

Quadrees, Transforms and Image Coding

R. R. Martin & M. M. Anguh
Department of Computing Mathematics,
University of Wales College of Cardiff

Revised 23 April 1991

Abstract

Transforms and quadrees are both methods of representing information in an image in terms of the presence of information at differing length scales. This paper presents a mathematical relationship between these two approaches to describing images in the particular case when Walsh transforms are used. Furthermore, both methods have been used for the compression of images for transmission. This paper notes that under certain circumstances, quadtree compression produces identical results to Walsh transform coding, but requires less computational effort to do so. Remarks are also made about the differences between these approaches.

1 Introduction

Images represented as an array of pixels typically take up large amounts of storage, and hence large times to transmit over computer communication links. Thus, various methods have been investigated for the compression of such images. Whilst general purpose compression schemes such as Huffmann coding have been used, other methods take note of the fact that an image is a two-dimensional array containing structured information rather than random values. The idea of area coherence is thus used to lead to compression techniques specifically for image data. Two particular approaches are those based on transforms, and on the quadtree. However, both of these techniques basically represent the information in the image in terms of its components at different length (or area) scales. It is thus natural to seek if there is some connection between these representations.

The first result of this paper is to show a mathematical relationship that exists between the components of the Walsh transform of an image, and the values stored in nodes of a particular method of representing image data as a quadtree.

Secondly, the Walsh transform is a typical transform, and gives similar results to other transform methods when used for lossy image compression (when some loss of information is acceptable in producing the compressed image). Using our first result, we show that if Walsh transform coding is performed in a certain way, its usage leads to identical results to the given quadtree method, in that the reconstructed image in each case is the same.

However, the quadtree method for compression is faster than Walsh (and other) transform based methods, and hence, in principle, quadtree methods should be used in preference to transform methods when appropriate circumstances arise.

1.1 Walsh Transforms

Perhaps the most well known transform is the Fourier transform, the discrete version of which is defined in one dimension by the sum

$$F(u) = \frac{1}{N} \sum_{x=0}^{N-1} f(x) e^{-2\pi i u x / N}, \quad (1)$$

where u takes values $0, 1, \dots, N-1$. Each transform value $F(u)$ is a weighted sum of the original $f(x)$ values. (Here and throughout the paper we will assume N to be a power of 2, such that $N = 2^n$.)

The discrete Walsh transform can also be thought of as computing similar weighted sums, where we now have

$$W(u) = \frac{1}{N} \sum_{x=0}^{N-1} f(x) \prod_{i=0}^{n-1} (-1)^{b_i(x) b_{n-1-i}(u)}, \quad (2)$$

and $b_k(r)$ is the k th bit in the binary representation of r , b_0 being the least significant bit. Despite the complicated appearance of the above formulation, the weighting factors in the case of the Walsh transform are considerably simpler than those in the Fourier transform, just being either $+1$ or -1 . These weighting factors for $N = 4$ are shown in Table 1. The set of weighting factors will in future be referred to as the *kernel* of the transform.

$u \setminus x$	0	1	2	3
0	+1	+1	+1	+1
1	+1	+1	-1	-1
2	+1	-1	+1	-1
3	+1	-1	-1	+1

Table 1: The Walsh weighting functions for $N = 4$

Exactly the same weighting factors can be used to compute the inverse Walsh transform, apart from a factor of $1/N$, so

$$f(x) = \sum_{u=0}^{N-1} W(u) \prod_{i=0}^{n-1} (-1)^{b_i(x) b_{n-1-i}(u)}. \quad (3)$$

The Walsh Transform of a two-dimensional array of data can be defined in an analogous manner as

$$W(u, v) = \frac{1}{N} \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x, y) \prod_{i=0}^{n-1} (-1)^{b_i(x) b_{n-1-i}(u)} \prod_{j=0}^{n-1} (-1)^{b_j(y) b_{n-1-j}(v)}. \quad (4)$$

There is now a factor of $1/N$ in the inverse transform as well as the forward transform.

1.2 Quadrees and Binary Trees

We will begin by considering two-dimensional data, and its representation by quadrees. We will then look at binary trees as the one-dimensional analogue of quadrees. A thorough account of quadrees and their applications can be found in the pair of books by Samet [12, 13].

Consider an image of size $N \times N$ pixels, $N = 2^n$. A quadtree can be recursively computed by Algorithm 1.

Algorithm 1

- 1 *Create a node, called the root node, which represents the whole image.*
- 2 *Set the region of interest to be the whole image. Set the current node to be the root node.*
- 3.1 *If the region of interest is a single colour, set the current node to contain that colour.*
- 3.2.1 *Otherwise, split the region of interest into four sub-regions by dividing it in half vertically and horizontally.*
- 3.2.2 *Represent each of these new regions by a child node of the current node.*
- 3.2.3 *Set each of these sub-regions to be the region of interest in turn, and their node to be the current node, returning to Step 3.1 each time.*

Note that the recursion stops when single pixels are reached if not before, as these are guaranteed to be a single colour. Note also that in practice, the quadtree would be generated bottom up, not top down as the algorithm suggests. It can be seen that all the nodes of the quadtree are either terminal nodes, which store a single colour representing some square part of the image of size $2^j \times 2^j$ pixels for some j , $0 \leq j \leq n$, or they are non-terminal nodes, which have four children representing each of their quadrants.

We will also be interested in a variation of the basic quadtree above, where each of the non-terminal nodes also stores data. In particular, at each non-terminal node let us store the average value of the colours stored at each of its children. This will be discussed further later.

The one-dimensional analogue of the quadtree is a particular type of binary tree. Given an array of data, each node is either a non-terminal node with two children representing the left and right halves of a given segment of the array, or a terminal node indicating that all of the values in the given segment are equal. From now on, any reference to a binary tree will mean this type of tree. Again, we can also give the non-terminal nodes average values as above.

2 Image Coding for Transmission

Both quadtrees and various transforms have been used for image compression. We will firstly review the methods used in each case, and then using the results of the previous section, we will show that under certain circumstances, both approaches can lead to identical results. However, both approaches are not equally efficient in producing these results.

In both cases of image coding, the basic idea is that we have some computer network or other communication link of limited bandwidth, and that we wish to reduce the amount of information to be transmitted in order to reduce transmission times and/or transmission costs. Image compression techniques can be classified into two categories — those (lossless) from which the image can be perfectly reconstructed, and those (lossy) which discard some information. In the latter case the objective is to produce an image which after reconstruction appears as close as possible to the original as far as the human eye is concerned. We may be prepared to accept a small amount of degradation in the image if the savings are great enough. In what follows, we shall generally consider lossy image compression techniques.

2.1 Transform Coding of Images

A good introduction to the use of transforms for image encoding may be found in Gonzalez & Wintz [5], while a fuller treatment may be found in the book devoted to the subject by Clarke [2]. A summary of the main idea is presented below.

Given a two-dimensional image of size $N \times N$, it is first subdivided into a set of smaller subimages, each of size $M \times M$, where for the purposes of this paper M will also be assumed to be a power of 2. Each of these subimages is then encoded and transmitted independently of the other subimages. The purpose of dividing into subimages is to best allow for large variations between different regions of the image.

Let us start by considering a one-dimensional image. Let us divide it into pairs of pixels, and for each pair, compute the Walsh transform independently for each pair. Thus, if our original image is x_i, x_{i+1} for $i = 0, 2, \dots, N - 2$, after performing the transform we have u_i, u_{i+1} for $i = 0, 2, \dots, N - 2$, where

$$u_i = \frac{x_i + x_{i+1}}{2}, \quad u_{i+1} = \frac{x_i - x_{i+1}}{2}, \quad i = 0, 2, \dots, N - 2. \quad (5)$$

In this case, if we assume the x_i are varying slowly across the image in general, the u_i for even i are just the average of the corresponding pair x_i, x_{i+1} , and will thus be about the same size as either x_i or x_{i+1} . On the other hand, the u_{i+1} values are the differences between the pixel values in a pair, and on the assumption that the pixel values are changing fairly slowly, we would expect normally the u_{i+1} values to be rather smaller than x_i or x_{i+1} .

Thus, for each pair, we transmit the u_i values for even i with as many bits as we would use for the x_i values, while we transmit the u_{i+1} values for even i with a reduced number of bits, thus compressing the data. For many pairs, we will still be able to reconstruct x_i and x_{i+1} exactly, and only in the neighbourhood of edges, for example, where there is a large

difference between x_i and x_{i+1} will there be an error. This error will not be cumulative, however, and will only affect a given pair.

Indeed, we can even transmit the u_{i+1} values for even i with zero bits, in other words, not transmit them at all. In this case, on reconstruction, each pair of pixels x_i and x_{i+1} will just be recreated as equal values, each equal to the previous average of the pair. Whilst this will be a degradation of the image, it will nevertheless be a recognisable and usable one in most cases.

Returning to the general case, we now compute an $M \times M$ Walsh Transform for each subimage. Generalising the above observations, we expect the lowest frequency Walsh components to be largest, and the higher frequency ones to be smaller. Thus, again, the higher frequency components are transmitted with a reduced number of bits, or are simply not transmitted at all.

In practice, the Walsh Transform does not have to be used, but any other similar transform such as the Fourier or Hotelling Transform can also be used. It is found that the relative advantages of using one type of transform over another are quite small. To give an idea of the fidelity of transform encoding, Gonzalez & Wintz [5] give an example of an image broken into 16×16 pixel subimages, where the 128 higher frequency components are simply discarded, and on reconstruction, each of the types of transform mentioned above give mean square errors in the reconstructed pixel values of between $\frac{1}{3}\%$ and $\frac{1}{2}\%$.

2.2 Quadtree Coding of Images

Various authors have proposed the use of quadtrees [7, 9, 14] or bintrees [6, 8] for transmission of images, particularly in the context of improving user interaction over low-speed transmission links. Here, the image is stored in quadtree form, except that at a non-terminal node of the quadtree we store the average value of its children. A quadtree for an $N \times N$ image, $N = 2^n$, will in general have nodes at $n + 1$ levels. Thus, assuming there are some terminal nodes at this lowest level, the quadtree down to this lowest level represents the image at full resolution. However, if we omit the nodes at the lowest level, and make their parents into new terminal nodes, we now have a new image with resolution $N/2 \times N/2$, where the new “pixels” at the lower resolution are the average of the four corresponding neighbouring pixels at higher resolution. Obviously, this process can be repeated to produce further images with resolution $N/4 \times N/4$, $N/8 \times N/8$ and so on by discarding successive levels of the quadtree upwards.¹

The way in which this technique is usually used is to transmit the tree in breadth-first order: first the root node is transmitted, then all of its children, then all of their children, and so on. In this way, the user sees an image in which the whole screen is always an approximation to the desired image, but with increasingly higher resolution, until finally the lowest level nodes have arrived and the image is presented at full resolution. The important thing to note is that this can be very useful for rapid interaction. The user may be able to decide after just a few levels of the quadtree have been transmitted that he does

¹Other rules such as maximum, median, and so on may also be used instead of averaging in constructing the parent’s value from the child values in generalised versions of this process.

not want to see any more of this image, but perhaps move on to the next one. A relatively small amount of transmitted information is required in such cases compared to the total image size. The disadvantage is that if the image is finally desired at full resolution, the transmission of all the intermediate nodes in the tree as well as the terminal nodes will roughly increase the transmission time by a factor of $1\frac{1}{3}$. This can be reduced by intelligently reusing information already gained from each previous level, and indeed [6, 8] mainly concern themselves with this issue. However, for many applications most images, and perhaps all, will not need to be viewed at full resolution, giving an overall saving. Alternatively, after seeing the image at low resolution, the user may interactively decide that he only needs a certain part of the image at higher resolution [7].

Note that for some other applications, quadrees may be used for image compression in a rather different way. Here, no values are calculated or transmitted for the non-terminal nodes of the tree, but only values for the terminal nodes, plus sufficient information to determine where the nodes are in the tree [1, 3, 4, 7, 11, 15]. Here, it is hoped that the coherence of the image (*i.e.* large areas of a single colour) will mean that not all of the terminal nodes are at the lowest possible level of the tree, and that the reduced number of node values to be transmitted (compared to the number of pixels) will more than compensate for the added information which indicates the positions in the tree. This method as described is a lossless compression method, and the image can be perfectly reconstructed. (Depending on the source of the image, neighbours having almost equal rather than exactly equal values may be much more likely. In this case, it is possible when constructing the initial quadtree to combine neighbours if their values are equal to within some tolerance, rather than require them to be absolutely equal. The method then becomes a lossy one.)

Finally, we can also use a combination of the two previous approaches (for example, see [10]). We can discard terminal nodes at the lowest level (or several lowest levels), again making their parents into terminal nodes with values equal to the average of their children. However, averages are not calculated for the interior nodes of the quadtree, this time. We now have a quadtree corresponding to an image of reduced resolution, and a correspondingly lower amount of data to be transmitted. In this case, the averaging process has discarded information, and it is *not* possible to perfectly reconstruct the original image. It is this method that we will wish to compare directly with the transform approach.

3 Relation Between Transforms and Trees

3.1 Theory

Let us consider a one dimensional array of, say four, data values, a , b , c and d from left to right. Let us also consider the Walsh transform and the binary tree which represent that data. To start off, let us assume that the binary tree is a complete tree, *i.e.* all terminal nodes are at the lowest possible level. Let us denote the root node in the tree as $Tree(0, 1)$, the nodes at the next level as $Tree(1, 1)$ and $Tree(1, 2)$ going from left to right, and the

nodes at the lowest level as $Tree(2, 1)$, $Tree(2, 2)$, $Tree(2, 3)$ and $Tree(2, 4)$, as shown in Figure 1.

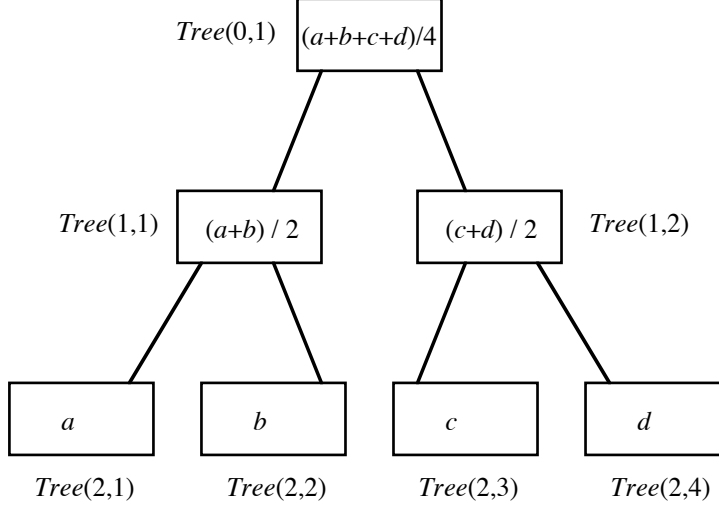


Figure 1: Binary Tree Representation

Using the information in Table 1, we can see that

$$W(0) = \frac{a + b + c + d}{4} = Tree(0, 1), \quad (6)$$

$$W(1) = \frac{a + b - c - d}{4} = \frac{1}{2} (Tree(1, 1) - Tree(1, 2)), \quad (7)$$

$$W(2) = \frac{a - b + c - d}{4} = \frac{1}{4} (Tree(2, 1) - Tree(2, 2) + Tree(2, 3) - Tree(2, 4)), \quad (8)$$

$$W(3) = \frac{a - b - c + d}{4} = \frac{1}{4} (Tree(2, 1) - Tree(2, 2) - Tree(2, 3) + Tree(2, 4)). \quad (9)$$

Note that $W(0)$ *only* depends on the information at the root level of the tree, whilst $W(1)$ only depends on information at level 1, and finally $W(2)$ and $W(3)$ both depend on information at level 2.

This observation can be generalised to an array of $N = 2^n$ values:

Theorem 1 *Given an array of data represented by a complete binary tree $Tree(i, k)$, its Walsh transform coefficients $W(2^{i-1})$ to $W(2^i - 1)$ depend only on the values $Tree(i, 1)$, $Tree(i, 2), \dots, Tree(i, 2^i)$, all at level i of the tree, for $i = 1, \dots, n$.*

To see this, consider Equation 2. The sign of the kernel can be -1 if and only if both $b_{n-1-i}(u) = 1$ and $b_i(x) = 1$ for some i . Let the binary representation of u have α leading zeros when expressed as an n -bit binary number. Then $b_{n-1-i}(u) = b_i(v)$ where v is the binary number obtained by writing the bits of u in reverse order. Thus, the lowest 1 bit is in position α in v . As we increase x , $b_i(x)b_i(v)$ can only change sign whenever the α th or higher order bits of x change. Thus, for a value of u with α leading zeros, the runs of

+1 or -1 values in the kernel must be 2^α long, and they also start when the α th bit of x changes. This corresponds exactly, however, to how the information is grouped within the binary tree.

Let us now suppose that the binary tree is no longer a complete tree. Let us suppose that in computing $W(k)$ we need the value of $Tree(l, m)$ for some l and m , but that node $Tree(l, m)$ does not exist. If this is the case, it is because node $Tree(l, m)$ has the same value as its parent, $Tree(l-1, \lfloor m/2 \rfloor)$, and the parent node has not been subdivided². Thus, we can just use the value $Tree(l-1, \lfloor m/2 \rfloor)$ in our computation instead. (In practice the parent node may not exist either, whereupon this process can be repeated, as often as necessary).

Theorem 2 *Given an array of data represented by any binary tree $Tree(i, k)$, Walsh transform coefficients $W(2^{i-1})$ to $W(2^i - 1)$ only depend upon the tree at levels i and above, and do not depend on lower levels of the tree.*

Finally, let us take the same argument one step further. Again consider the array with four data values, its binary tree and its Walsh transform. If we look at successive pairs of values in the kernel corresponding to $W(2)$ and $W(3)$, we see that they have opposite signs, *i.e.* $([+1][-1])([+1][-1])$ for $W(2)$, and $([+1][-1])([-1][+1])$ for $W(3)$. Now, let us suppose that the binary tree representation of this data has *no* nodes at the lowest level. This means that $a = b$ and $c = d$, and hence $W(2) = 0 = W(3)$. Going back to $W(1)$, in its kernel, if we now take the data values two at a time, again, a (the only) successive pair has opposite signs, *i.e.* $([+1 + 1][-1 - 1])$. If we now suppose that the binary tree has no nodes either at the lowest level, or the next level up (in this case, just at the root node), then $a = b = c = d$, and so the above property of the kernel means that $W(1) = 0$ as well.

Again, this argument can be generalised for $N = 2^n$ values.

Theorem 3 *Given an array of $N = 2^n$ data values represented by a binary tree $Tree(i, k)$, if levels j to n inclusive of the binary tree are all empty, then $W(u) = 0$ for $u \geq 2^{j-1}$.*

The justification for this follows a similar line of reasoning as above. The last $N/2$ values of u have a most significant bit of 1, and when u is bit-reversed to give v , this becomes the least significant bit of v . Thus, as successive pairs of values of x have one value with the least significant bit as 0, and the other as 1, while the other bits are the same, the kernel must take on both possible signs within a pair. Now take the previous $N/4$ values of u . Their most significant bit is 0, but their next most significant bit is 1. The same reversal argument now leads us to deduce that adjacent pairs of kernel values have the *same* sign, but two consecutive pairs starting with an even pair have *opposite* signs. This argument can be extended to appropriately sized blocks for all values of u by considering the number of leading zeros in u .

In fact, the same argument also shows that the following stronger form of Theorem 2 equivalent to Theorem 1 exists:

²The notation $\lfloor x \rfloor$ means the largest integer less than or equal to x .

Theorem 4 *Given an array of data represented by a binary tree $Tree(i, k)$, the Walsh transform coefficients $W(2^{i-1})$ to $W(2^i - 1)$ only depend upon the data in the tree at level i .*

Any data at a higher level in the tree can not affect these coefficients as the alternating signs in the kernel will ensure its effect cancels out.

As an aside, it should be noted that the above arguments do not directly give an efficient method of computing the Walsh transform of data held in the form of a tree. Such procedures are currently the subject of further research.

Let us now consider the extension of the above ideas to two dimensions. Standard results in transform theory show that a two-dimensional Walsh transform for an $N \times N$ array can be computed by first taking the one-dimensional Walsh transform of each row of the input data to produce a new array, and then taking a one-dimensional Walsh transform of each column of that array to produce the desired two-dimensional Walsh transform. Now, if we consider the two-dimensional data stored in the form of a quadtree, it is quite clear that if the quadtree has no nodes below a given level, the corresponding binary tree for a row of data will have the same depth, and so each of the row transforms will have zeros for the corresponding higher components. When we come to process the data column-wise, the row transforms have not mixed any of the data between one row and the next, and so we can also deduce that there may be zeros resulting in the higher components of each column transform depending on the depth of the quadtree. Finally, taking the two directions together, we can deduce the following:

Theorem 5 *Given an array of $N \times N$, $N = 2^n$ data points represented in quadtree form with averages at non-terminal nodes, if levels j to n inclusive of the quadtree tree are all empty, then $W(u, v) = 0$ whenever $u \geq 2^{j-1}$ or $v \geq 2^{j-1}$.*

Thus, for example, if the lowest level of the quadtree is missing, only the lowest 1/4 Walsh transform components will be present.

3.2 Application to Image Coding

In their original paper on quadtree methods for image transmission, Sloan & Tanimoto [14] noted that both the quadtree and transform methods have the “prefix property”, which is that “truncating the series at any point gives an approximation to the original image”. However, using the results above, we can now go further, and state that under certain conditions these two approaches produce exactly the same results.

Theorem 6 *If an array of $N \times N$, $N = 2^n$ data points is represented in quadtree form, there will in general be nodes at levels 0 to n . Let us produce a lossy compressed version of the data by removing all nodes at levels j to n , in each case making parent nodes of removed children take on the average value of their removed children. Let us also represent the data by its Walsh transform, and discard all values of $W(u, v)$ whenever $u \geq 2^{j-1}$ or $v \geq 2^{j-1}$. On taking the inverse Walsh transform the result represents the same data as the compressed quadtree.*

Thus, in this special case, transform and quadtree image compression techniques do exactly the same thing, in that the restored image regenerated from these truncated representations will be identical.

The above relation between trees and transforms also helps us to justify how the transform coding method works. If we consider the quadtree representation of the data, we normally expect there to be relatively few terminal nodes at the lowest possible level. Keeping the explanation to one-dimension for simplicity, every pair of “missing” nodes at the lowest level will represent a contribution of 0 when the sum for $W(u)$ is computed for $u = N/2$ to $u = N - 1$; similar results are true for missing nodes at higher levels. It is readily apparent that many missing nodes at the lowest level will cause $W(u)$ for $u = N/2$ to $u = N - 1$ to be relatively smaller in size, as asserted previously. In fact, even if pairs of nodes *are* present, we may expect them often to contain almost equal values, whereupon the adjacent +1 and -1 values in the kernel will lead to only a small contribution being made to the appropriate $W(u)$ values.

3.3 Efficiency

As pointed out by Knowlton [8], tree and transform encoding methods *do not* have the same efficiency. A binary tree representation of N data values has at most $2N - 1$ nodes, and can be constructed in $O(N)$ time. On the other hand, whilst the Walsh transform has N values, a naive computation of the Walsh transform takes $O(N^2)$ time, and even using the Fast Walsh Transform algorithm (a variant on the Fast Fourier Transform, see [5] for an implementation) takes $O(N \log N)$ time.

In two dimensions, for an $N \times N$ array of data, a complete quadtree has $\lfloor 4N^2/3 \rfloor$ nodes, and can be constructed in $O(N^2)$ time, whilst a two-dimensional Walsh transform can be computed from $2N$ one-dimensional Walsh transforms as mentioned previously, taking time $O(N^2 \log N)$.

In practice, as we have observed, the whole input image is not normally subjected to a single Walsh transform when transform coding methods are used, but rather is divided into subimages of size $M \times M$, which are then transformed independently. Thus, the total amount of work performed is $O(N^2 \log M)$. Typically, M might be 16, and can be considered to be a small constant independent of N . Thus, in fact, with this modification to the basic idea, both coding methods are now $O(N^2)$. However, the added complexity involved in computing the transforms and the constant factor of $\log_2 M$ result in implementations of the transform method taking several times as long. Similar remarks also apply to reconstructing the image from its compressed form.

4 Conclusions

Quadtrees and Walsh transforms have both been used for image compression purposes. This paper shows that under certain modes of use, they both produce exactly the same result in terms of the image regenerated on decompression. Although in principle both

methods have $O(N^2)$ running time for compression and decompression of an $N \times N$ image (assuming transform coding first splits the image up into fixed size blocks), the quadtree method is inherently much simpler and more straightforward in terms of the computations involved, and it thus runs several times faster.

Nevertheless, it should perhaps be remarked that both methods still have their own peculiar advantages. The transform methods do offer some additional flexibility. Rather than just throwing away a rigid number of coefficients which make the method the same as the quadtree method, we are at liberty to discard *any* number of high frequency components, *or* to transmit these components with a lower number of bits. Doing this means that some of the high frequency information in the image is kept; it is distributed across the whole image.

On the other hand, with quadtrees, rather than averaging *all* the nodes below a given level of the quadtree, we can retain those in certain areas of the image to give extra detail in a particular region of interest. In this case, we are again keeping some high frequency information, but this time it is concentrated in particular areas we choose. This is obviously useful when interactively viewing pictures.

References

- [1] C. Annedda & L. Felican, P-Compressed Quadtrees for Image Storing *Computer Journal*, **31** (4) 353-357, 1988.
- [2] R. J. Clarke *Transform Coding of Images*. Academic Press, 1985.
- [3] W. A. Davis & X. Wang, A New Approach to Linear Quadtrees *Graphics Interface '85*, 195-202, 1985.
- [4] I. Gargantini, An Effective Way to Represent Quadtrees *Communications of the ACM*, **25** (12) 905-910, 1982.
- [5] R. C. Gonzalez & P. Wintz, *Digital Image Processing (2nd Edn.)*, Addison Wesley, 1987.
- [6] F. S. Hill, S. Walker & F. Gao, Interactive Query System using Progressive Transmission *ACM Computer Graphics*, **17** (3) 323-330, 1980.
- [7] A. Hunter & P. J. Willis, Breadth-First Quad Encoding for Networked Picture Browsing *Computers and Graphics*, **13** (4) 419-432, 1989.
- [8] K. Knowlton, Progressive Transmission of Grey-Scale and Binary Pictures by Simple, Efficient and Lossless Encoding Schemes *Proceedings of the IEEE*, **68** (7) 885-896, 1980.
- [9] D. J. Milford & P. J. Willis, Quad Encoded Display *IEE Proceedings*, **131E** (3) 70-75, 1984.

- [10] M. A. Oliver & N. E. Wiseman, Operations on Quadtree Encoded Images *Computer Journal*, **26** (4) 83-90, 1983.
- [11] H. Samet, Data Structures for Quadtree Approximation and Compression *Communications of the ACM*, **28** (9) 973-993, 1985.
- [12] H. Samet *The Design and Analysis of Spatial Data Structures* Addison Wesley, 1990.
- [13] H. Samet *Applications of Spatial Data Structures* Addison Wesley, 1990.
- [14] K. R. Sloan & S. L. Tanimoto, Progressive Refinement of Raster Images *IEEE Transactions on Computers*, **C28** (11) 871-874, 1979.
- [15] J. R. Woodwark, Compressed Quad Trees *Computer Journal*, **27** (3) 225-229, 1984.