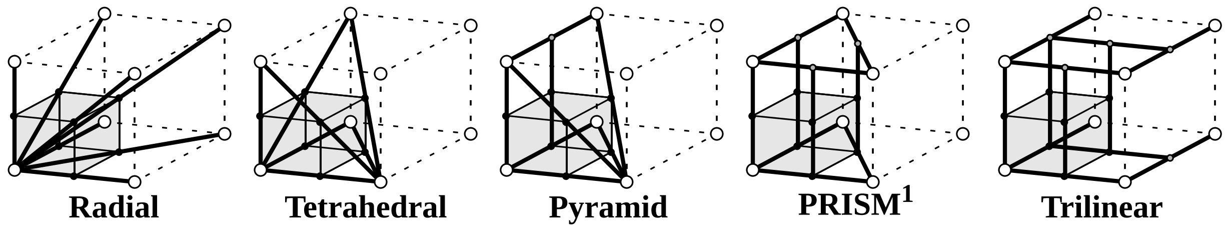


Figure 7. Subcube Generation

Radial (Vondran [22]) is a new approach that offers substantial savings in computation complexity. Radial Interpolation is discussed in detail in section 2.1. Tetrahedral partitions the cube into tetrahedra. Some tetrahedral implementations partition the cube into five tetrahedra (Sakamoto [12] and Sakamoto and Itooka [13]), while others six. Jay Gondek [2] implemented the six tetrahedra subcube generation approach in his presentation. Section 2.2 presents a modified version of the Cube Subdivision with Tetrahedral subcube generation that dramatically reduces the computation complexity of the six tetrahedra interpolation through pruning (Vondran [23]). Finally, Pyramid (Franklin [3]), PRISM[1] (Imao and Ohuchi [5], Kanamori et. al. [6], and Kanamori et. al. [7]) and Trilinear are interpolation methods that can be applied to subcube generation. (A version of PRISM is also known, termed Slant-PRISM, developed by Matsushita for RGB/CMY color spaces.) Although the base concepts of Pyramid, PRISM, and Trilinear can be applied to subcube generation, there is no computation advantage in using these approaches with Cube Subdivision. Thus, Pyramid, Prism, and Trilinear subcube generation methods are not developed further in this document. Other approaches, Clark et. al. [1], Ikegami [4], and Van de Capelle et. al. [15], use mathematical interpolation approaches that do not map to subcube generation and are thus, not presented. Analysis of Tetrahedral, Prism, and Trilinear Interpolations are presented in Kasson et. al. [8] and Stone et. al. [14].

It should be understood that each interpolation method can generate different results. However, because the interpolation process is an approximation and not an exact model of the true behavior, there is no one best method for all cases. Thus, each interpolation method has its place.

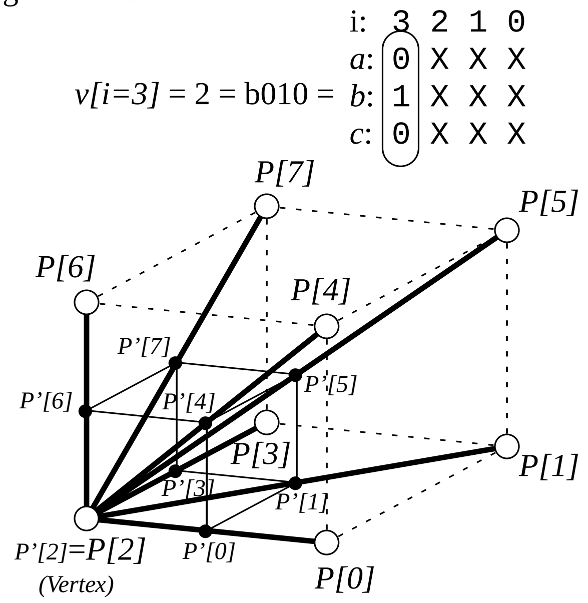
1. PRISM is the acronym *PRogrammable Interpolation by Small Memory*, which also describes its geometry.

2. Cube Subdivision Interpolation Methods

Presented in this section are new Cube Subdivision Interpolation methods based on Radial and Tetrahedral subcube generation. The first is Radial Interpolation. As the name implies, this method is based on the Radial subcube generation. The second, Pruned Tetrahedral, exploits several key observations to reduce the complexity of the six tetrahedra interpolation approach. Also presented is a common implementation that can be used for both Radial and Pruned Tetrahedral Interpolation. Finally, modifications to the Radial and Pruned Tetrahedral implementations are presented that allow for these methods to interpolate when using a non-symmetric color lattice.

2.1 Radial Interpolation

It is observed that the subcube can be generated by averaging the value at the vertex (*a[i],b[i],c[i]*) with the values at all the other vertices of the cube. Figure 8 shows the generation of a subcube using Radial Subcube Generation. Variable *P[v]* denotes the value at vertex *v* of the cube being subdivided and variable *P'[v]* denotes the value at the vertex *v* of the subcube being generated.



v[i] = 4a[i] + 2b[i] + c[i]

Vertex = P[v[i]]

P'[7] = (P[7] + Vertex) / 2

P'[6] = (P[6] + Vertex) / 2

P'[5] = (P[5] + Vertex) / 2

P'[4] = (P[4] + Vertex) / 2

P'[3] = (P[3] + Vertex) / 2

P'[2] = (P[2] + Vertex) / 2

P'[1] = (P[1] + Vertex) / 2

P'[0] = (P[0] + Vertex) / 2

Figure 8. Radial Subcube Generation

Figure 9 shows the cube subdivision process using Radial subcube generation, given n=4 and (a_l,b_l,c_l) = (1010, 1110, 0011), over the four iterations. The division by 2 in each iteration is not performed in order to avoid propagation of rounding errors. Thus, the output (*x,y,z*) is to be divided by 2^n=16 and rounded (not shown).

The dark lines in Figure 9 show the path from the output (*x,y,z*), back. The output (*x,y,z*) is equal to the value at vertex 0 (*P[0]*) of the last subcube. The value of vertex 0 is the sum of the values at vertices 0 and 1 (*P[0]* and *P[v[0]]*, respectively) of the previous subcube. The values at these vertices are the value at vertex 7 (*P[v[1]]*) added to the values at vertices 0 and 1 (*P[0]* and *P[v[0]]*) of the second-to-last subcube. The values at vertices 0, 1, and 7 of the second-to-last subcube are the sum of the value at vertex 2 (*P[v[2]]*) added to the values at vertices 0, 1, and 7 (*P[0]*,

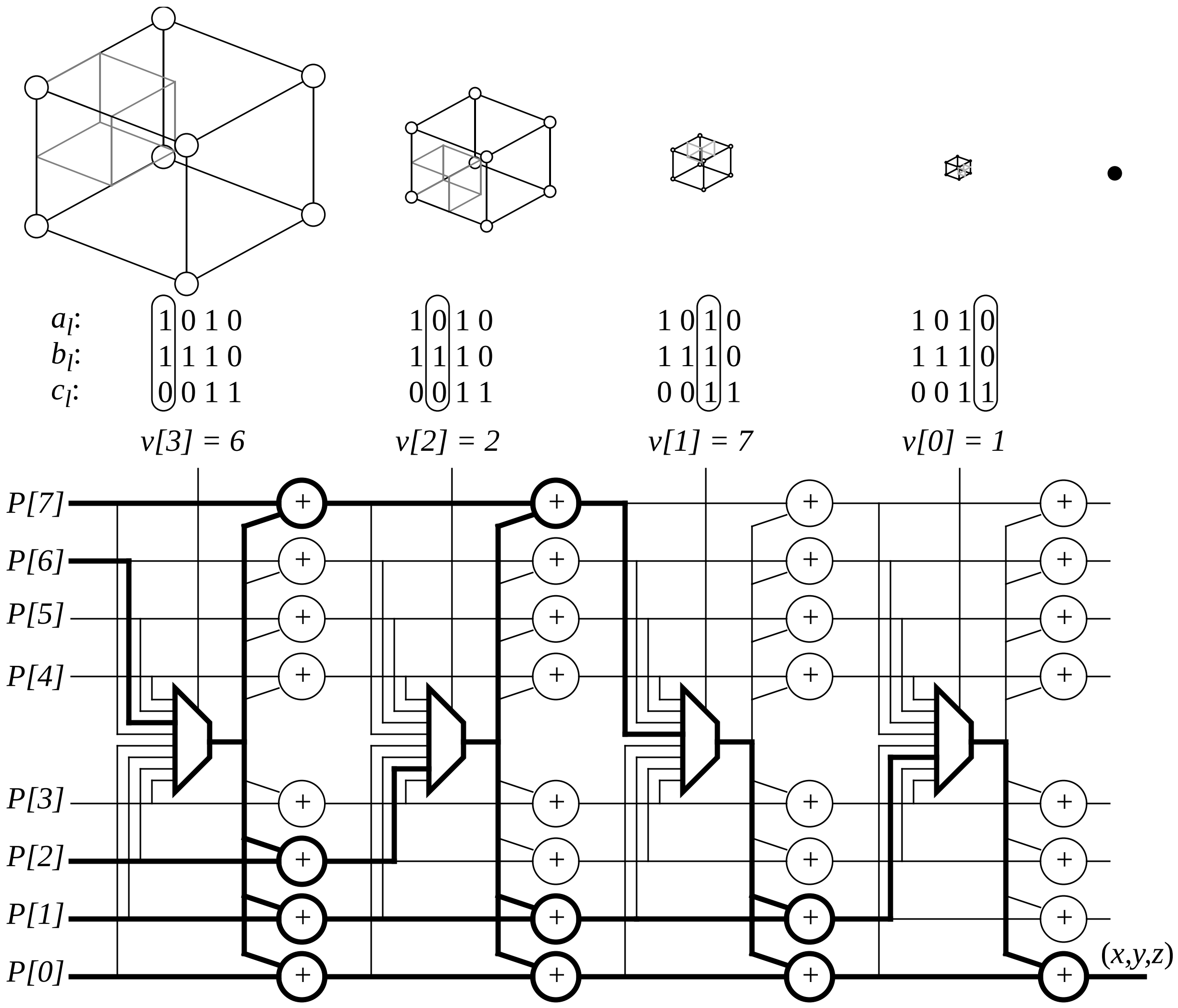


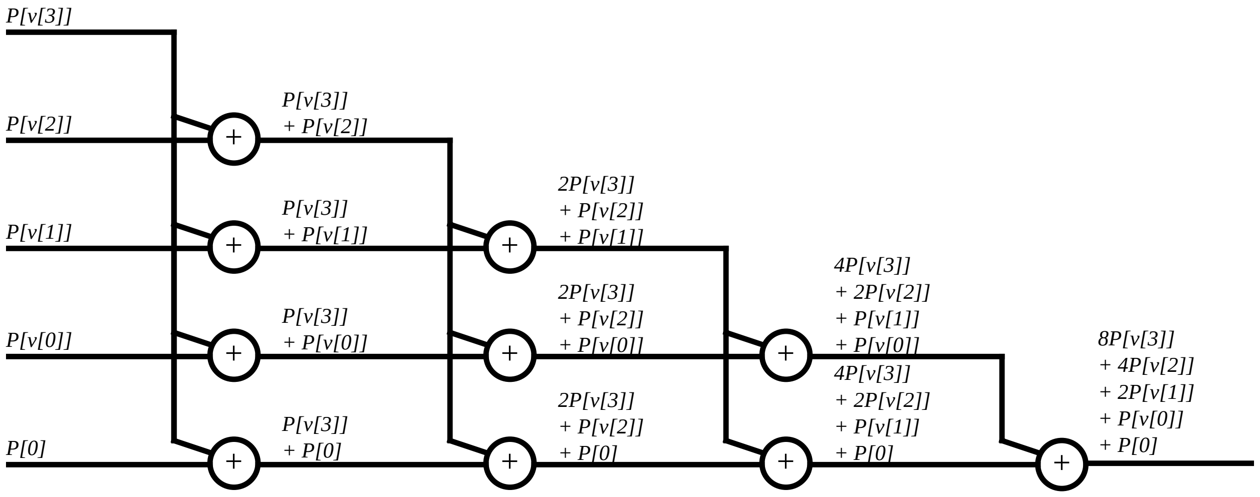
Figure 9. Cube Subdivision Interpolation Using Radial Subcube Generation

P[v[0]], and *P[v[1]]*) of the third-to-last subcube. The values at vertices 0, 1, 2, and 7 of the third-to-last subcube are the sum of the value at vertex 6 (*P[v[3]]*) and vertices 0, 1, 7, and 2 (*P[0]*, *P[v[0]]*, *P[v[1]]*, and *P[v[2]]*) of the cube input from the color table.

Note, only 10 of the 32 adders actually contribute to the output (*x*,*y*,*z*). In fact, only five of the eight vertices contribute to the output (vertices *P[7]*, *P[6]*, *P[2]*, *P[1]* and *P[0]*). Figure 10 shows the pruned implementation of the Cube Subdivision with Radial Cube Generation. Also shown are the values at each stage in the interpolation process.

As can be seen in Figure 10, the output can be computed directly (i.e. *8P[v[3]] + 4P[v[2]] + 2P[v[1]] + P[v[0]] + P[0]*). A general equation can be developed to compute the output, and is presented in Figure 11.

Given d input dimensions, D output dimensions and 2^n values between coarse lattice points, the number of computations is: Each *v[i]* requires d-1 adds and d-1 shifts. There are n vertices *v[i]*



P[v[3]]
P[v[2]]
P[v[1]]
P[v[0]]
P[0]
P[v[3]] + P[v[2]]
P[v[3]] + P[v[1]]
P[v[3]] + P[v[0]]
P[v[3]] + P[0]
2P[v[3]] + P[v[2]] + P[v[1]]
2P[v[3]] + P[v[2]] + P[v[0]]
2P[v[3]] + P[v[2]] + P[0]
4P[v[3]] + 2P[v[2]] + P[v[1]] + P[v[0]]
4P[v[3]] + 2P[v[2]] + P[v[1]] + P[0]
8P[v[3]] + 4P[v[2]] + 2P[v[1]] + P[v[0]] + P[0]

Figure 10. Pruned Cube Subdivision Interpolation Using Radial Subcube Generation

$$\boxed{\left(\sum_{i=0}^{n-1} 2^i P[v[i]]\right) + P[0] \qquad \text{where } v[i] = 4a[i] + 2b[i] + c[i]}$$

$$\begin{aligned} &= 16P[v[4]] + 8P[v[3]] + 4P[v[2]] + 2P[v[1]] + P[v[0]] + P[0] && \text{when n=5} \\ &= 8P[v[3]] + 4P[v[2]] + 2P[v[1]] + P[v[0]] + P[0] && \text{when n=4} \\ &= 4P[v[2]] + 2P[v[1]] + P[v[0]] + P[0] && \text{when n=3} \end{aligned}$$

Figure 11. Radial Interpolation

generated, resulting in 2n(d-1) operations. To compute each output component requires n+1 memory references, n additions, n-1 shifts, plus one additional add and shift to divide and round. With D output components the total number of computations is

$$\begin{aligned} 2n(d-1) + D(n + (n-1) + 1 + 1) &= 2n(d + D - 1) + D \Rightarrow O(n(d + D)) && \text{ALU Operations} \\ \text{and} \qquad & D(n+1) \Rightarrow O(nD) && \text{Memory Accesses} \end{aligned}$$

Radial Interpolation requires the fewest ALU operations and memory accesses of the interpolations presented. In fact, Radial Interpolation is the only interpolation technique known to the author to be linear in n, D, and d in both the number of ALU operations and Memory accesses. Using Radial Interpolation to convert a 24 bit RGB pixel to a 32 bit CMYK pixel with n=4 requires 52 ALU operations and 20 memory accesses.

Shifts and concatenations come for free in hardware, while array indexing results in multiplexers. Thus the hardware complexity is as follows: Each *v[i]* is computed by extracting bit position *i* from each input component and concatenating into a d-bit value, *v[i]*. This operation is free in hardware. To compute each output component requires the selection of n+1 values followed by n additions. The multiplies by 2^i are implemented as shifts and are free. Rounding requires one additional add and a truncation. The truncation is a right shift, which is also free. With D output components the total number of hardware elements is

$$D(n+1) \Rightarrow O(nD) \qquad \text{Adders}$$

$$D(n+1) \Rightarrow O(nD) \qquad \text{Multiplexers } (2^d{:}1)$$

Figure 12 presents a C code implementation of Radial Interpolation. Note, the presented implementation is designed to minimize control flow dependencies, i.e. branches, which are becoming more costly in high performance architectures. It is understood that implementations can be developed to minimize other aspects, e.g. memory accesses, etc.. The implementation operates as follows: Passed in is an array of input components, a color table pointer array, and an array of output components. Each input component is checked if it is the maximum value (i.e. 255). Remember that the last point of each dimension is stored in the color table to allow for interpolation between the last point that is a multiple of 2^n and the final value in the color space. It can be shown that the segment from the last multiple of 2^n and the final value has 2^n-1 steps instead of 2^n, which the interpolation process assumes. Therefore, when the maximum value is observed, its value is incremented by one to assure the final point is reached.

```c
#define C_INC     1
#define B_INC     C_INC*17
#define A_INC     B_INC*17
#define TABLE_INC A_INC*17
/*****************************************/
/** radial_interpolation()             **/
/*****************************************/
void radial_interpolation(unsigned char input[],
unsigned char table[][TABLE_INC], unsigned char output[])
{
    register int       origin, v0, v1, v2, v3 ;
    register int       a, b, c ;
    register int       x, y, z ;

    /*************************************/
    /* Snap max value to last table value */
    /*************************************/
    a = ( input[0] == 0x0ff ) ? 0x100 : (int) input[0] ;
    b = ( input[1] == 0x0ff ) ? 0x100 : (int) input[1] ;
    c = ( input[2] == 0x0ff ) ? 0x100 : (int) input[2] ;

    /**************************************/
    /* Compute Indexes into table         */
    /**************************************/
    origin = A_INC*(a >> 4)
          + B_INC*(b >> 4)
          +       (c >> 4) ;
    v0 = origin + A_INC*( (a & 0x01) )
               + B_INC*( (b & 0x01) )
               +         (c & 0x01 ) ;
    v1 = origin + A_INC*( (a & 0x02) >> 1 )
               + B_INC*( (b & 0x02) >> 1 )
               +         (c & 0x02 ) >> 1) ;
    v2 = origin + A_INC*( (a & 0x04) >> 2 )
               + B_INC*( (b & 0x04) >> 2 )
               +         (c & 0x04) >> 2) ;
    v3 = origin + A_INC*( (a & 0x08) >> 3 )
               + B_INC*( (b & 0x08) >> 3 )
               +         (c & 0x08) >> 3 ) ;

    /*************/
    /* Compute x */
    /*************/
    x = ( ( (int) table[0][origin]
        +   (int) table[0][v0]
        + ( (int) table[0][v1] << 1 )
        + ( (int) table[0][v2] << 2 )
        + ( (int) table[0][v3] << 3 )
        +   0x08 ) >> 4 ) ;        /* Round */

    /*************/
    /* Compute y */
    /*************/
    y = ( ( (int) table[1][origin]
        +   (int) table[1][v0]
        + ( (int) table[1][v1] << 1 )
        + ( (int) table[1][v2] << 2 )
        + ( (int) table[1][v3] << 3 )
        +   0x08 ) >> 4 ) ;        /* Round */

    /*************/
    /* Compute z */
    /*************/
    z = ( ( (int) table[2][origin]
        +   (int) table[2][v0]
        + ( (int) table[2][v1] << 1 )
        + ( (int) table[2][v2] << 2 )
        + ( (int) table[2][v3] << 3 )
        +   0x08 ) >> 4 ) ;        /* Round */

    output[0] = (char) x ;
    output[1] = (char) y ;
    output[2] = (char) z ;
}
```

Figure 12. Radial Interpolation C Implementation with n=4

Next, the indices into the color table are generated. The indices include the *origin* (i.e. point 0), *v[0]*, *v[1]*, *v[2]*, and *v[3]*. Note, in this implementation, the values, origin, v0, v1, v2, and v3, are actually the indices into the color table and not just the offset from the origin. This was done to simplify the presented implementation. Once the indices are generated, the output components are computed using the equation *8P[v[3]] + 4P[v[2]] + 2P[v[1]] + P[v[0]] + P[0],* where *P[]* is table[0], table[1], or table[2] for the respective component being generated.

Note, it is understood that processor specific code optimizations can be used. For example, by packing all the components for a lattice entry into the same word, (i.e. table[entry][color component]) the number of memory references can be reduced. Also the number of ALU operations can be reduced by exploiting the 32 bit ALU by packing the x values in bits 16 through 23 and y values in bits 0 through 7. This will allow a single sequence of shifts and adds (e.g. *8P[v[3]] + 4P[v[2]] + 2P[v[1]] + P[v[0]] + P[0]*) to generate the x and y components in parallel. These optimizations are further discussed in Appendix B.

Figure 13 presents the VHDL synthesizable implementation of Radial Interpolation. This implementation has an array of components as input and an array of components as output. Instead of providing an array of pointers to the color table, the eight points of the selected cube are supplied. This is done because in the hardware implementation, additional functions are performed regarding the RAM access that are not associated with the interpolation process (discussed in Vondran [16] and Vondran [17]). The cleanest implementation in our case was to separate the interpolation process from the table access. Also handled in the table access process is the maximum value boundary condition, previously discussed. In the table access process the cube is collapsed in the dimensions in which the input equals 255. For example, if a=0xff, the table access process will collapse the cube by equating *P[0]=P[4], P[1]=P[5], P[2]=P[6],* and *P[3]=P[7].*

The VHDL implementation consists of three processes. The first, VERTICES, generates vertex variables, corresponding to *v[0]* through *v[3]*. Next, the MUX process multiplexes and shifts the values selected by the vertex values and point 0, and generates the rounding constant. Note, the shift is performed by the i value in the term "(7+i downto i)". The SUM process then adds these values and truncates to eight bits. Note, to minimize propagation delay through the SUM process, the process is implemented by successive pairing and parallel additions. For the six SUM input case (P[0], P[v[0], ..., P[v[3], and Rounding Constant), this results in 3 adds followed by 1 add, followed by 1 add. It is suggested that because the synthesis time and quality are adversely affected as the complexity of the design increases, the SUM process should be a separate module during synthesis. Lastly, it should be understood that the complex operations (multiplies *, divides /, modulus mod, for loops, etc.) in the implementation are performed during synthesis and are not implemented in gates. The implementation is designed to be used for any value of n and d, i.e. NUM_INTERP_BITS and NUM_OUTPUT_DIMEN.

2.2 Pruned Tetrahedral Interpolation

RGB and CMY color spaces have the property that the luminance axis is the diagonal through the center of the cube. It has been found that biasing this axis produces more pleasing output in certain situations. Figure 14 shows the luminance axis. Also shown is the partitioning of the cube into six tetrahedra. The cube is partitioned into tetrahedra by dividing the cube along all the axes radiating from the White and Black vertices to all other vertices.

Tetrahedral Interpolation biases the axes by using the tetrahedra edges to calculate the subcube vertices. Figure 15 shows the generation of a subcube from the tetrahedra. By observation, it is determined that for all cases, the Tetrahedral Interpolation value *P'[k]*, for subcube vertex k, is

```
-----------------------------------------------------
-- Radial Interpolation.                          --
-----------------------------------------------------
entity radial is
port (
     -- Input Pixel (a,b,c) --
     input:         in    INPUT_ARRAY ;
     -- Vertex Points of Cube from Color Table --
     point:         in    POINT_ARRAY ;
     -- Output Pixel (x,y,z) --
     output:        out   OUTPUT_ARRAY
) ;
end radial ;

---------------------------------------------
architecture radialrtl of radial is
---------------------------------------------

     signal  vertex:       VERTEX_ARRAY ;
     signal  point_value:  POINT_VALUE_ARRAY ;

begin

---------------------------------------------------------
-- Compute selected vertices from slices across input.
---------------------------------------------------------
VERTICIES: process ( input )
begin
     ---------------------------------------------
     -- For each slice across the input lower bits --
     ---------------------------------------------
     for i in NUM_INTERP_BITS-1 downto 0 loop
          vertex(i) <= input(0)(i) & input(1)(i)
                                   & input(2)(i);
     end loop ;
end process ;

-------------------------------------------------
-- Select the values of vertices to be used by --
-- interpolation.  The last two vertices are   --
-- always point(0) and the rounding constant.  --
-------------------------------------------------
MUX: process ( vertex, point )
begin
     ------------------------------
     -- For each output dimension --
     ------------------------------
     for d in 0 to NUM_OUTPUT_DIMEN-1 loop
        ----------------
        -- Initialize --
        ----------------
        point_value(d) <= (others => (others => ‘0’));

        -----------------------------------------
        -- point_value( 0 : NUM_INTERP_BITS-1 ) --
        -----------------------------------------
        for i in 0 to NUM_INTERP_BITS-1 loop
          case vertex(i) is
             when “000” => point_value(d)(i)(7+i downto i)
                                      <= point(d)(0) ;
             when “001” => point_value(d)(i)(7+i downto i
                                      <= point(d)(1) ;
             when “010” => point_value(d)(i)(7+i downto i)
                                      <= point(d)(2) ;
             when “011” => point_value(d)(i)(7+i downto i)
                                      <= point(d)(3) ;
             when “100” => point_value(d)(i)(7+i downto i)
                                      <= point(d)(4) ;
             when “101” => point_value(d)(i)(7+i downto i)
                                      <= point(d)(5);
             when “110” => point_value(d)(i)(7+i downto i)
                                      <= point(d)(6);
            when others => point_value(d)(i)(7+i downto i)
                                      <= point(d)(7);
          end case ;
```

```
        -----------------
        -- point(d)(0) --
        -----------------
        point_value(d)(NUM_INTERP_BITS)(7 downto 0)
                                      <= point(d)(0) ;
        ----------------------
        -- Rounding constant --
        ----------------------
        point_value(d)(NUM_INTERP_BITS+1)
                         (NUM_INTERP_BITS-1) <= ‘1’ ;
        end loop ;
     end loop ;
end process ;

-----------------------------------------------------
-- SUM adds the selected vertices values and trun- --
-- cates to eight bits. The process is implemented --
-- to minimize time by successive pairing and      --
-- adding.                                         --
-----------------------------------------------------
SUM: process ( point_value )
     variable  temp:       POINT_VALUES ;
     variable  bound:      INTEGER ;
begin
     ------------------------------
     -- For each output dimension --
     ------------------------------
     for d in 0 to NUM_OUTPUT_DIMEN-1 loop
        ------------------------------
        -- Initialize temp and bound. --
        ------------------------------
        temp := point_value(d) ;
        bound := NUM_INTERP_BITS + 2 ;
        ------------------------------------------
        -- Loop for ceil(log2(NUM_INTERP_BITS-1)). --
        ------------------------------------------
        for i in 0 to NUM_INTERP_BITS-1 loop
             ------------------------------------
             -- Pair temp array elements and add. --
             ------------------------------------
             for j in 0 to (NUM_INTERP_BITS/2) loop
                  ------------------------------
                  -- Step 1: Move temp(even’s) --
                  ------------------------------
                  if ( 2*j < bound ) then
                       temp(j) := temp(2*j) ;
                  else
                       temp(j) := (others => ‘0’) ;
                  end if ;
                  -----------------------------
                  -- Step 2: Add temp(odd’s) --
                  -----------------------------
                  if ( (2*j)+1 < bound ) then
                       temp(j) := temp(j)
                                  + temp((2*j)+1);
                  else
                       temp(j) := temp(j) ;
                  end if ;
             end loop ;
             ------------------------------
             -- Set bound for next iteration --
             ------------------------------
             bound := ( bound / 2 ) + ( bound mod 2 ) ;
        end loop ;
        ---------------------
        -- Truncate output. --
        ---------------------
        output(d) <= temp(0)(7+NUM_INTERP_BITS
                             downto NUM_INTERP_BITS) ;
     end loop ;
end process ;

---------------------------------------------------------
end radialrtl;
---------------------------------------------------------
```

Figure 13. Radial Interpolation VHDL Implementation with n=NUM_INTERP_BITS

equal to the average of the subdivided cube values $P[k \,\&\, v[i]]$ and $P[k \mid v[i]]$. Note, characters “&” and “|” denote the bitwise AND and OR operations, respectively.

Figure 16 shows the Pruned Tetrahedral Interpolation for n=4. *P’[]* denotes the values of the sub-

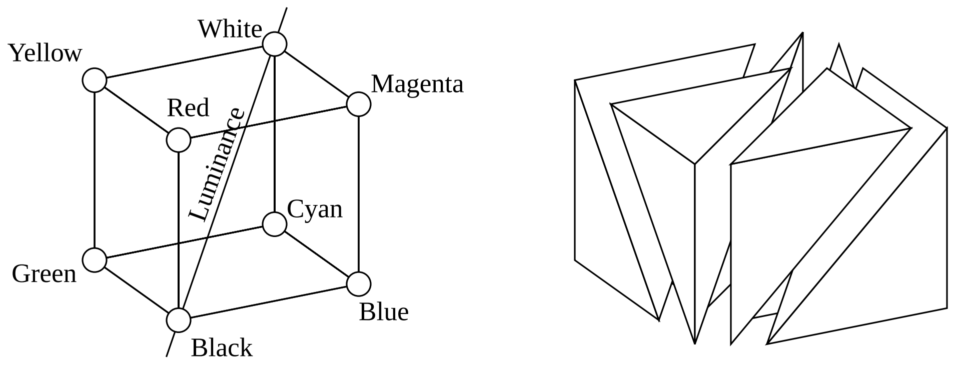
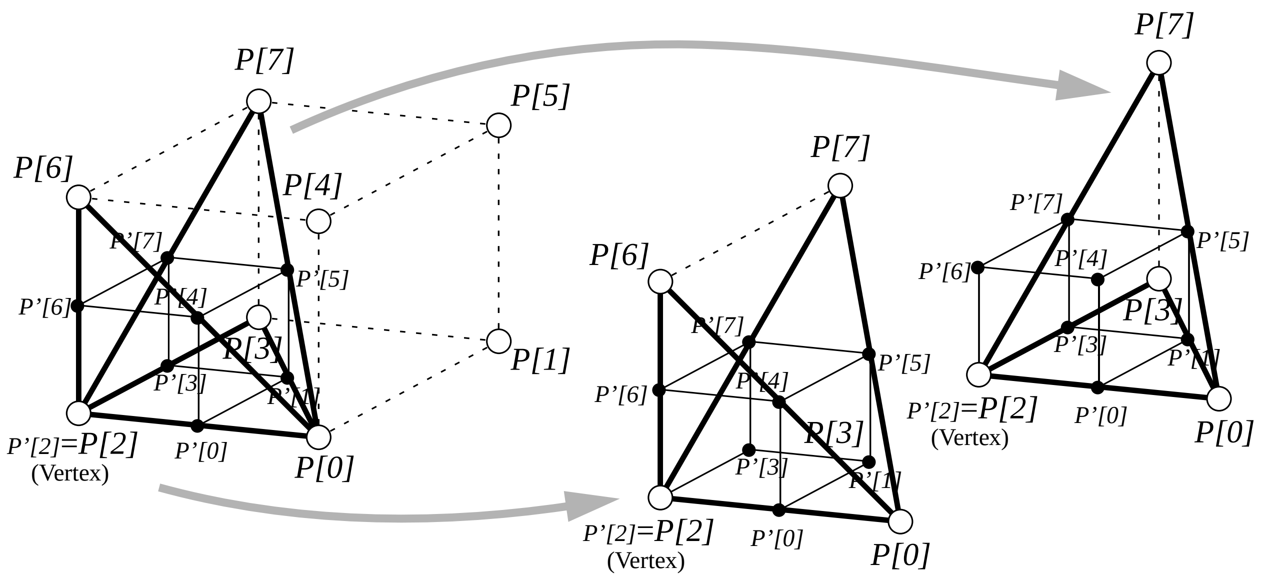


Figure 14. Luminance Axis in RGB/CMY Color Space and Tetrahedral Cube Partitioning



$P'[7] = (P[v[i] \& 7] + P[v[i] \mid 7]) / 2$

$= (P[v[i]] + P[7]) / 2$

$P'[6] = (P[v[i] \& 6] + P[v[i] \mid 6]) / 2$

$P'[5] = (P[v[i] \& 5] + P[v[i] \mid 5]) / 2$

$P'[4] = (P[v[i] \& 4] + P[v[i] \mid 4]) / 2$

$P'[3] = (P[v[i] \& 3] + P[v[i] \mid 3]) / 2$

$P'[2] = (P[v[i] \& 2] + P[v[i] \mid 2]) / 2$

$P'[1] = (P[v[i] \& 1] + P[v[i] \mid 1]) / 2$

$P'[0] = (P[v[i] \& 0] + P[v[i] \mid 0]) / 2$

$= (P[0] + P[v[i]]) / 2$

where $v[i] = 4a[i] + 2b[i] + c[i]$

Figure 15. Tetrahedral Subcube Generation

cube vertices after the first subdivision iteration. *P”[]* denotes the values of the subcube vertices after the second subdivision iteration. And *P’”[]* and *P””[]* denote the values of the vertices after the third and fourth iterations, respectively. Note, the division by 2 in each iteration is not performed in order to avoid propagation of rounding errors. Thus the output (*x*,*y*,*z*) is to be divided by 2^n=16 and rounded (not shown).

Given d input dimensions, D output dimensions and 2^n values between coarse lattice points, the

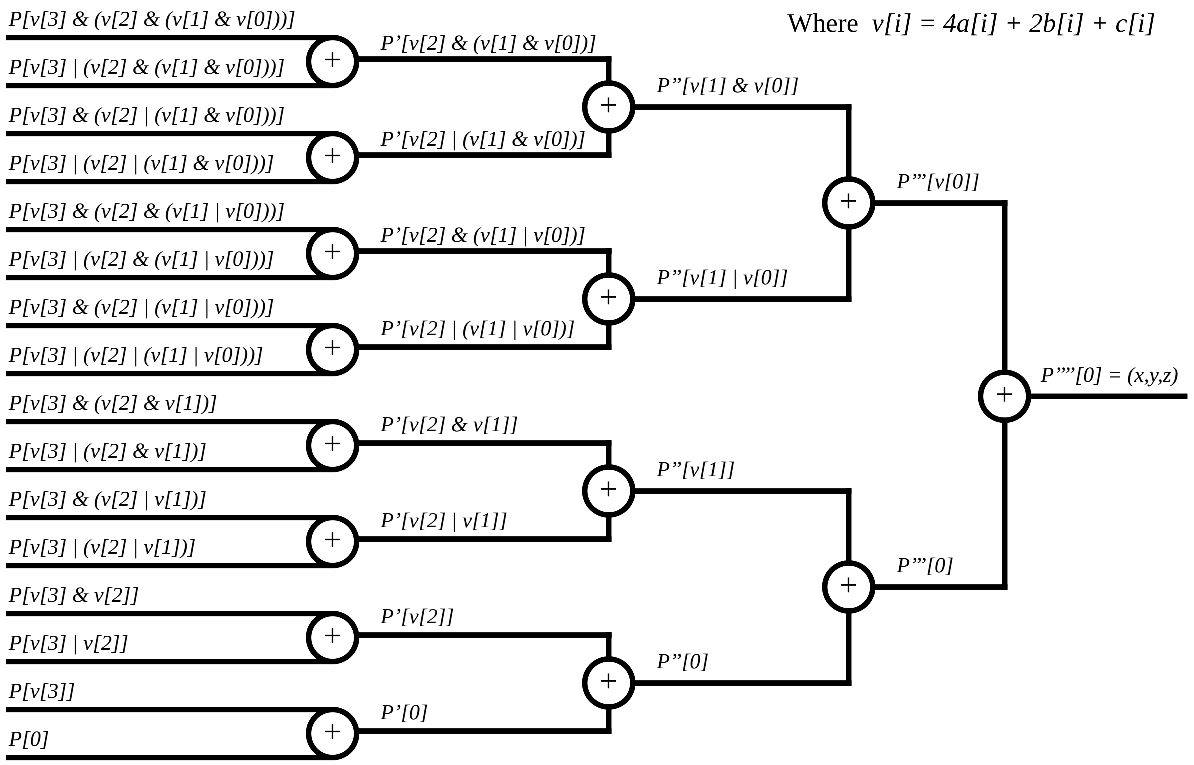


Figure 16. Pruned Tetrahedral Interpolation

number of computations is: Each *v[i]* requires d adds and d-1 shifts. There are n vertices *v[i]* generated, resulting in n(2d-1) operations. From this, 2^n memory indexes are computed, some as simple as “0” and “*v[n-1]*”, while others are a series of ANDs and ORs. One form of reduction is that common subexpressions exists in the index computations (e.g. “*v[1] & v[0]*” is used in many expressions in the example). The number of logic operations to compute the indices is equal to

$$2\sum_{i=1}^{n-1}(2^i-1) = 2(1+3+7+\ldots)$$

As can be seen in Figure 16, 2^n-1 additions are used. One additional add and shift is needed per output component to divide and round, totaling $(2^n-1)+1+1 = 2^n+1$. Also in Figure 16, 2^n memory references are shown. However, Pruned Tetrahedral Interpolation can be implemented such that a cap is placed on the number of memory accesses at the number of vertices of the cube (2^d vertices are on the cube). Thus, the minimum of the two is used, i.e. $\min(2^d, 2^n)$. With D output components the total number of computations is

$$n(2d-1) + 2\sum_{i=1}^{n-1}(2^i-1) + D(2^n+1) \Rightarrow O(nd + 2^n D) \qquad \text{ALU Operations}$$

$$\text{and} \qquad D(min(2^d, 2^n)) \Rightarrow O(D(min(2^d, 2^n))) \qquad \text{Memory Accesses}$$

Using Pruned Tetrahedral Interpolation to convert a 24 bit RGB pixel to a 32 bit CMYK pixel with n=4 requires 110 ALU operations and 32 memory accesses. Clearly, biasing the luminance axis comes at the cost of more operations (unbiased Radial Cube Subdivision Interpolation requires a total of 52 ALU operations and 20 memory accesses for the same conversion).

Shifts and concatenations come for free in hardware, while array indexing results in multiplexers. Thus, the hardware complexity is as follows: Each *v[i]* is computed by extracting bit position *i* from each input component and concatenating into a d-bit value, *v[i]*. The number of AND/OR gates to compute the indices is equal to

$$d\left(2\sum_{i=1}^{n-1}(2^i-1)\right) = 2d(1+3+7+\ldots)$$

To compute each output component requires the selection of 2^n-1 values followed by 2^n-1 additions. Rounding requires one additional add and a truncation. The truncation is a right shift, which is free. With D output components the total number of hardware elements is

$$D((2^n-1)+1) = D(2^n) \Rightarrow O(2^nD) \qquad \text{Adders}$$

$$D(2^n-1) \Rightarrow O(2^nD) \qquad \text{Multiplexers } (2^d{:}1)$$

$$d\left(2\sum_{i=1}^{n-1}(2^i-1)\right) \Rightarrow O(2^nd) \qquad \text{Additional AND/OR Gates}$$

Figure 17 presents the C implementation for Pruned Tetrahedral Interpolation. The implementation operates as follows: Passed in is an array of input components, an array of color table pointers, and an array of output components. Each input component is snapped to the last table value if the input is the maximum value (i.e. 255) in that dimension. Next, the selected vertices, v0 through v3, and indices into the color table are generated. Finally the table is accessed and the values summed, rounded and truncated to eight bits.

Figure 18 presents the VHDL implementation of Pruned Tetrahedral Interpolation. The VHDL implementation consists of three processes. The first, VERTICES, generates the vertex variables and performs the bitwise AND and OR operations to produce the vertex selections for the MUX process. The MUX process multiplexes the values selected by the vertex values and point 0, and generates the rounding constant. The SUM process then adds these values and truncates to eight bits. Note, it should be understood that the complex operations (multiplies *, divides /, modulus mod, for loops, etc.) in the implementation are performed during synthesis and are not implemented in gates.

2.3 Common Radial and Pruned Tetrahedral Implementation

A user may wish to use Pruned Tetrahedral for conversions of RGB and CMY pixels, and Radial Interpolation for all others (CieLab, LUV, YC_bC_r, etc.). In software this is easily done by calling a different routine. Separate implementations in hardware, however, mean separate logic that may be idle when the other interpolation is in use. Thus it is desirable to develop a common implementation that can be used for both Radial or Pruned Tetrahedral Interpolation. Figure 19 (Vondran [19]) presents the common implementation for n=4.

Notice that the difference between the two types of interpolation is the index of the vertices. Because the number of bits of *v[]* is generally less than the number of bits of variable *P[]*, it is better to multiplex the indices prior to the memory access. Also note that the last two values (*P[v[3]]* and *P[0]*) are the same and will be for all values of n. Thus, a common Radial and

```
#define C_INC     1
#define B_INC     C_INC*17
#define A_INC     B_INC*17
#degine TABLE_INC A_INC*17
/***********************************************/
/** pruned_tetrahedral_interpolation()         **/
/***********************************************/
void pruned_tetrahedral_interpolation(
  unsigned char input[], unsigned char table[][TABLE_INC],
  unsigned char output[] )
{
 register int  a, b, c, x, y, z ;
 register char v0, v1, v2, v3;
 register char v1_and_v0, v2_and_v1, v3_and_v2 ;
 register char v1_or_v0,  v2_or_v1,  v3_or_v2 ;
 register char v2_and_v1_and_v0, v3_or_v2_or_v1_or_v0;
 register char v2_and_v1_or_v0, v3_or_v2_or_v1_and_v0;
 register char v2_or_v1_and_v0, v3_or_v2_and_v1_or_v0;
 register char v2_or_v1_or_v0, v3_or_v2_and_v1_and_v0;
 register char v3_and_v2_and_v0, v3_and_v2_or_v1_or_v0;
 register char v3_and_v2_or_v0, v3_and_v2_or_v1_and_v0;
 register char v3_or_v2_and_v0, v3_and_v2_and_v1_or_v0;
 register char v3_or_v2_or_v0, v3_and_v2_and_v1_and_v0;
 register int  index0,  index1,  index2,  index3 ;
 register int  index4,  index5,  index6,  index7 ;
 register int  index8,  index9,  index10, index11 ;
 register int  index12, index13, index14, index15 ;

   /**************************************/
   /* Snap last value to last table value */
   /**************************************/
   a = ( input[0] == 0x0ff ) ? 0x100 : (int) input[0] ;
   b = ( input[1] == 0x0ff ) ? 0x100 : (int) input[1] ;
   c = ( input[2] == 0x0ff ) ? 0x100 : (int) input[2] ;
   /*********************************************/
   /* Compute slices across the input components */
   /*********************************************/
   v0 = ( (a & 0x01) << 2 )
      + ( (b & 0x01) << 1 ) +   (c & 0x01) ;
   v1 = ( (a & 0x02) << 1 )
      +   (b & 0x02)        + ( (c & 0x02) >> 1 ) ;
   v2 =   (a & 0x04)
      + ( (b & 0x04) >> 1 ) + ( (c & 0x04) >> 2 ) ;
   v3 = ( (a & 0x08) >> 1 )
      + ( (b & 0x08) >> 2 ) + ( (c & 0x08) >> 3 ) ;
   /******************************/
   /* Compute offset from origin. */
   /******************************/
   v1_and_v0              = v1 & v0 ;
   v1_or_v0               = v1 | v0 ;
   v2_and_v1              = v2 & v1 ;
   v2_and_v1_and_v0       = v2 & v1_and_v0 ;
   v2_and_v1_or_v0        = v2 & v1_or_v0 ;
   v2_or_v1               = v2 | v1 ;
   v2_or_v1_and_v0        = v2 | v1_and_v0 ;
   v2_or_v1_or_v0         = v2 | v1_or_v0 ;
   v3_and_v2              = v3 & v2 ;
   v3_and_v2_and_v1       = v3 & v2_and_v1 ;
   v3_and_v2_and_v1_and_v0 = v3 & v2_and_v1_and_v0 ;
   v3_and_v2_and_v1_or_v0 = v3 & v2_and_v1_or_v0 ;
   v3_and_v2_or_v1        = v3 & v2_or_v1 ;
   v3_and_v2_or_v1_and_v0  = v3 & v2_or_v1_and_v0 ;
   v3_and_v2_or_v1_or_v0   = v3 & v2_or_v1_or_v0 ;
   v3_or_v2               = v3 | v2 ;
   v3_or_v2_and_v1        = v3 | v2_and_v1 ;
   v3_or_v2_and_v1_and_v0  = v3 | v2_and_v1_and_v0 ;
   v3_or_v2_and_v1_or_v0   = v3 | v2_and_v1_or_v0 ;
   v3_or_v2_or_v1         = v3 | v2_or_v1 ;
   v3_or_v2_or_v1_and_v0   = v3 | v2_or_v1_and_v0 ;
   v3_or_v2_or_v1_or_v0    = v3 | v2_or_v1_or_v0 ;
   /******************************/
   /* Generate indices into table */
   /******************************/
   index0  = ( (a >> 4) * A_INC )
     +  ( (b >> 4) * B_INC ) + (c >> 4) ;
   index1  = index0
     + A_INC*((v3 & 0x0004) >> 2)
     + B_INC*((v3 & 0x0002) >> 1)
     +        (v3 & 0x0001) ;
   index2  = index0
     + A_INC*((v3_and_v2 & 0x0004) >> 2)
     + B_INC*((v3_and_v2 & 0x0002) >> 1)
     +        (v3_and_v2 & 0x0001) ;
   index3  = index0
     + A_INC*((v3_and_v2_and_v1 & 0x0004) >> 2)
     + B_INC*((v3_and_v2_and_v1 & 0x0002) >> 1)
     +        (v3_and_v2_and_v1 & 0x0001) ;
   index4  = index0
     + A_INC*((v3_and_v2_and_v1_and_v0 & 0x0004) >> 2)
     + B_INC*((v3_and_v2_and_v1_and_v0 & 0x0002) >> 1)
     +       (v3_and_v2_and_v1_and_v0 & 0x0001) ;
   index5  = index0
     + A_INC*((v3_and_v2_and_v1_or_v0 & 0x0004) >> 2)
     + B_INC*((v3_and_v2_and_v1_or_v0 & 0x0002) >> 1)
     +        (v3_and_v2_and_v1_or_v0 & 0x0001) ;
   index6  = index0
     + A_INC*((v3_and_v2_or_v1 & 0x0004) >> 2)
     + B_INC*((v3_and_v2_or_v1 & 0x0002) >> 1)
     +        (v3_and_v2_or_v1 & 0x0001) ;
   index7  = index0
     + A_INC*((v3_and_v2_or_v1_and_v0 & 0x0004) >> 2)
     + B_INC*((v3_and_v2_or_v1_and_v0 & 0x0002) >> 1)
     +        (v3_and_v2_or_v1_and_v0 & 0x0001) ;
   index8  = index0
     + A_INC*((v3_and_v2_or_v1_or_v0 & 0x0004) >> 2)
     + B_INC*((v3_and_v2_or_v1_or_v0 & 0x0002) >> 1)
     +        (v3_and_v2_or_v1_or_v0 & 0x0001) ;
   index9  = index0
     + A_INC*((v3_or_v2 & 0x0004) >> 2)
     + B_INC*((v3_or_v2 & 0x0002) >> 1)
     +        (v3_or_v2 & 0x0001) ;
   index10 = index0
     + A_INC*((v3_or_v2_and_v1 & 0x0004) >> 2)
     + B_INC*((v3_or_v2_and_v1 & 0x0002) >> 1)
     +        (v3_or_v2_and_v1 & 0x0001) ;
   index11 = index0
     + A_INC*((v3_or_v2_and_v1_and_v0 & 0x0004) >> 2)
     + B_INC*((v3_or_v2_and_v1_and_v0 & 0x0002) >> 1)
     +        (v3_or_v2_and_v1_and_v0 & 0x0001) ;
   index12 = index0
     + A_INC*((v3_or_v2_and_v1_or_v0 & 0x0004) >> 2)
     + B_INC*((v3_or_v2_and_v1_or_v0 & 0x0002) >> 1)
     +        (v3_or_v2_and_v1_or_v0 & 0x0001) ;
   index13 = index0
     + A_INC*((v3_or_v2_or_v1 & 0x0004) >> 2)
     + B_INC*((v3_or_v2_or_v1 & 0x0002) >> 1)
     +        (v3_or_v2_or_v1 & 0x0001) ;
   index14 = index0
     + A_INC*((v3_or_v2_or_v1_and_v0 & 0x0004) >> 2)
     + B_INC*((v3_or_v2_or_v1_and_v0 & 0x0002) >> 1)
     +        (v3_or_v2_or_v1_and_v0 & 0x0001) ;
   index15 = index0
     + A_INC*((v3_or_v2_or_v1_or_v0 & 0x0004) >> 2)
     + B_INC*((v3_or_v2_or_v1_or_v0 & 0x0002) >> 1)
     +        (v3_or_v2_or_v1_or_v0 & 0x0001) ;
   /*************/
   /* Compute x */
   /*************/
   x = ( ( table[0][index0]  + table[0][index1]
         + table[0][index2]  + table[0][index3]
         + table[0][index4]  + table[0][index5]
         + table[0][index6]  + table[0][index7]
         + table[0][index8]  + table[0][index9]
         + table[0][index10] + table[0][index11]
         + table[0][index12] + table[0][index13]
         + table[0][index14] + table[0][index15]
         + 0x08 ) >> 4 );                /*Round*/
   /*************/
   /* Compute y */
   /*************/
   y = ( ( table[1][index0]  + table[1][index1]
         + table[1][index2]  + table[1][index3]
         + table[1][index4]  + table[1][index5]
         + table[1][index6]  + table[1][index7]
         + table[1][index8]  + table[1][index9]
         + table[1][index10] + table[1][index11]
         + table[1][index12] + table[1][index13]
         + table[1][index14] + table[1][index15]
         + 0x08 ) >> 4 ) ;               /*Round*/
   /*************/
   /* Compute z */
   /*************/
   z = ( ( table[2][index0]  + table[2][index1]
         + table[2][index2]  + table[2][index3]
         + table[2][index4]  + table[2][index5]
         + table[2][index6]  + table[2][index7]
         + table[2][index8]  + table[2][index9]
         + table[2][index10] + table[2][index11]
         + table[2][index12] + table[2][index13]
         + table[2][index14] + table[2][index15]
         + 0x08 ) >> 4 ) ;               /*Round*/
   /*********************/
    * Write back results */
   /*********************/
   output[0] = (char) x ;
   output[1] = (char) y ;
   output[2] = (char) z ;
}
```

Figure 17. Pruned Tetrahedral Interpolation C Implementation with n=4

```
----------------------------------------------------
-- Pruned Tetrahedral Interpolation              --
----------------------------------------------------
entity pruned_tetrahedral is
port (
     -- Input Pixel --
     input:         in    INPUT_ARRAY ;
     -- Vertex Points of Cube from Color Table --
     point:         in    POINT_ARRAY ;
     -- Output Pixel --
     output:        out   OUTPUT_ARRAY
) ;
end pruned_tetrahedral ;

-----------------------------------------------------
architecture pruned_tetrtl of pruned_tetrahedral is
---------------------- ----------------------------

 constant VERTEX_COUNT: INTEGER := 2**NUM_INTERP_BITS;
 signal   vertex:       VERTEX_ARRAY ;
 signal   point_value:  POINT_VALUE_ARRAY ;

begin

-----------------------------------------------------
-- Compute vertices.                              --
-----------------------------------------------------
VERTICIES: process ( input )
     variable  temp1, temp2:   VERTEX_ARRAY ;
     variable  limit:          INTEGER ;
begin
     ----------------------
     -- Initialize Array --
     ----------------------
     temp2 := ( others => “000” ) ;
     ----------------------------------------------
     -- For each slice across the interpolate bits --
     ----------------------------------------------
     for i in 0 to NUM_INTERP_BITS-1 loop
          ---------------------------------
          -- limit is 0, 2, 6, 14, 30, ... --
          ---------------------------------
          limit := ( 2**(i+1) ) - 2 ;
          -----------------------------------
          -- Load slice into limit location --
          -----------------------------------
          temp2(limit) := input(0)(i) & input(1)(i)
                                      & input(2)(i);
          temp1 := temp2 ;
          ----------------------------
          -- for all values in temp1 --
          ----------------------------
          for j in (VERTEX_COUNT/2)-1 downto 0 loop
               if ( j < limit/2 ) then
                    temp2(2*j)     := temp1(limit)
                                   and temp1(j) ;
                    temp2((2*j)+1) := temp1(limit)
                                   or  temp1(j) ;
               else
                    temp2 := temp2 ;
               end if ;
          end loop ;
     end loop ;
     --------------------------
     -- Move into index array --
     --------------------------
     vertex <= temp2 ;
end process ;

-----------------------------------------------------
-- Select the values of vertices to be used by the --
-- interpolation. The last two vertices are always --
-- point(0) and the rounding constant.             --
-----------------------------------------------------
MUX: process ( vertex, point )
begin
     ------------------------------
     -- For each output dimension --
     ------------------------------
     for d in 0 to NUM_OUTPUT_DIMEN-1 loop
        ---------------
        -- Initialize --
        ---------------
        point_value(d) <= (others => (others => ‘0’));
        -------------------------------------
        -- point_value(d)(0 : VERTEX_COUNT-2) --
        -------------------------------------
        for i in 0 to VERTEX_COUNT-2 loop
          case vertex(i) is
```

```
             when “000”  => point_value(d)(i)(7 downto 0)
                                       <= point(d)(0) ;
             when “001”  => point_value(d)(i)(7 downto 0)
                                       <= point(d)(1) ;
             when “010”  => point_value(d)(i)(7 downto 0)
                                       <= point(d)(2) ;
             when “011”  => point_value(d)(i)(7 downto 0)
                                       <= point(d)(3) ;
             when “100”  => point_value(d)(i)(7 downto 0)
                                       <= point(d)(4) ;
             when “101”  => point_value(d)(i)(7 downto 0)
                                       <= point(d)(5) ;
             when “110”  => point_value(d)(i)(7 downto 0)
                                       <= point(d)(6) ;
             when others => point_value(d)(i)(7 downto 0)
                                       <= point(d)(7) ;
          end case ;
        end loop ;
        ----------------
        -- point(d)(0) --
        --------------
        point_value(d)(VERTEX_COUNT-1)
                        (7 downto 0) <= point(d)(0) ;
        ----------------------
        -- Rounding constant --
        ----------------------
        point_value(d)(VERTEX_COUNT)
                        (NUM_INTERP_BITS-1) <= ‘1’ ;
     end loop ;
end process ;

-----------------------------------------------
-- SUM adds the selected vertices values and  --
-- truncates to eight bits. The process is    --
-- implemented to minimize prop delay by      --
-- successive pairing and adding.             --
-----------------------------------------------
SUM: process ( point_value )
     variable  temp:      POINT_VALUES ;
     variable  bound:     INTEGER ;
begin
     -------------------------------
     -- For each output dimension --
     -------------------------------
     for d in 0 to NUM_OUTPUT_DIMEN-1 loop
        -------------------------------
        -- Initialize temp and bound. --
        -------------------------------
        temp := point_value(d) ;
        bound := VERTEX_COUNT + 1 ;
        -------------------------------------
        -- Loop for ceil(log2(VERTEX_COUNT)) --
        -------------------------------------
        for i in 0 to NUM_INTERP_BITS loop
             -------------------------------------
             -- Pair temp array elements and add. --
             -------------------------------------
             for j in 0 to VERTEX_COUNT/2 loop
                  ------------------------------
                  -- Step 1: Move temp(even’s) --
                  ------------------------------
                  if ( 2*j < bound ) then
                       temp(j) := temp(2*j) ;
                  else
                       temp(j) := (others => ‘0’) ;
                  end if ;
                  -----------------------------
                  -- Step 2: Add temp(odd’s) --
                  -----------------------------
                  if ( (2*j)+1 < bound ) then
                      temp(j) := temp(j) + temp((2*j)+1) ;
                  else
                       temp(j) := temp(j) ;
                  end if ;
             end loop ;
             ---------------------------------
             -- Set bound for next iteration --
             ---------------------------------
             bound := ( bound / 2 ) + ( bound mod 2 ) ;
        end loop ;
        ---------------------
        -- Truncate output. --
        ---------------------
        output(d) <= temp(0)(7+NUM_INTERP_BITS
                        downto NUM_INTERP_BITS) ;
     end loop ;
end process ;
-----------------------------------------------
end pruned_tetrtl ;
-----------------------------------------------
```

Figure 18. Pruned Tetrahedral Interpolation VHDL Implementation with n=NUM_INTERP_BITS

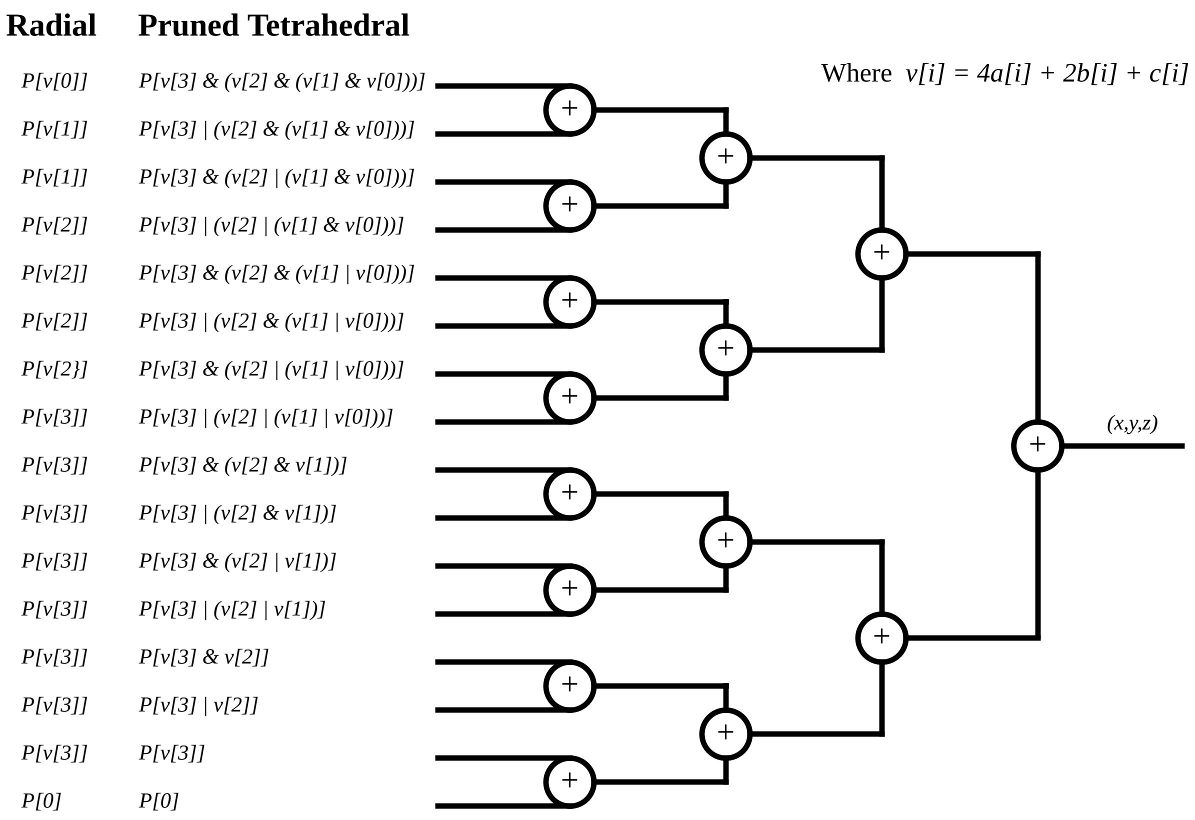

Radial
Pruned Tetrahedral
Where v[i] = 4a[i] + 2b[i] + c[i]
P[v[0]]
P[v[3] & (v[2] & (v[1] & v[0]))]
P[v[1]]
P[v[3] | (v[2] & (v[1] & v[0]))]
P[v[1]]
P[v[3] & (v[2] | (v[1] & v[0]))]
P[v[2]]
P[v[3] | (v[2] | (v[1] & v[0]))]
P[v[2]]
P[v[3] & (v[2] & (v[1] | v[0]))]
P[v[2]]
P[v[3] | (v[2] & (v[1] | v[0]))]
P[v[2}]
P[v[3] & (v[2] | (v[1] | v[0]))]
P[v[3]]
P[v[3] | (v[2] | (v[1] | v[0]))]
P[v[3]]
P[v[3] & (v[2] & v[1])]
P[v[3]]
P[v[3] | (v[2] & v[1])]
P[v[3]]
P[v[3] & (v[2] | v[1])]
P[v[3]]
P[v[3] | (v[2] | v[1])]
P[v[3]]
P[v[3] & v[2]]
P[v[3]]
P[v[3] | v[2]]
P[v[3]]
P[v[3]]
P[0]
P[0]
+
(x,y,z)

Figure 19. Common Radial and Pruned Tetrahedral Implementation

Pruned Tetrahedral implementation would require only 2^n-2 additional d-bit multiplexers to be added to the Pruned Tetrahedral Implementation. Figure 20 presents the Common Radial and Pruned Tetrahedral VHDL implementation. Text in **bold** denote changes from the Pruned Tetrahedral VHDL implementation.

2.4 Non-Symmetric Radial and Non-Symmetric Pruned Tetrahedral Interpolations

In a color space, one dimension may have sudden dramatic changes while others are steady and gradual. In this case it may be desirable to have different resolutions for each dimension in the color lattice. Further, it may be desirable to vary the resolution of each dimension dependent on the region in the color space to optimize the size of the color table.

In order for the interpolator to be able to handle the varying resolutions of the color lattice, the resolution must be specified. Thus, included with the input (a,b,c), the value (n,p,q) is defined such that 2^n points exist between lattice points in the a dimension, 2^p points exist between lattice points in the b dimension, and 2^q points exist between lattice points in the c dimension. The values (n,p,q) may be set for the entire color space or may vary over the color space. Shown in Figure 21 is the cube subdivision process for the case in which $2^n = 2*2^p = 4*2^q$. Note that the shape of the “cube” and number of subcubes vary with the number of valid bits in the slice across the input set.

Presented in Figure 22 is the Radial (Vondran [20]) and Tetrahedral (Vondran [21]) subcube gen-

```
----------------------------------------------------
-- Pruned Tetrahedral Interpolation              --
----------------------------------------------------
entity pruned_tetrahedral is
port (
     -- Interpolation Select --
     tetra_not_radial: in  INPUT_ARRAY ;
     -- Input Pixel --
     input:          in  INPUT_ARRAY ;
     -- Vertex Points of Cube from Color Table --
     point:          in  POINT_ARRAY ;
     -- Output Pixel --
     output:         out OUTPUT_ARRAY
) ;
end pruned_tetrahedral ;

-----------------------------------------------------
architecture pruned_tetrtl of pruned_tetrahedral is
-----------------------------------------------------

 constant VERTEX_COUNT: INTEGER := 2**NUM_INTERP_BITS;
 signal   vertice:      VERTEX_ARRAY ;
 signal   point_value:  POINT_VALUE_ARRAY ;

begin

-----------------------------------------------------
-- Compute vertices.                              --
-----------------------------------------------------
VERTICIES: process ( tetra_not_radial, input )
     variable vertex:   STD_LOGIC_VECTOR(2 downto 0) ;
     variable temp1, temp2:      VERTEX_ARRAY ;
     variable limit, prev_limit:  INTEGER ;
begin
     ---------------------
     -- Initialize Array --
     ---------------------
     limit := 0 ;
     temp2 := ( others => "000" ) ;
     ----------------------------------------------
     -- For each slice across the interpolate bits --
     ----------------------------------------------
     for i in 0 to NUM_INTERP_BITS-1 loop
          -----------------------------------
          -- Load slice into limit location --
          -----------------------------------
          vertex := input(0)(i) & input(1)(i)
                              & input(2)(i) ;
          ---------------------------------
          -- Select Radial or Tetrahedral --
          ---------------------------------
          if ( tetra_not_radial = '0' ) then
               -----------
               -- Radial --
               -----------
               -----------------------
               -- limit is 1, 2, 4, 8 --
               -----------------------
               prev_limit := limit ;
               limit := 2**i ;
               ----------------------------
               -- Assign multiplex selects --
               ----------------------------
               for j in (VERTEX_COUNT/2)-1 downto 0
               loop
                    if ( j < limit ) then
                         temp2(prev_limit + j) := vertex ;
                    else
                         temp2 := temp2 ;
                    end if ;
               end loop ;
          else
               ----------------
               -- Tetrahedral --
               ----------------
               ---------------------------------
               -- limit is 0, 2, 6, 14, 30, ... --
               ---------------------------------
               limit := ( 2**(i+1) ) - 2 ;
               ----------------------------------
               -- load slice into limit location --
               ----------------------------------
               temp2(limit) := vertex ;
               temp1 := temp2 ;
               ---------------------------
               -- for all values in temp1 --
               ---------------------------
               for j in (VERTEX_COUNT/2)-1 downto 0
               loop
                    if ( j < limit/2 ) then
                         temp2(2*j)   := temp1(limit)
                                      and temp1(j) ;
                         temp2((2*j)+1) := temp1(limit)
                                      or  temp1(j) ;
                    else
                         temp2 := temp2 ;
                    end if ;
               end loop ;
          end if ;
     end loop ;
     ---------------------------
     -- Move into index array --
     ---------------------------
     vertice <= temp2 ;
end process ;

-----------------------------------------------------
-- Select the values of vertices to be used by the --
-- interpolation. The last two vertices are always --
-- point(0) and the rounding constant.             --
-----------------------------------------------------
MUX: process ( vertice, point )
begin
     for d in 0 to NUM_OUTPUT_DIMEN-1 loop
        point_value(d) <= (others => (others => '0'));
        for i in 0 to VERTEX_COUNT-2 loop
          case vertice(i) is
              when "000"  => point_value(d)(i)(7 downto 0)
                                           <= point(d)(0) ;
              when "001"  => point_value(d)(i)(7 downto 0)
                                           <= point(d)(1) ;
              when "010"  => point_value(d)(i)(7 downto 0)
                                           <= point(d)(2) ;
              when "011"  => point_value(d)(i)(7 downto 0)
                                           <= point(d)(3) ;
              when "100"  => point_value(d)(i)(7 downto 0)
                                           <= point(d)(4) ;
              when "101"  => point_value(d)(i)(7 downto 0)
                                           <= point(d)(5) ;
              when "110"  => point_value(d)(i)(7 downto 0)
                                           <= point(d)(6) ;
              when others => point_value(d)(i)(7 downto 0)
                                           <= point(d)(7) ;
          end case ;
        end loop ;
        point_value(d)(VERTEX_COUNT-1)
                        (7 downto 0) <= point(d)(0) ;
        point_value(d)(VERTEX_COUNT)
                        (NUM_INTERP_BITS-1) <= '1' ;
     end loop ;
end process ;

------------------------------------------------
-- SUM adds the selected vertices values and  --
-- truncates to eight bits. The process is    --
-- implemented to minimize prop delay by      --
-- successive pairing and adding.             --
------------------------------------------------
SUM: process ( point_value )
     variable  temp:      POINT_VALUES ;
     variable  bound:     INTEGER ;
begin
     for d in 0 to NUM_OUTPUT_DIMEN-1 loop
        temp := point_value(d) ;
        bound := VERTEX_COUNT + 1 ;
        for i in 0 to NUM_INTERP_BITS loop
             for j in 0 to VERTEX_COUNT/2 loop
                  if ( 2*j < bound ) then
                       temp(j) := temp(2*j) ;
                  else
                       temp(j) := (others => '0') ;
                  end if ;
                  if ( (2*j)+1 < bound ) then
                      temp(j) := temp(j) + temp((2*j)+1) ;
                  else
                       temp(j) := temp(j) ;
                  end if ;
             end loop ;
             bound := ( bound / 2 ) + ( bound mod 2 ) ;
        end loop ;
        output(d) <= temp(0)(7+NUM_INTERP_BITS
                        downto NUM_INTERP_BITS) ;
     end loop ;
end process ;

------------------------------------------------
end pruned_tetrtl ;
------------------------------------------------
```

Figure 20. Common Radial and Pruned Tetrahedral VHDL Implementation

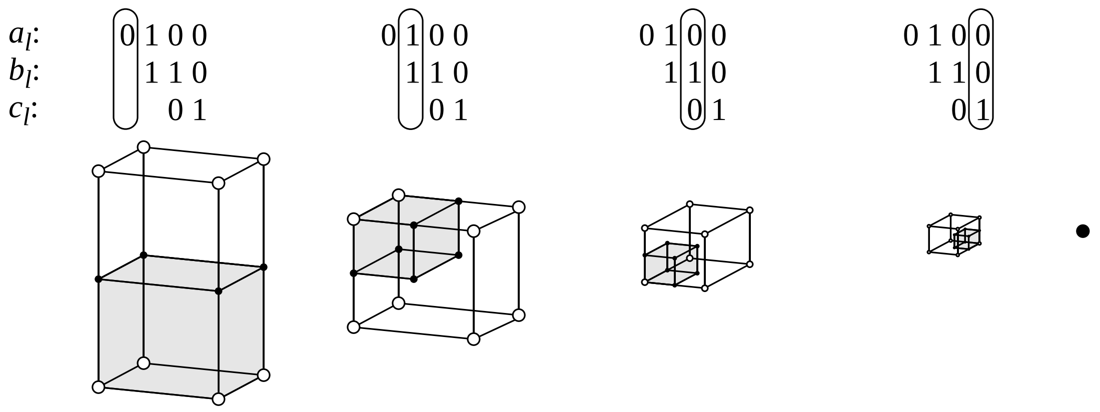
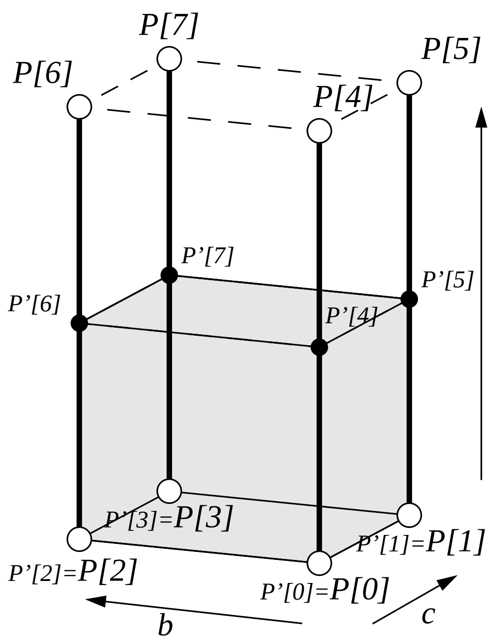


Figure 21. Non-Symmetric Cube Subdivision for (n,p,q) = (4,3,2)

erations given a non-symmetric color lattice. Functions *f(N,i)*, *g(N,i)*, and *h(N,i)* are used to simplify the equations. Notice that the values at the subcube vertices, *P'[]*, are still computed by taking the average of points on the cube being subdivided.



$Mask_a = 2^n\text{-}1, \quad Mask_b = 2^p\text{-}1, \quad Mask_c = 2^q\text{-}1$

$m[i] = 4Mask_a[i] + 2Mask_b[i] + Mask_c[i]$

$v[i] = m[i] \,\&\, (4a[i] + 2b[i] + c[i])$

Radial

$f(N,i) = (v[i] \,\&\, m[i]) \mid (N \,\&\, \sim m[i])$

$= v[i] \mid (N \,\&\, \sim m[i])$

$P'[7] = (P[7] + P[\, f(7,i)\,]) / 2$

$P'[6] = (P[6] + P[\, f(6,i)\,]) / 2$

$P'[5] = (P[5] + P[\, f(5,i)\,]) / 2$

$P'[4] = (P[4] + P[\, f(4,i)\,]) / 2$

$P'[3] = (P[3] + P[\, f(3,i)\,]) / 2$

$P'[2] = (P[2] + P[\, f(2,i)\,]) / 2$

$P'[1] = (P[1] + P[\, f(1,i)\,]) / 2$

$P'[0] = (P[0] + P[\, f(0,i)\,]) / 2$

$= (P[0] + P[\, v[i]\,])$

Tetrahedral

$g(N,i) = (v[i] \mid \sim m[i]) \,\&\, N \qquad h(N,i) = (v[i] \,\&\, m[i]) \mid N = v[i] \mid N$

$P'[7] = (P[\, g(7,i)\,] + P[\, h(7,i)\,]) / 2 \qquad P'[3] = (P[\, g(3,i)\,] + P[\, h(3,i)\,]) / 2$

$P'[6] = (P[\, g(6,i)\,] + P[\, h(6,i)\,]) / 2 \qquad P'[2] = (P[\, g(2,i)\,] + P[\, h(2,i)\,]) / 2$

$P'[5] = (P[\, g(5,i)\,] + P[\, h(5,i)\,]) / 2 \qquad P'[1] = (P[\, g(1,i)\,] + P[\, h(1,i)\,]) / 2$

$P'[4] = (P[\, g(4,i)\,] + P[\, h(4,i)\,]) / 2 \qquad P'[0] = (P[\, g(0,i)\,] + P[\, h(0,i)\,]) / 2$

$= (P[0] + P[v[i]]) / 2$

Figure 22. Non-Symmetric Radial and Tetrahedral Subcube Generation

Figure 23 and 24 present the Non-Symmetric Radial and Pruned Tetrahedral Interpolations. Note, a common implementation (Vondran [18]) can be used for both by multiplexing the indices of the input values.

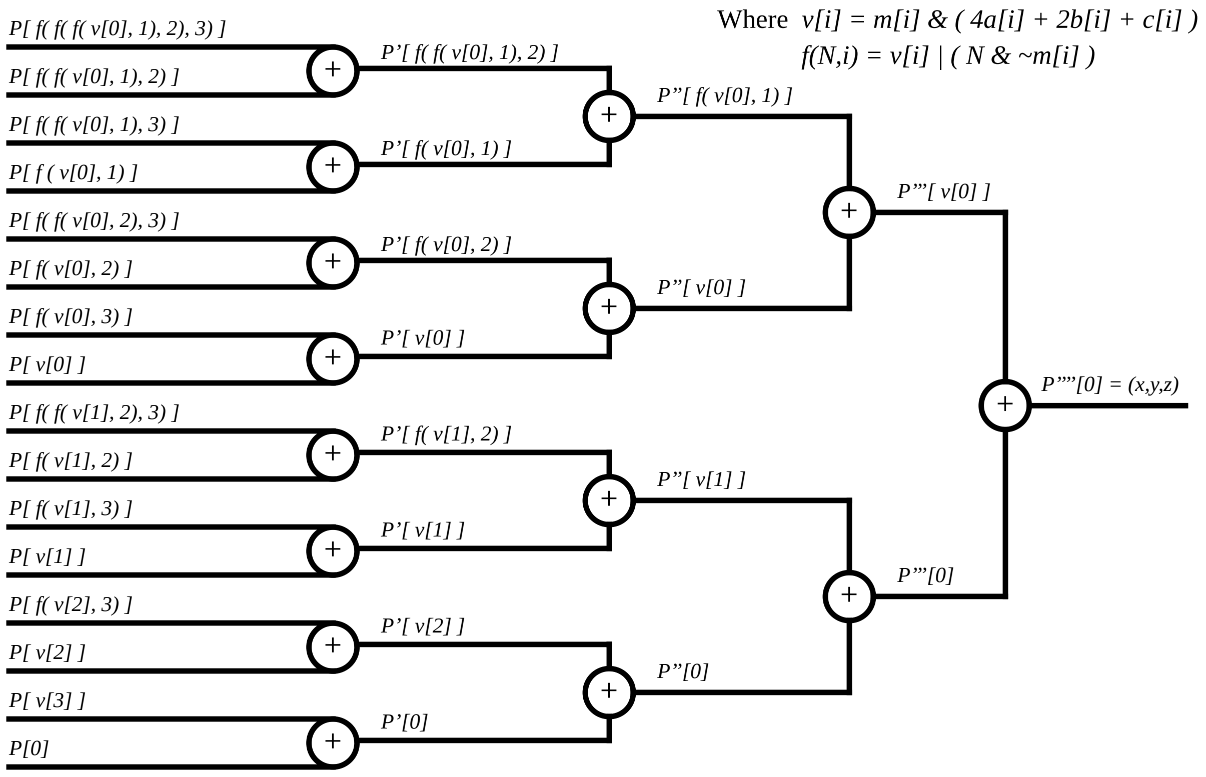


Figure 23. Non-Symmetric Radial Interpolation

Given d input dimensions, D output dimensions and $2^{(n,p,q)}$ values between coarse lattice points, the number of computations for Non-Symmetric Radial Interpolation is: Each *v[i]* and *m[i]* require d adds, and d-1 shifts. One additional bitwise AND operation masks *v[i]* using *m[i]*. There are max(n,p,q) vertices *v[i]* and max(n,p,q) masks *m[i]* generated, resulting in max(n,p,q)*((2d-1)+(2d-1)+1) = max(n,p,q)*(4d-1) operations. Each *f(N,i)* requires 3 bitwise logic operations (one bitwise OR, one AND, and one NOT). It can be shown that the number of *f(N,i)* references is

$$\text{number } f(N, i) \text{ references } = 2^{max(n,\,p,\,q)-1} \sum_{i=1}^{max(n,\,p,\,q)-1} (1-2^{-i})$$

As shown in Figure 23, $2^{\max(n,p,q)}$-1 additions are used. One additional add and shift is needed per output component to divide and round, totaling ($2^{\max(n,p,q)}$-1)+1+1 = $2^{\max(n,p,q)}$+1. In Figure 23, $2^{\max(n,p,q)}$ memory references are shown, however, a cap can be implemented on the number of memory accesses at the number of vertices of the cube (2^d vertices are on the cube). Thus, the minimum of the two is used, i.e. min(2^d, $2^{\max(n,p,q)}$). With D output components the total number

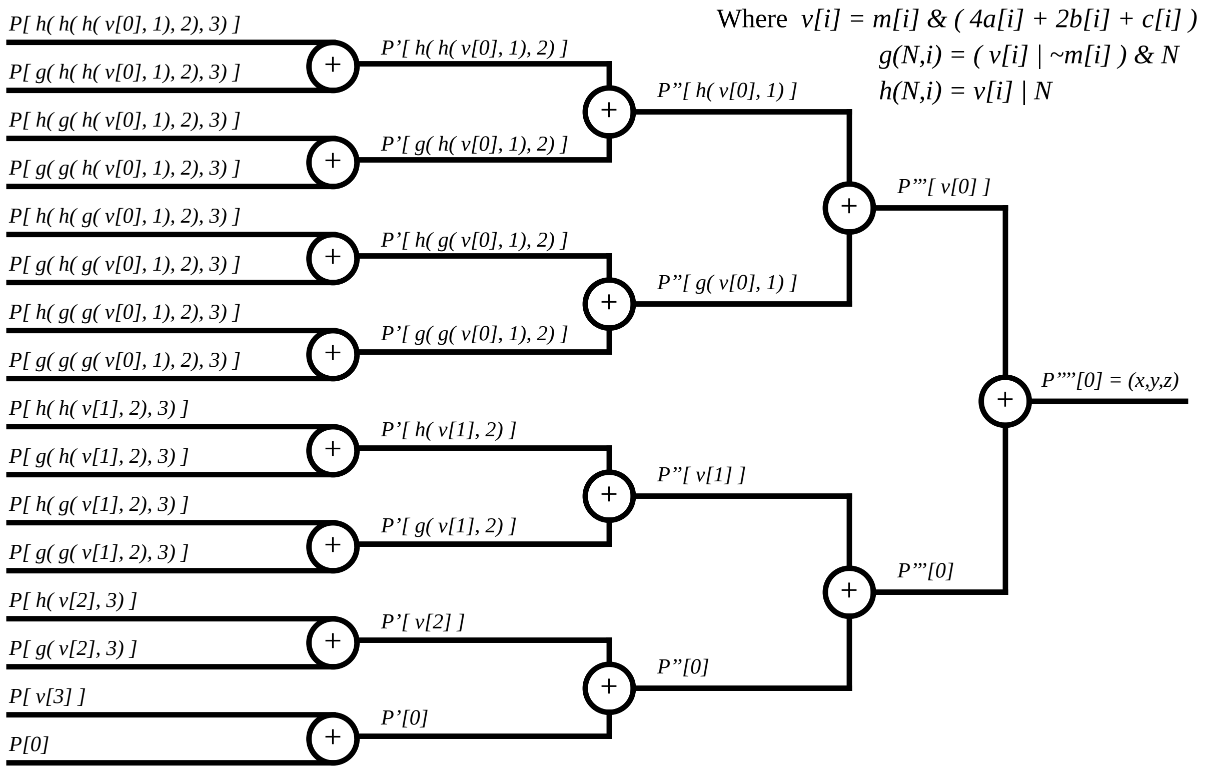


Figure 24. Non-Symmetric Pruned Tetrahedral Interpolation

of computations for the Non-Symmetric Radial Interpolation is (assuming n=max(n,p,q))

$$n(4d-1)+3\left(2^{n-1}\sum_{i=1}^{n-1}(1-2^{-i})\right)+D(2^n+1)\Rightarrow O(nd+2^nD) \qquad \text{ALU Operations}$$

$$\text{and} \qquad D(min(2^d,2^n))\Rightarrow O(D(min(2^d,2^n))) \qquad \text{Memory Accesses}$$

Using the Non-Symmetric Radial Interpolation to convert a 24 bit RGB pixel to a 32 bit CMYK pixel with n=p=q=4 requires 163 ALU operations and 32 memory accesses

The hardware complexity for Non-Symmetric Radial Interpolation is as follows: The number of AND/OR gates to compute the indices is equal to the number of *f(N,i)* references times *d*. To compute each output component requires the selection of $2^{\max(n,p,q)}$-1 values followed by $2^{\max(n,p,q)}$-1 additions. Rounding requires one additional add and a truncation. The truncation is a right shift, which is free. With D output components the total number of hardware elements is (assuming n=max(n,p,q))

$$D((2^n-1)+1) = D(2^n)\Rightarrow O(2^nD) \qquad \text{Adders}$$

$$D(2^n-1)\Rightarrow O(2^nD) \qquad \text{Multiplexers } (2^d{:}1)$$

$$3d\left(2^{n-1}\sum_{i=1}^{n-1}(1-2^{-i})\right)\Rightarrow O(2^nd) \qquad \text{Additional AND/OR Gates}$$

Figure 25 presents the Non-Symmetric Radial Interpolation C implementation. In this case, the input consists of arrays of input components, cube vertices (points), and lattice resolution values. The output is an array of components corresponding to the color in the output space. A cube vertex point array is passed in rather than a table pointer because the color lattice resolution for each dimension may vary depending upon the region in the color space. How the color lattice is packed in the color table will determine the hashing scheme needed. Thus the presented implementation assumes the resolution and address into the color table have been computed prior to accessing the interpolation routine. The remaining portion of Figure 24 follows closely the Non-Symmetric Radial Interpolation method previously derived.

```c
#define f(v, m, N)  ( v | ( N & ~m ) )
/*************************************************/
/** nonsym_radial_interpolation()               **/
/*************************************************/
void nonsym_radial_interpolation( unsigned char input[],
 unsigned char point[][8], unsigned char resolution[],
 unsigned char output[] )
{
 register unsigned char Amask, Bmask, Cmask ;
 register          int  a, b, c, x, y, z ;
 register unsigned char m0, m1, m2, m3 ;
 register unsigned char v0, v1, v2, v3 ;
 register unsigned char f_v1_v0, f_v2_v1 ;
 register unsigned char f_v2_f_v1_v0, f_v2_v0 ;
 register unsigned char f_v3_v2, f_v3_f_v2_v1 ;
 register unsigned char f_v3_f_v2_f_v1_v0 ;
 register unsigned char f_v3_f_v2_v0, f_v3_v1 ;
 resister unsigned char f_v3_f_v1_v0, f_v3_v0 ;

   /******************/
   /* Generate Masks */
   /******************/
   Amask = ( 1 << resolution[0] ) - 1 ;
   Bmask = ( 1 << resolution[1] ) - 1 ;
   Cmask = ( 1 << resolution[2] ) - 1 ;

   /***************************************/
   /* Snap last value to last table value */
   /***************************************/
   a = ( input[0] == 0x0ff ) ? 0x100 : (int) input[0] ;
   b = ( input[1] == 0x0ff ) ? 0x100 : (int) input[1] ;
   c = ( input[2] == 0x0ff ) ? 0x100 : (int) input[2] ;

   /*************************************************/
   /* Compute slices across masks and input components */
   /*************************************************/
   m0 = ( (Amask & 0x01) << 2 )
      + ( (Bmask & 0x01) << 1 ) + (Cmask & 0x01);
   m1 = ( (Amask & 0x02) << 1 )
      +   (Bmask & 0x02) + ( (Cmask & 0x02) >> 1 );
   m2 =   (Amask & 0x04)
       + ( (Bmask & 0x04) >> 1 ) + ( (Cmask & 0x04) >> 2 );
   m3 = ( (Amask & 0x08) >> 1 )
       + ( (Bmask & 0x08) >> 2 ) + ( (Cmask & 0x08) >> 3 );
   v0 = m0 & ( ( (a & 0x01) << 2 )
      + ( (b & 0x01) << 1 ) +   (c & 0x01) );
   v1 = m1 & ( ( (a & 0x02) << 1 )
      +   (b & 0x02) +        ( (c & 0x02) >> 1 ) ) ;
   v2 = m2 & (   (a & 0x04)
      + ( (b & 0x04) >> 1 ) + ( (c & 0x04) >> 2 ) ) ;
   v3 = m3 & ( ( (a & 0x08) >> 1 )
      + ( (b & 0x08) >> 2 ) + ( (c & 0x08) >> 3 ) ) ;

   /*******************************/
   /* Compute offset from origin. */
   /*******************************/
   f_v1_v0                = f(v1,m1,v0) ;
   f_v2_v1                = f(v2,m2,v1) ;
   f_v2_f_v1_v0           = f(v2,m2,f_v1_v0) ;
   f_v2_v0                = f(v2,m2,v0) ;
   f_v3_v2                = f(v3,m3,v2) ;
   f_v3_f_v2_v1           = f(v3,m3,f_v2_v1) ;
   f_v3_f_v2_f_v1_v0      = f(v3,m3,f_v2_f_v1_v0) ;
   f_v3_f_v2_v0           = f(v3,m3,f_v2_v0) ;
   f_v3_v1                = f(v3,m3,v1) ;
   f_v3_f_v1_v0           = f(v3,m3,f_v1_v0) ;
   f_v3_v0                = f(v3,m3,v0) ;
```

```c
   /*************/
   /* Compute x */
   /*************/
   x = ( ( point[0][f_v3_f_v2_f_v1_v0]
         + point[0][f_v2_f_v1_v0]
         + point[0][f_v3_f_v1_v0]
         + point[0][f_v1_v0]
         + point[0][f_v3_f_v2_v0]
         + point[0][f_v2_v0]
         + point[0][f_v3_v0]
         + point[0][v0]
         + point[0][f_v3_f_v2_v1]
         + point[0][f_v2_v1]
         + point[0][f_v3_v1]
         + point[0][v1]
         + point[0][f_v3_v2]
         + point[0][v2]
         + point[0][v3]
         + point[0][0]
         + 0x08 ) >> 4 ) ;                /*Round*/
   /*************/
   /* Compute y */
   /*************/
   y = ( ( point[1][f_v3_f_v2_f_v1_v0]
         + point[1][f_v2_f_v1_v0]
         + point[1][f_v3_f_v1_v0]
         + point[1][f_v1_v0]
         + point[1][f_v3_f_v2_v0]
         + point[1][f_v2_v0]
         + point[1][f_v3_v0]
         + point[1][v0]
         + point[1][f_v3_f_v2_v1]
         + point[1][f_v2_v1]
         + point[1][f_v3_v1]
         + point[1][v1]
         + point[1][f_v3_v2]
         + point[1][v2]
         + point[1][v3]
         + point[1][0]
         + 0x08 ) >> 4 ) ;                /*Round*/
   /*************/
   /* Compute z */
   /*************/
   z = ( ( point[2][f_v3_f_v2_f_v1_v0]
         + point[2][f_v2_f_v1_v0]
         + point[2][f_v3_f_v1_v0]
         + point[2][f_v1_v0]
         + point[2][f_v3_f_v2_v0]
         + point[2][f_v2_v0]
         + point[2][f_v3_v0]
         + point[2][v0]
         + point[2][f_v3_f_v2_v1]
         + point[2][f_v2_v1]
         + point[2][f_v3_v1]
         + point[2][v1]
         + point[2][f_v3_v2]
         + point[2][v2]
         + point[2][v3]
         + point[2][0]
         + 0x08 ) >> 4 ) ;                /*Round*/
   /**********************/
   /* Write back results */
   /**********************/
   output[0] = (char) x ;
   output[1] = (char) y ;
   output[2] = (char) z ;
}
```

Figure 25. Non-Symmetric Radial Interpolation C Implementation

Figure 26 presents the VHDL implementation of the Non-Symmetric Radial Interpolation. Note, the implementation closely follows that of the Pruned Tetrahedral Interpolation in Figure 18. The differences consist of the addition of the MASKS process and the lines “P’[2*j] = P[j] ;” and “P’[2*i+1]=v[i]|(P[i] & ~m[i]) ;” in the VERTICES process. No changes are made in the MUX and SUM processes (with the exception of several comment statements)

The number of computations for the Non-Symmetric Pruned Tetrahedral Interpolation is nearly identical to the Non-Symmetric Radial Interpolation. In the Pruned Tetrahedral case each *g(i,N)* requires 3 bit-wise logic operations, while *h(i,N)* requires only 1. It can be shown that the number of *g(i,N)* and *h(i,N)* references are

$$\text{number } g(N,i) \text{ references} = \text{number } h(N,i) \text{ references} = 2^{max(n,p,q)-1}\sum_{i=1}^{max(n,p,q)-1}(1-2^{-i})$$

Thus the total number of computations for the Non-Symmetric Pruned Tetrahedral Interpolation is (assuming n=max(n,p,q))

$$n(4d-1)+(3+1)\left(2^{n-1}\sum_{i=1}^{n-1}(1-2^{-i})\right)+D(2^n+1)\Rightarrow O(nd+2^nD) \qquad \text{ALU Operations}$$

$$\text{and} \qquad D(min(2^d,2^n))\Rightarrow O(D(min(2^d,2^n))) \qquad \text{Memory Accesses}$$

Using the Non-Symmetric Pruned Tetrahedral Interpolation to convert a 24 bit RGB pixel to a 32 bit CMYK pixel with n=4 requires 180 ALU and 32 memory operations.

The hardware complexity for Non-Symmetric Pruned Tetrahedral is as follows: The number of AND/OR gates to compute the indices is equal to the number of *g(N,i)* and *h(N,i)* references times *d*. To compute each output component requires the selection of $2^{\max(n,p,q)}$-1 values followed by $2^{\max(n,p,q)}$-1 additions. Rounding requires one additional add and a truncation. With D output components the total number of hardware elements is (assuming n=max(n,p,q))

$$D((2^n-1)+1) = D(2^n)\Rightarrow O(2^nD) \qquad \text{Adders}$$

$$D(2^n-1)\Rightarrow O(2^nD) \qquad \text{Multiplexers } (2^d{:}1)$$

$$4d\left(2^{n-1}\sum_{i=1}^{n-1}(1-2^{-i})\right)\Rightarrow O(2^nd) \qquad \text{Additional AND/OR Gates}$$

Figures 27 and 28 present the Non-Symmetric Pruned Tetrahedral C and VHDL implementation, respectively. These implementations are nearly identical to the Non-Symmetric Radial implementations (Figure 25 and 26) with the exception of the computations of the offsets in the C implementation and the computations of the vertex points in the VERTICES process in the VHDL implementation.

Figure 29 presents the Common Non-Symmetric Radial and Pruned Tetrahedral VHDL implementation. Note, text in **bold** denote the portion that selects Radial or Pruned Tetrahedral. All other portions are common.

```
----------------------------------------
-- Non-Symmetric Radial Interpolation --
----------------------------------------
entity nonsym_radial is
port (
     -- Input Pixel --
     input:         in    INPUT_ARRAY ;
     -- Vertex Points of Cube from Color Table --
     point:         in    POINT_ARRAY ;
     -- Lattice Resolution --
     resolution:    in    RESOLUTION_ARRAY ;
     -- Output Pixel --
     output:        out   OUTPUT_ARRAY
) ;
end nonsym_radial ;

-----------------------------------------------------
architecture nonsym_radialrtl of nonsym_radial is
---------------------- ------------------------------

 constant VERTEX_COUNT: INTEGER := 2**NUM_INTERP_BITS;
 signal   mask:         MASK_ARRAY ;
 signal   vertex:       VERTEX_ARRAY ;
 signal   point_value:  POINT_VALUE_ARRAY ;

begin

-----------------------------------------------------
-- Compute Lattice Resolution Masks               --
-----------------------------------------------------
MASKS: process ( resolution )
     variable  top:           INTEGER ;
begin
     ------------------------
     -- For each Component --
     ------------------------
     for i in 0 to 2 loop
       top := 0 ;
       -------------------------------------
       -- Convert BIT_VECTOR to INTEGER --
       -------------------------------------
       for j in 2 downto 0 loop
            if ( resolution(i)(j) = ‘1’ ) then
                 top := top + 2**j ;
            else
                 top := top ;
            end if ;
       end loop ;
       -------------------
       -- Generate Mask --
       -------------------
       for j in NUM_INTERP_BITS-1 downto 0 loop
            if ( j < top ) then
                 mask(j)(i) <= ‘1’ ;
            else
                 mask(j)(i) <= ‘0’ ;
            end if ;
       end loop ;
     end loop ;
end process ;

-----------------------------------------------------
-- Compute vertices.                              --
-----------------------------------------------------
VERTICIES: process ( input, mask )
     variable  temp1, temp2:   VERTEX_ARRAY ;
     variable  limit:          INTEGER ;
begin
     -----------------------
     -- Initialize Arrays --
     -----------------------
     temp2 := ( others => “000” ) ;
     ------------------------------------------------
     -- For each slice across the interpolate bits --
     ------------------------------------------------
     for i in 0 to NUM_INTERP_BITS-1 loop
          ----------------------------------
          -- limit is 0, 2, 6, 14, 30, ... --
          ----------------------------------
          limit := ( 2**(i+1) ) - 2 ;
          -----------------------------------
          -- Load slice into limit location --
          -----------------------------------
          temp2(limit) := mask(i) and ( input(0)(i)
                      & input(1)(i) & input(2)(i) ) ;
          temp1 := temp2 ;
          ----------------------------
          -- for all values in temp1 --
          ----------------------------
          for j in (VERTEX_COUNT/2)-1 downto 0 loop
               if ( j < limit/2 ) then
                    --------------------
                    -- P’[2*j] = P[j] --
                    --------------------
                    temp2(2*j)     := temp1(j) ;
                    ----------------------------------
                    -- P’[2*j+1]=v[i]|(P[j] & ~m[i])--
                    ----------------------------------
                    temp2((2*j)+1) := temp1(limit)
                         or ( temp1(j) and not mask[i] ) ;
               else
                    temp2 := temp2 ;
               end if ;
          end loop ;
     end loop ;
     ---------------------------
     -- Move into index array --
     ---------------------------
     vertex <= temp2 ;
end process ;

-----------------------------------------------------
-- Select the values of vertices to be used by the --
-- interpolation. The last two vertices are always --
-- point(0) and the rounding constant.             --
-----------------------------------------------------
MUX: process ( vertex, point )
begin
     for d in 0 to NUM_OUTPUT_DIMEN-1 loop
        point_value(d) <= (others => (others => ‘0’));
        for i in 0 to VERTEX_COUNT-2 loop
          case vertex(i) is
              when “000”  => point_value(d)(i)(7 downto 0)
                                      <= point(d)(0) ;
              when “001”  => point_value(d)(i)(7 downto 0)
                                      <= point(d)(1) ;
              when “010”  => point_value(d)(i)(7 downto 0)
                                      <= point(d)(2) ;
              when “011”  => point_value(d)(i)(7 downto 0)
                                      <= point(d)(3) ;
              when “100”  => point_value(d)(i)(7 downto 0)
                                      <= point(d)(4) ;
              when “101”  => point_value(d)(i)(7 downto 0)
                                      <= point(d)(5) ;
              when “110”  => point_value(d)(i)(7 downto 0)
                                      <= point(d)(6) ;
              when others => point_value(d)(i)(7 downto 0)
                                      <= point(d)(7) ;
          end case ;
        end loop ;
        point_value(d)(VERTEX_COUNT-1)
                        (7 downto 0) <= point(d)(0) ;
        point_value(d)(VERTEX_COUNT)
                        (NUM_INTERP_BITS-1) <= ‘1’ ;
     end loop ;
end process ;

-----------------------------------------------------
-- Select the values of vertices to be used by the --
-- interpolation. The last two vertices are always --
-- point(0) and the rounding constant.             --
-----------------------------------------------------
SUM: process ( point_value )
     variable  temp:      POINT_VALUES ;
     variable  bound:     INTEGER ;
begin
     for d in 0 to NUM_OUTPUT_DIMEN-1 loop
        temp := point_value(d) ;
        bound := VERTEX_COUNT + 1 ;
        for i in 0 to NUM_INTERP_BITS loop
             for j in 0 to VERTEX_COUNT/2 loop
                  if ( 2*j < bound ) then
                       temp(j) := temp(2*j) ;
                  else
                       temp(j) := (others => ‘0’) ;
                  end if ;
                  if ( (2*j)+1 < bound ) then
                      temp(j) := temp(j) + temp((2*j)+1) ;
                  else
                       temp(j) := temp(j) ;
                  end if ;
             end loop ;
             bound := ( bound / 2 ) + ( bound mod 2 ) ;
        end loop ;
        output(d) <= temp(0)(7+NUM_INTERP_BITS
                        downto NUM_INTERP_BITS) ;
     end loop ;
end process ;

------------------------------------------------
end nonsym_radialrtl ;
------------------------------------------------
```

Figure 26. Non-Symmetric Radial Interpolation VHDL Implementation

```c
#define g(v, m, N)  ( ( v | ~m ) & N )
#define h(v, N)       ( v | N )
/*************************************************/
/** nonsym_pruned_tetrahedral()                 **/
/*************************************************/
void nonsym_pruned_tetrahedral( unsigned char input[],
   unsigned char point[][8], unsigned char resolution[],
   unsigned char output[] )
{
   register unsigned char Amask, Bmask, Cmask ;
   register          int  a, b, c, x, y, z ;
   register unsigned char m0, m1, m2, m3 ;
   register unsigned char v0, v1, v2, v3 ;
   register unsigned char g_v1_v0, h_v1_v0 ;
   register unsigned char g_v2_v1, g_v2_g_v1_v0 ;
   register unsigned char g_v2_h_v1_v0, h_v2_v1 ;
   register unsigned char h_v2_g_v1_v0, h_v2_h_v1_v0 ;
   register unsigned char g_v3_v2, g_v3_g_v2_v1 ;
   register unsigned char g_v3_h_v2_v1 ;
   register unsigned char g_v3_g_v2_g_v1_v0 ;
   register unsigned char g_v3_g_v2_h_v1_v0 ;
   register unsigned char g_v3_h_v2_g_v1_v0 ;
   register unsigned char g_v3_h_v2_h_v1_v0 ;
   register unsigned char h_v3_v2, h_v3_g_v2_v1 ;
   register unsigned char h_v3_h_v2_v1 ;
   register unsigned char h_v3_g_v2_g_v1_v0 ;
   register unsigned char h_v3_g_v2_h_v1_v0 ;
   register unsigned char h_v3_h_v2_g_v1_v0 ;
   register unsigned char h_v3_h_v2_h_v1_v0 ;

   /******************/
   /* Generate Masks */
   /******************/
   Amask = ( 1 << resolution[0] ) - 1 ;
   Bmask = ( 1 << resolution[1] ) - 1 ;
   Cmask = ( 1 << resolution[2] ) - 1 ;

   /***************************************/
   /* Snap last value to last table value */
   /***************************************/
   a = ( input[0] == 0x0ff ) ? 0x100 : (int) input[0] ;
   b = ( input[1] == 0x0ff ) ? 0x100 : (int) input[1] ;
   c = ( input[2] == 0x0ff ) ? 0x100 : (int) input[2] ;

   /*************************************************/
   /* Compute slices across masks and input components */
   /*************************************************/
   m0 = ( (Amask & 0x01) << 2 )
      + ( (Bmask & 0x01) << 1 ) + (Cmask & 0x01);
   m1 = ( (Amask & 0x02) << 1 )
      +   (Bmask & 0x02) + ( (Cmask & 0x02) >> 1 );
   m2 =   (Amask & 0x04)
       + ( (Bmask & 0x04) >> 1 ) + ( (Cmask & 0x04) >> 2 );
   m3 = ( (Amask & 0x08) >> 1 )
       + ( (Bmask & 0x08) >> 2 ) + ( (Cmask & 0x08) >> 3 );
   v0 = m0 & ( ( (a & 0x01) << 2 )
      + ( (b & 0x01) << 1 ) +   (c & 0x01) ) ;
   v1 = m1 & ( ( (a & 0x02) << 1 )
      +   (b & 0x02) +        ( (c & 0x02) >> 1 ) ) ;
   v2 = m2 & (   (a & 0x04)
      + ( (b & 0x04) >> 1 ) + ( (c & 0x04) >> 2 ) ) ;
   v3 = m3 & ( ( (a & 0x08) >> 1 )
      + ( (b & 0x08) >> 2 ) + ( (c & 0x08) >> 3 ) ) ;

   /*******************************/
   /* Compute offset from origin. */
   /*******************************/
   g_v1_v0                = g(v1,m1,v0) ;
   h_v1_v0                = h(v1,v0) ;
   g_v2_v1                = g(v2,m2,v1) ;
   g_v2_g_v1_v0           = g(v2,m2,g_v1_v0) ;
   g_v2_h_v1_v0           = g(v2,m2,h_v1_v0) ;
   h_v2_v1                = h(v2,v1) ;
   h_v2_g_v1_v0           = h(v2,g_v1_v0) ;
   h_v2_h_v1_v0           = h(v2,h_v1_v0) ;
   g_v3_v2                = g(v3,m3,v2) ;
   g_v3_g_v2_v1           = g(v3,m3,g_v2_v1) ;
   g_v3_g_v2_g_v1_v0      = g(v3,m3,g_v2_g_v1_v0) ;
   g_v3_g_v2_h_v1_v0      = g(v3,m3,g_v2_h_v1_v0) ;
   g_v3_h_v2_v1           = g(v3,m3,h_v2_v1) ;
   g_v3_h_v2_g_v1_v0      = g(v3,m3,h_v2_g_v1_v0) ;
   g_v3_h_v2_h_v1_v0      = g(v3,m3,h_v2_h_v1_v0) ;
```

```c
   h_v3_v2                = h(v3,v2) ;
   h_v3_g_v2_v1           = h(v3,g_v2_v1) ;
   h_v3_g_v2_g_v1_v0      = h(v3,g_v2_g_v1_v0) ;
   h_v3_g_v2_h_v1_v0      = h(v3,g_v2_h_v1_v0) ;
   h_v3_h_v2_v1           = h(v3,h_v2_v1) ;
   h_v3_h_v2_g_v1_v0      = h(v3,h_v2_g_v1_v0) ;
   h_v3_h_v2_h_v1_v0      = h(v3,h_v2_h_v1_v0) ;

   /*************/
   /* Compute x */
   /*************/
   x = ( ( point[0][0]
         + point[0][v3]
         + point[0][g_v3_v2]
         + point[0][g_v3_g_v2_v1]
         + point[0][g_v3_g_v2_g_v1_v0]
         + point[0][g_v3_g_v2_h_v1_v0]
         + point[0][g_v3_h_v2_v1]
         + point[0][g_v3_h_v2_g_v1_v0]
         + point[0][g_v3_h_v2_h_v1_v0]
         + point[0][h_v3_v2]
         + point[0][h_v3_g_v2_v1]
         + point[0][h_v3_g_v2_g_v1_v0]
         + point[0][h_v3_g_v2_h_v1_v0]
         + point[0][h_v3_h_v2_v1]
         + point[0][h_v3_h_v2_g_v1_v0]
         + point[0][h_v3_h_v2_h_v1_v0]
         + 0x08 ) >> 4 ) ;                /*Round*/

   /*************/
   /* Compute y */
   /*************/
   y = ( ( point[1][0]
         + point[1][v3]
         + point[1][g_v3_v2]
         + point[1][g_v3_g_v2_v1]
         + point[1][g_v3_g_v2_g_v1_v0]
         + point[1][g_v3_g_v2_h_v1_v0]
         + point[1][g_v3_h_v2_v1]
         + point[1][g_v3_h_v2_g_v1_v0]
         + point[1][g_v3_h_v2_h_v1_v0]
         + point[1][h_v3_v2]
         + point[1][h_v3_g_v2_v1]
         + point[1][h_v3_g_v2_g_v1_v0]
         + point[1][h_v3_g_v2_h_v1_v0]
         + point[1][h_v3_h_v2_v1]
         + point[1][h_v3_h_v2_g_v1_v0]
         + point[1][h_v3_h_v2_h_v1_v0]
         + 0x08 ) >> 4 ) ;                /*Round*/

   /*************/
   /* Compute z */
   /*************/
   z = ( ( point[2][0]
         + point[2][v3]
         + point[2][g_v3_v2]
         + point[2][g_v3_g_v2_v1]
         + point[2][g_v3_g_v2_g_v1_v0]
         + point[2][g_v3_g_v2_h_v1_v0]
         + point[2][g_v3_h_v2_v1]
         + point[2][g_v3_h_v2_g_v1_v0]
         + point[2][g_v3_h_v2_h_v1_v0]
         + point[2][h_v3_v2]
         + point[2][h_v3_g_v2_v1]
         + point[2][h_v3_g_v2_g_v1_v0]
         + point[2][h_v3_g_v2_h_v1_v0]
         + point[2][h_v3_h_v2_v1]
         + point[2][h_v3_h_v2_g_v1_v0]
         + point[2][h_v3_h_v2_h_v1_v0]
         + 0x08 ) >> 4 ) ;                /*Round*/

   /**********************/
   /* Write back results */
   /**********************/
   output[0] = (char) x ;
   output[1] = (char) y ;
   output[2] = (char) z ;

}
```

Figure 27. Non-Symmetric Pruned Tetrahedral Interpolation C Implementation

```
----------------------------------------------------
-- Non-Symmetric Pruned Tetrahedral Interpolation --
----------------------------------------------------
entity nonsym_pruned_tetrahedral is
port (
     -- Input Pixel --
     input:         in    INPUT_ARRAY ;
     -- Vertex Points of Cube from Color Table --
     point:         in    POINT_ARRAY ;
     -- Lattice Resolution --
     resolution:    in    RESOLUTION_ARRAY ;
     -- Output Pixel --
     output:        out   OUTPUT_ARRAY
) ;
end nonsym_pruned_tetrahedral ;

-----------------------------------------------------
architecture nonsym_tetrtl of nonsym_pruned_tetrahedral is
--------------------- -------------------------------

 constant VERTEX_COUNT: INTEGER := 2**NUM_INTERP_BITS;
 signal   mask:         MASK_ARRAY ;
 signal   vertex:       VERTEX_ARRAY ;
 signal   point_value:  POINT_VALUE_ARRAY ;

begin

-----------------------------------------------------
-- Compute Lattice Resolution Masks               --
-----------------------------------------------------
MASKS: process ( resolution )
     variable  top:           INTEGER ;
begin
     -----------------------
     -- For each Component --
     -----------------------
     for i in 0 to 2 loop
       top := 0 ;
       ------------------------------------
       -- Convert BIT_VECTOR to INTEGER --
       ------------------------------------
       for j in 2 downto 0 loop
            if ( resolution(i)(j) = ‘1’ ) then
                 top := top + 2**j ;
            else
                 top := top ;
            end if ;
       end loop ;
       -------------------
       -- Generate Mask --
       -------------------
       for j in NUM_INTERP_BITS-1 downto 0 loop
            if ( j < top ) then
                 mask(j)(i) <= ‘1’ ;
            else
                 mask(j)(i) <= ‘0’ ;
            end if ;
       end loop ;
     end loop ;
end process ;

-----------------------------------------------------
-- Compute vertices.                              --
-----------------------------------------------------
VERTICIES: process ( input, mask )
     variable  temp1, temp2:   VERTEX_ARRAY ;
     variable  limit:          INTEGER ;
begin
     ----------------------
     -- Initialize Arrays --
     ----------------------
     temp2 := ( others => “000” ) ;
     ----------------------------------------------
     -- For each slice across the interpolate bits --
     ----------------------------------------------
     for i in 0 to NUM_INTERP_BITS-1 loop
          ----------------------------------
          -- limit is 0, 2, 6, 14, 30, ... --
          ----------------------------------
          limit := ( 2**(i+1) ) - 2 ;
          ------------------------------------
          -- Load slice into limit location --
          ------------------------------------
          temp2(limit) := mask(i) and ( input(0)(i)
                      & input(1)(i) & input(2)(i) ) ;
          temp1 := temp2 ;
          -----------------------------
          -- for all values in temp1 --
          -----------------------------
          for j in (VERTEX_COUNT/2)-1 downto 0 loop
               if ( j < limit/2 ) then
                    ------------------------------
                    -- P’[2*j]=(v[i] | ~m[i]) & P[j]--
                    ------------------------------
                    temp2(2*j)    := ( temp1(limit)
                           or not mask(i) ) and temp1(j) ;
                    ----------------------------
                    -- P’[2*j+1] = v[i] | P[j] --
                    ----------------------------
                    temp2((2*j)+1) := temp1(limit)
                                          or temp1(j) ;
               else
                    temp2 := temp2 ;
               end if ;
          end loop ;
     end loop ;
     --------------------------
     -- Move into index array --
     --------------------------
     vertex <= temp2 ;
end process ;

-----------------------------------------------------
-- Select the values of vertices to be used by the --
-- interpolation. The last two vertices are always --
-- point(0) and the rounding constant.             --
-----------------------------------------------------
MUX: process ( vertex, point )
begin
     for d in 0 to NUM_OUTPUT_DIMEN-1 loop
        point_value(d) <= (others => (others => ‘0’));
        for i in 0 to VERTEX_COUNT-2 loop
          case vertex(i) is
              when “000”  => point_value(d)(i)(7 downto 0)
                                      <= point(d)(0) ;
              when “001”  => point_value(d)(i)(7 downto 0)
                                      <= point(d)(1) ;
              when “010”  => point_value(d)(i)(7 downto 0)
                                      <= point(d)(2) ;
              when “011”  => point_value(d)(i)(7 downto 0)
                                      <= point(d)(3) ;
              when “100”  => point_value(d)(i)(7 downto 0)
                                      <= point(d)(4) ;
              when “101”  => point_value(d)(i)(7 downto 0)
                                      <= point(d)(5) ;
              when “110”  => point_value(d)(i)(7 downto 0)
                                      <= point(d)(6) ;
              when others => point_value(d)(i)(7 downto 0)
                                      <= point(d)(7) ;
          end case ;
        end loop ;
        point_value(d)(VERTEX_COUNT-1)
                        (7 downto 0) <= point(d)(0) ;
        point_value(d)(VERTEX_COUNT)
                        (NUM_INTERP_BITS-1) <= ‘1’ ;
     end loop ;
end process ;

-----------------------------------------------------
-- Select the values of vertices to be used by the --
-- interpolation. The last two vertices are always --
-- point(0) and the rounding constant.             --
-----------------------------------------------------
SUM: process ( point_value )
     variable  temp:      POINT_VALUES ;
     variable  bound:     INTEGER ;
begin
     for d in 0 to NUM_OUTPUT_DIMEN-1 loop
        temp := point_value(d) ;
        bound := VERTEX_COUNT + 1 ;
        for i in 0 to NUM_INTERP_BITS loop
             for j in 0 to VERTEX_COUNT/2 loop
                  if ( 2*j < bound ) then
                       temp(j) := temp(2*j) ;
                  else
                       temp(j) := (others => ‘0’) ;
                  end if ;
                  if ( (2*j)+1 < bound ) then
                      temp(j) := temp(j) + temp((2*j)+1) ;
                  else
                       temp(j) := temp(j) ;
                  end if ;
             end loop ;
             bound := ( bound / 2 ) + ( bound mod 2 ) ;
        end loop ;
        output(d) <= temp(0)(7+NUM_INTERP_BITS
                        downto NUM_INTERP_BITS) ;
     end loop ;
end process ;

------------------------------------------------
end nonsym_pruned_tetrtl ;
------------------------------------------------
```

Figure 28. Non-Symmetric Pruned Tetrahedral Interpolation VHDL Implementation

```
----------------------------------------------------
-- Non-Symmetric Pruned Tetrahedral Interpolation --
----------------------------------------------------
entity nonsym_radial_tetr is
port (
     -- Interpolation Select --
     tetra_not_radial: in  STD_LOGIC ;
     -- Input Pixel --
     input:            in  INPUT_ARRAY ;
     -- Vertex Points of Cube from Color Table --
     point:            in  POINT_ARRAY ;
     -- Lattice Resolution --
     resolution:       in  RESOLUTION_ARRAY ;
     -- Output Pixel --
     output:           out OUTPUT_ARRAY
) ;
end nonsym_pruned_tetrahedral ;

-----------------------------------------------------
architecture nonsym_radial_tetrtl of nonsym_radial_tetr is
-----------------------------------------------------

 constant VERTEX_COUNT: INTEGER := 2**NUM_INTERP_BITS;
 signal   mask:         MASK_ARRAY ;
 signal   vertex:       VERTEX_ARRAY ;
 signal   point_value:  POINT_VALUE_ARRAY ;

begin

-----------------------------------------------------
-- Compute Lattice Resolution Masks               --
-----------------------------------------------------
MASKS: process ( resolution )
     variable  top:           INTEGER ;
begin
     for i in 0 to 2 loop
       top := 0 ;
       for j in 2 downto 0 loop
            if ( resolution(i)(j) = '1' ) then
                 top := top + 2**j ;
            else
                 top := top ;
            end if ;
       end loop ;
       for j in NUM_INTERP_BITS-1 downto 0 loop
            if ( j < top ) then
                 mask(j)(i) <= '1' ;
            else
                 mask(j)(i) <= '0' ;
            end if ;
       end loop ;
     end loop ;
end process ;

-----------------------------------------------------
-- Compute vertices.                               --
-----------------------------------------------------
VERTICIES: process ( tetra_not_radial, input, mask )
     variable  temp1, temp2:   VERTEX_ARRAY ;
     variable  limit:          INTEGER ;
begin
     temp2 := ( others => "000" ) ;
     for i in 0 to NUM_INTERP_BITS-1 loop
          limit := ( 2**(i+1) ) - 2 ;
          temp2(limit) := mask(i) and ( input(0)(i)
                      & input(1)(i) & input(2)(i) ) ;
          temp1 := temp2 ;
          for j in (VERTEX_COUNT/2)-1 downto 0 loop
               if ( j < limit/2 ) then
                    ---------------------------------
                    -- Select Radial or Tetrahedral--
                    ---------------------------------
                    if ( tetra_not_radial = '0' ) then
                        ------------
                        -- Radial --
                        ------------
                        --------------------
                        -- P'[2*j] = P[j] --
                        --------------------
                        temp2(2*j)     := temp1(j) ;
                        ------------------------------
                        -- P'[2*j+1]=v[i]|(P[j]&~m[i])
                        ------------------------------
                        temp2((2*j)+1) := temp1(limit)
                            or (temp1(j) and not mask(i));
                    else
                        ------------------------------
                        -- NonSym Pruned Tetrahedral -
                        ------------------------------
                        ------------------------------
                        -- P'[2*j]=(v[i]|~m[i])&P[j]--
                        ------------------------------
                        temp2(2*j)   := ( temp1(limit)
                           or not mask(i) ) and temp1(j) ;
                        ----------------------------
                        -- P'[2*j+1] = v[i]|P[j] --
                        ----------------------------
                        temp2((2*j)+1) := temp1(limit)
                                          or temp1(j) ;
                    end if ;
               else
                    temp2 := temp2 ;
               end if ;
          end loop ;
     end loop ;
     vertex <= temp2 ;
end process ;

-----------------------------------------------------
-- Select the values of vertices to be used by the --
-- interpolation. The last two vertices are always --
-- point(0) and the rounding constant.            --
-----------------------------------------------------
MUX: process ( vertex, point )
begin
     for d in 0 to NUM_OUTPUT_DIMEN-1 loop
        point_value(d) <= (others => (others => '0'));
        for i in 0 to VERTEX_COUNT-2 loop
          case vertex(i) is
              when "000"  => point_value(d)(i)(7 downto 0)
                                     <= point(d)(0) ;
              when "001"  => point_value(d)(i)(7 downto 0)
                                     <= point(d)(1) ;
              when "010"  => point_value(d)(i)(7 downto 0)
                                     <= point(d)(2) ;
              when "011"  => point_value(d)(i)(7 downto 0)
                                     <= point(d)(3) ;
              when "100"  => point_value(d)(i)(7 downto 0)
                                     <= point(d)(4) ;
              when "101"  => point_value(d)(i)(7 downto 0)
                                     <= point(d)(5) ;
              when "110"  => point_value(d)(i)(7 downto 0)
                                     <= point(d)(6) ;
              when others => point_value(d)(i)(7 downto 0)
                                     <= point(d)(7) ;
          end case ;
        end loop ;
        point_value(d)(VERTEX_COUNT-1)
                        (7 downto 0) <= point(d)(0) ;
        point_value(d)(VERTEX_COUNT)
                        (NUM_INTERP_BITS-1) <= '1' ;
     end loop ;
end process ;

-----------------------------------------------------
-- Select the values of vertices to be used by the --
-- interpolation. The last two vertices are always --
-- point(0) and the rounding constant.            --
-----------------------------------------------------
SUM: process ( point_value )
     variable  temp:      POINT_VALUES ;
     variable  bound:     INTEGER ;
begin
     for d in 0 to NUM_OUTPUT_DIMEN-1 loop
        temp := point_value(d) ;
        bound := VERTEX_COUNT + 1 ;
        for i in 0 to NUM_INTERP_BITS loop
             for j in 0 to VERTEX_COUNT/2 loop
                  if ( 2*j < bound ) then
                       temp(j) := temp(2*j) ;
                  else
                       temp(j) := (others => '0') ;
                  end if ;
                  if ( (2*j)+1 < bound ) then
                      temp(j) := temp(j) + temp((2*j)+1) ;
                  else
                       temp(j) := temp(j) ;
                  end if ;
             end loop ;
             bound := ( bound / 2 ) + ( bound mod 2 ) ;
        end loop ;
        output(d) <= temp(0)(7+NUM_INTERP_BITS
                        downto NUM_INTERP_BITS) ;
     end loop ;
end process ;
------------------------------------------------
end nonsym_radial_tetrtl ;
------------------------------------------------
```

Figure 29. Common Non-Symmetric Radial and Pruned Tetrahedral VHDL Implementation

3. Summary

Presented are two new interpolation approaches based on Cube Subdivision. The first, Radial Interpolation, exploits the observation that the subcube in each iteration can be generated by averaging the vertex of the cube indexed by the slice across the input values with all other vertices of the cube. This leads to further optimizations, resulting in Radial Interpolation requiring only *n+1* table accesses and *n+1* additions to generate each pixel output component. The second, Pruned Tetrahedral Interpolation, takes advantage of pruning techniques to implement a tetrahedral interpolation in 2^n additions for each pixel output component. Also presented is an implementation that can perform both Radial and Pruned Tetrahedral Interpolations. Finally, Non-Symmetric Radial and Pruned Tetrahedral Interpolation approaches are presented that allow for interpolation with a non-symmetric color table.

Table 1 presents the total number of ALU operations and memory accesses to convert a 24 bit RGB pixel (d=3) to a 32 bit CMYK pixel (D=4), all with n=4. The D=1 column presents the cost of conversion for just one output component (C, M, Y, or K).

Table 1: Number Operations to Convert 24 bit RGB to 32 bit CMYK with n=4

	ALU Operations		Memory Accesses	
	D=1	D=4	D=1	D=4
Radial	25	52	5	20
Pruned Tetrahedral	59	110	8	32
Non-Symmetric Radial	92	163	8	32
Non-Symmetric Pruned Tetrahedral	109	180	8	32

It is understood that processor specific code optimizations can reduce the number of operations. For example, packing all components (C,M,Y, and K) at a lattice location into one memory word reduces the memory accesses by a factor of four. However, this comes at the cost of additional extraction operations of 4 mask and 3 shift operations for each access. Also, the software implementation can take advantage of 32 bit and 64 bit ALUs to add multiple components in a single operation. Both of these processor specific optimizations allow the software implementation of the RGB to CMYK conversion to approach the complexity of the D=1 conversion. These optimizations are further discussed in Appendix B.

Table 2 presents the number of hardware elements (Adders, Multiplexers, and additional AND/OR Gates to compute indices) needed to implement the various approaches. Also note that the number of additional AND and OR gates to compute the indices are independent of the number of output components D. This occurs because the same indices are used for all output components. Lastly, It is understood that the D=1 hardware implementation can be used even when converting to multiple output components. In this case, the hardware can be shared, via time-multiplexing, for all the output components. This is a reasonable implementation because the speed of the conversion is typically limited by the memory accesses time to bring in the data to be translated and to write out the translated data. Thus, in current technologies, plenty of time exists to time-multiplex the components through the conversion circuitry.

Table 2: Number Hardware Elements to Convert 24 bit RGB to 32 bit CMYK with n=4

	Adders		Multiplexers		AND/OR Gates	
	D=1	D=4	D=1	D=4	D=1	D=4
Radial	5	20	5	20	0	0
Pruned Tetrahedral	16	64	15	60	66	66
Non-Symmetric Radial	16	64	15	60	153	153
Non-Symmetric Pruned Tetrahedral	16	64	15	60	204	204

Appendix A. VHDL Package (Header) file

Figure 30 shows the VHDL package (header) file used for the VHDL implementations presented.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;
--------------------------------------------------

package interpolation_pkg is

  constant NUM_INTERP_BITS:         INTEGER := 4 ;
  constant NUM_OUTPUT_DIMEN:        INTEGER := 4 ;

  type INPUT_ARRAY is array (2 downto 0) of STD_LOGIC_VECTOR(NUM_INTERP_BITS-1 downto 0) ;
  type CUBE_POINTS is array (7 downto 0) of STD_LOGIC_VECTOR(7 downto 0) ;
  type POINT_ARRAY is array (NUM_OUTPUT_DIMEN-1 downto 0) of CUBE_POINTS ;
  type OUTPUT_ARRAY is array (NUM_OUTPUT_DIMEN-1 downto 0) of STD_LOGIC_VECTOR(7 downto 0) ;
  type VERTEX_ARRAY is array (NUM_INTERP_BITS-1 downto 0) of STD_LOGIC_VECTOR(2 downto 0) ;
  type POINT_VALUES is array (NUM_INTERP_BITS+1 downto 0) of STD_LOGIC_VECTOR(7+NUM_INTERP_BITS downto 0);
  type POINT_VALUE_ARRAY is array (NUM_OUTPUT_DIMEN-1 downto 0) of POINT_VALUES ;
  type RESOLUTION_ARRAY is array (2 downto 0) of STD_LOGIC_VECTOR(2 downto 0) ;
  type MASK_ARRAY is array (VERTEX_COUNT-2 downto 0) of STD_LOGIC_VECTOR(2 downto 0) ;

end interpolation_pkg ;
------------------------------------------
package body interpolation_pkg is
------------------------------------------
end interpolation_pkg ;
```

Figure 30. Package File for VHDL Implementations

Appendix B. C Code Optimizations

By packing all the color components for a lattice point into the same word, the number of memory references can be reduced. Figure 31 presents an alternative implementation to that presented in Figure 12. Notice that the number of memory operations is reduce to five accesses, which comes at the cost of additional mask and shift operations. This optimization can be applied to the other presented interpolations. Table 3 presents the number of operations and memory accesses for the various interpolation techniques with this optimization. It should be understood, the presents optimization does not apply to the hardware implementation.

```
#define C_INC     1
#define B_INC     C_INC*17
#define A_INC     B_INC*17
#define TABLE_INC A_INC*17
/******************************************/
/** radial_interpolation()               **/
/******************************************/
void radial_interpolation(unsigned char input[],
unsigned int table[TABLE_INC], unsigned char output[])
{
    register unsigned  origin, p0, p1, p2, p3, p4 ;
    register int       a, b, c ;
    register int       w, x, y, z ;

    /**************************************/
    /* Snap max value to last table value */
    /**************************************/
    a = ( input[0] == 0x0ff ) ? 0x100 : (int) input[0] ;
    b = ( input[1] == 0x0ff ) ? 0x100 : (int) input[1] ;
    c = ( input[2] == 0x0ff ) ? 0x100 : (int) input[2] ;

    /***************************************/
    /* Compute Indexes into table          */
    /***************************************/
    origin = A_INC*(a >> 4)
           + B_INC*(b >> 4)
           +       (c >> 4) ;

    p0 = table[ origin ] ;
    p1 = table[ origin + A_INC*( (a & 0x01) )
                       + B_INC*( (b & 0x01) )
                       +         (c & 0x01) ] ;
    p2 = table[ origin + A_INC*( (a & 0x02) >> 1 )
                       + B_INC*( (b & 0x02) >> 1 )
                       +         (c & 0x02) >> 1 ] ;
    p3 = table[ origin + A_INC*( (a & 0x04) >> 2 )
                       + B_INC*( (b & 0x04) >> 2 )
                       +         (c & 0x04) >> 2 ] ;
    p4 = table[ origin + A_INC*( (a & 0x08) >> 3 )
                       + B_INC*( (b & 0x08) >> 3 )
                       +         (c & 0x08) >> 3 ] ;

    /*************/
    /* Compute w */
    /*************/
    w = ( (   p0                >> 24 )
        + (   p1                >> 24 )
        + ( ( p2 & 0xff000000 ) >> 23 )
        + ( ( p3 & 0xff000000 ) >> 22 )
        + ( ( p4 & 0xff000000 ) >> 21 )
        +   0x08 ) >> 4 ;        /* Round */

    /*************/
    /* Compute x */
    /*************/
    x = ( ( ( p0 & 0x00ff0000 ) >> 16 )
        + ( ( p1 & 0x00ff0000 ) >> 16 )
        + ( ( p2 & 0x00ff0000 ) >> 15 )
        + ( ( p3 & 0x00ff0000 ) >> 14 )
        + ( ( p4 & 0x00ff0000 ) >> 13 )
        +   0x08 ) >> 4 ;        /* Round */

    /*************/
    /* Compute y */
    /*************/
    y = ( ( ( p0 & 0x0000ff00 ) >> 8 )
        + ( ( p1 & 0x0000ff00 ) >> 8 )
        + ( ( p2 & 0x0000ff00 ) >> 7 )
        + ( ( p3 & 0x0000ff00 ) >> 6 )
        + ( ( p4 & 0x0000ff00 ) >> 5 )
        +   0x08 ) >> 4 ;        /* Round */

    /*************/
    /* Compute z */
    /*************/
    z = (   ( p0 & 0x000000ff )
        +   ( p1 & 0x000000ff )
        + ( ( p2 & 0x000000ff ) << 1 )
        + ( ( p3 & 0x000000ff ) << 2 )
        + ( ( p4 & 0x000000ff ) << 3 )
        +   0x08 ) >> 4 ;        /* Round */

    output[0] = (char) w ;
    output[1] = (char) x ;
    output[2] = (char) y ;
    output[3] = (char) z ;
}
```

Figure 31. Radial Interpolation C Implementation with Processor Specific Optimizations

Table 3: Number Operations to Convert 24 bit RGB to 32 bit CMYK with Optimizations

	ALU Operations		Memory Accesses	
	D=1	D=4	D=1	D=4
Radial	30	76	5	5
Pruned Tetrahedral	67	166	8	8
Non-Symmetric Radial	100	219	8	8
Non-Symmetric Pruned Tetrahedral	117	236	8	8

References

[1] Clark, D., Strong, D., and White, T., *Method of Color Conversion with Improved Interpolation*, U.S. Patent 4477833, Oct. 1984, pp. 1-14.

[2] Gondek, J., *Subdivision Based Colormap Interpolation*, Invention Disclosure, 1994, pp. 1-16.

[3] Franklin, P., *Interpolation Methods and Apparatus*, U.S. Patent 4334240, July 1980, pp. 1-10.

[4] Ikegami, H., *Method for Transforming Color Signal and Apparatus for Executing the Method*, U.S. Patent 5313314, May 1994, pp. 1-20.

[5] Imao, K. and Ohuchi, S., *Interpolation Method and Color Correction Method Using Interpolation*, U.S. Patent 5311332, May 1994, pp. 1-20.

[6] Kanamori, K., Yamada, O., Motomura, H., Kurosawa, T., and Fumoto, T., *Method and Apparatus for Color Conversion*, U.S. Patent 5428465, pp. 1-18.

[7] Kanamori, K., Yamada, O., Motomura, H., Rika, H., Fumoto, T., and Kotera, H., *Color Converting Apparatus for Performing a Three-Dimensional Color Conversion of a Colored Picture in a Color Space with a Small Capacity of Memory*, U.S. Patent 5504821, Apr. 1996, pp. 1-60.

[8] Kasson, J., Nin, S., Plouffe, W., and Hafner, J., *Performing Color Space Conversions with Three-Dimensional Linear Interpolation*, Journal of Electronic Imaging, 4(3), July 1995, pp. 226-250.

[9] Kasson, J., Plouffe, W., *An Analysis of Selected Computer Interchange Color Spaces*, ACM Transactions on Graphics, 11(4), Oct. 1992, pp. 373-405.

[10] Pugsley, P. Colour Correcting Image Reproducing Methods and Apparatus, U.S. Patent 3893166, July 1, 1975, pp. 1-12.

[11] Pugsley, P., *Image Reproducing Methods and Apparatus*, U.K. Patent 1369702, Oct. 1974.

[12] Sakamoto, T., *Linear Interpolating Method and Color Conversion Apparatus Using This Method*, U.S. Patent 4511989, Apr. 1985, pp. 1-18.

[13] Sakamoto, T. and Itooka, A., *Linear Interpolator for Color Correction*, U.S. Patent 4275413, June 1981, pp. 1-16.

[14] Stone, M., Cowan, W., Beatty, J., *Color Gamut Mapping and the Printing of Digital Color Images*, ACM Transactions on Graphics, 7(4), Oct. 1988, pp. 249-292.

[15] Van de Capelle, J., Roose, K., Debaere, E., and Pletinckx, D., *Method and a Device for Converting a Color Coordinate Set*, U.S. Patent 5268754, Dec. 7, 1993, pp. 1-20.

[16] Vondran, G., *Apparatus for Generating Interplator Input Data*, U.S. Patent 5717507, Feb. 1998.

[17] Vondran, G., *Apparatus for Routing Interpolator Input Data by Performing a Selective Two's Complement Based on Sets of Lower and Higher Order Bits of Input Data*, U.S. Patent 5666437, Sept. 1997, pp. 1-20.

[18] Vondran, G., *Common Non-Symmetric Pruned Radial and Non-Symmetric Pruned Tetrahedral Interpolation Hardware Implementation*, Patent Application Docket No. 10971846-1, pp. 1-61.

[19] Vondran, G., *Common Pruned Radial and Pruned Tetrahedral Interpolation Hardware Implementation*, Patent Application Docket No. 10970157-1, pp. 1-60.

[20] Vondran, G., *Non-Symmetric Radial and Non-Symmetric Pruned Radial Interpolation*, Patent Application Docket No. 10970158-1, pp. 1-62.

[21] Vondran, G., *Non-Symmetric Tetrahedral and Non-Symmetric Pruned Tetrahedral Interpolation*, Patent Application Docket No. 10970156-1, pp. 1-64.

[22] Vondran, G., *Radial and Pruned Radial Interpolation*, Patent Application Docket No. 10970159-1, pp. 1-65.

[23] Vondran, G., *Tetrahedral and Pruned Tetrahedral Interpolation*, Patent Application Docket No. 10970160-1, pp. 1-67.