Representing Medical Images with Partitioning Trees

K.R. Subramanian and Bruce Naylor

AT&T Bell Laboratories
Murray Hill, NJ 07974

Abstract

Discrete space representation of images arise as a consequence of the transducers between the physical and informational domains. While discrete representations (arrays of pixels) are simple, they are also verbose and structureless. We present a method of converting between a discrete space representation to a particular continuous space representation, viz. the binary space partitioning tree. The conversion is accomplished using standard discrete space operators developed for edge detection, followed by a Hough transform to generate candidate hyperplanes that are used to construct the partitioning tree. The result is a segmented and compressed image represented in continuous space suitable for elementary computer vision operations and improved image transmission/storage. The method is more noise tolerant than methods whose target is a topological representation, and more adaptive than axis-aligned spatial partitioning schemes. Affine transformations needed for interactive manipulation are fast and edges do not blur with enlargement of the image. Efficient algorithms are known for spatial operations, such as masking/clipping and compositing. We give several examples of 256x256 medical images for which we have estimated the compression to range between 1 and 0.5 bits/pixel.

Introduction

A fundamental distinction in models of space is the discrete-continuous dichotomy. The informational domain appears to be inherently discrete while the physical domain is treated as being effectively continuous. In geometric computation, discrete space representations of sets and functions were initially introduced as a by-product of the transducers needed to convert between the physical and informational domains. For example, MRI and CT devices detect electromagnetic energy at a single point in space at any given moment. This generates a time continuos 1D signal which is sampled to yield a sequence of discrete values.

Given this transducer led entry into discrete space, the question arises as to whether the resulting representation is necessarily the most suitable for every kind of geometric computation involving images. Modeling a finite region of d-space as a set of lattice points and representing this computationally as d-dimensional arrays provides a simple representation, but one that is verbose and devoid of any set/function dependent structure provided by higher-level but more complex representations. Low level operations on discrete space may have simple algorithms. But providing higher level operations, such as those required by computer vision, usually requires increased verbosity and may, because of a dependency upon image structure, require more complex algorithms than ones employing higher level descriptions. Discrete space operations must also contend with aliasing.

Our view is this: while discrete space representations give an important methodology for geometric computation, continuous space representations in general provide a better fit, since the semantic domain is intrinsically continuous. This thesis is supported by the fact that to achieve semantically correct discrete space algorithms typically entails viewing them as discrete approximations of their continuous space analogs. For example, affine transformations of discrete space representations require the reconstruction (at least conceptually) of a continuous function from the sample points, transforming the function, and then resampling it. The transformation process can be time consuming for high resolution images, and since the image is treated as a band-limited signal, edges are smeared when the image is enlarged.

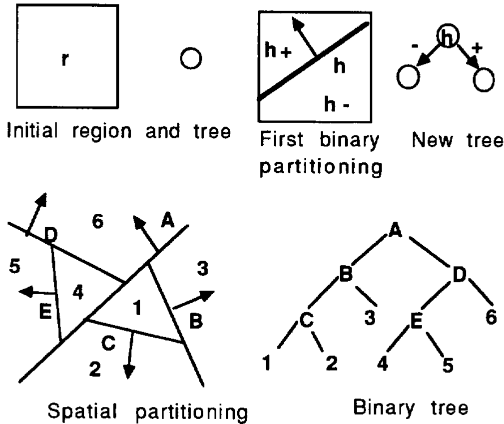
Our approach is to instead focus on the problem of converting from a discrete space representation to a particular continuous space representation, the *binary space partitioning tree* (also *partitioning tree* or *bsp tree*). This process proceeds by discovering the inherent structure in the image, yielding a segmentation of the image into regions containing no significant discontinuities, i.e. that contain only "texture". (Maximally connected components are present only in the form of a convex decomposition; however, it is possible to construct these components from their convex parts using incidence relations together with a graph search.) The segmentation provides the opportunity for compression by using more compact representations of the texture, and it can also be a significant aid in certain recognition problems. Matching could be facilitated in applications where texture is not needed, since affine transformations and set operations are very efficient when using partitioning trees, the texture is highly

0-8186-2897-9/92 $03.00 © 1992 IEEE

compressed, and moments are easily calculated. In addition, a hierarchical representation is generated that permits an efficient encoding as well as a form of multi-resolution image representation.

Partitioning Trees

Binary space partitioning trees [Fuchs, Kedem and Naylor 80] are defined via a generating algorithm, and for this only one operation is required: binary partitioning by a hyperplane of a region in a d-dimensional continuous space, d > 0. Figure 1 illustrates this. Given a homogeneous open region **r**, a hyperplane **h** that intersects **r** is chosen using some criteria. Then **h** is used to induce a binary partitioning on **r** that generates two new d-dimensional regions, $\mathbf{r^+} = \mathbf{r} \cap \mathbf{h^+}$ and $\mathbf{r^-} = \mathbf{r} \cap \mathbf{h^-}$, where $\mathbf{h^+}$ and $\mathbf{h^-}$ are the positive and negative *open* halfspaces of **h** respectively. Also, generated is a (d-1)-dimensional region $\mathbf{r^0} = \mathbf{r} \cap \mathbf{h}$, called a *sub-hyperplane* (abbr. as *shp*). Thus $\mathbf{r} = \mathbf{r^+} \cup \mathbf{r^-} \cup \mathbf{r^0} = (\mathbf{r} \cap \mathbf{h^+}) \cup (\mathbf{r} \cap \mathbf{h^-}) \cup (\mathbf{r} \cap \mathbf{h})$. Any of these new unpartitioned homogeneous regions can similarly be partitioned, and so on recursively. When the process is terminated, the remaining unpartitioned regions, called *cells*, together with the sub-hyperplanes forms a partitioning of the initial region. In figure 1, the cells are labeled with numbers and the sub-hyperplanes with letters.



Constructing a partitioning tree
Figure 1

This process, when begun with d-space as the initial region, induces a structure on d-space in the form of a hierarchical decomposition. A partitioning tree is the computational representation of this process, and its combinatorial/syntactic form is captured by a binary tree. This tree is simply the directed graph of an asymmetric relation defined on the set of regions generated by this process, where $\mathbf{r_1} \rightarrow \mathbf{r_2}$ if $\mathbf{r_2}$ was created by a partitioning of $\mathbf{r_1}$. The tree also corresponds to the graph of the partial ordering of the regions induced by the subset relation. In addition, the tree can be interpreted as a type of computation graph by interpreting the arcs as intersection operations: "moving" a set **s** contained in a region **r** and partitioned by hyperplane **h** along a left arc from **r** to $\mathbf{r^-}$ can be interpreted as computing $\mathbf{s} \cap \mathbf{h^-}$, and similarly the right arc computes $\mathbf{s} \cap \mathbf{h^+}$. This interpretation provides a set theoretic definition of any region **r'** as the intersection of open halfspaces corresponding to arcs on the path from the root to **r'**. In figure 1, **cell-3** = **2-space** $\cap$ $\mathbf{A^-} \cap \mathbf{B^+}$. Consequently, if the initial region is a convex and open set, it follows that all regions of the tree are convex and open.

Partitioning trees can represent functions whose domain and range are continuous spaces of finite dimensions $\mathbf{d_1}$ and $\mathbf{d_2}$ respectively: $\mathbf{f} : \mathbf{X} \in \mathbf{S^{d_1}} \Rightarrow \mathbf{Y} \in \mathbf{S^{d_2}}$. The partitioning tree partitions the domain into a hierarchical collection of sub-domains. Within each sub-domain a value-continuous function $\mathbf{f_i}$ defines the value of **f** within that sub-domain (typically, $\mathbf{f_i}$ is defined for all of $\mathbf{S^{d_1}}$ as well, although this is not essential). All points in $\mathbf{S^{d_1}}$ at which **f** is value-discontinuous are contained within partitioning hyperplanes. The function can be evaluated at any point **x** by following the path in the tree to the cell $\mathbf{c_i}$ that contains the point and evaluating the $\mathbf{f_i(x)}$. This is just the standard method of inserting a point into a search tree, and is commonly called *point classification* [Thibault and Naylor 87].

Partitioning Tree Representation of Images

To convert from a discrete space representation of a function to a partitioning tree representation, we need to find all points in the function at which the function is value discontinuous and then "absorb" them into partitioning hyperplanes. In the context of image representations, i.e. **f : 2-space -> color-space**, this constitutes segmentation of the image into regions containing no edges but only texture. The selection of the individual $\mathbf{f_i}$ for each subdomain used to represent the texture is not a topic we will address here, since we have only been able, up to now, to tackle the segmentation component. Consequently, we are currently using the simplest possible functions: constant-valued $\mathbf{f_i}$ corresponding to the mean value of the function in the subdomain. This should not be construed as an intrinsic component of our methodology, however. The general schema admits any function for an $\mathbf{f_i}$, and so we would expect to exploit more interesting functions in the future.

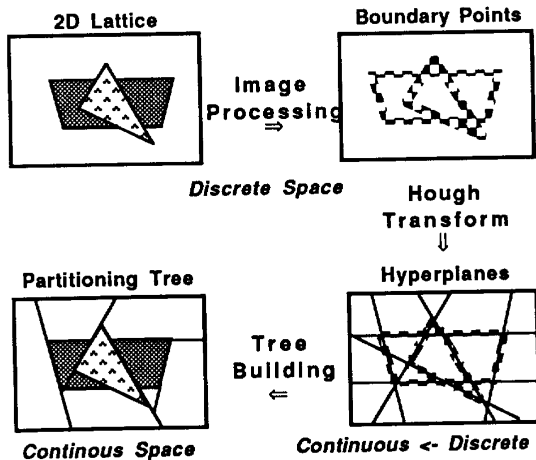
In [Thibault and Naylor 87], an algorithm was given for converting from the boundary representation of a polytope to a partitioning tree representation. We will use a modified version of this algorithm to instead convert from a discrete representation to a partitioning tree. The basic idea is this. We know that the schema for representing functions has as a necessary condition the requirement that all discontinuities, i.e. boundary points, lie on the sub-hyperplanes of the tree. Therefore, the hyperplane of

an edge must be among the set of partitioning hyperplanes, if the boundary points of that edge are to be contained in the sub-hyperplanes. This necessary condition can be met by recursively choosing an edge hyperplane and partitioning the b-rep by it, as given in Algorithm I below.

Algorithm I

```
Brep_to_Bspt: Brep b -> Bspt T
{
   IF b == NULL
   THEN
      T = a cell
   ELSE
      h = Choose_Edge_Hyperplane( b )
      { b+, b-, b0 } = Partition_Brep( b, h )
      T.faces = b0
      T.pos_subtree = Brep_to_Bspt( b+ )
      T.neg_subtree = Brep_to_Bspt( b- )
   END
}
```

We can employ this algorithm for converting from discrete space if we can generate something comparable to the b-rep representation. We could try to first convert from a discrete representation to a b-rep directly, and then apply algorithm I to generate a tree. But we claim that converting to a b-rep is more difficult than generating a partitioning tree directly. Instead, what we will do is generate a finite set of boundary points lying on the discrete lattice using standard image processing techniques. To these we apply a Hough transform in order to select candidate hyperplanes, and then we "distribute" the boundary points among these hyperplanes. This then is the input to an algorithm using the same schema as algorithm I, but with the b-rep data type replaced by a sub-hyperplane containing a list of boundary points. Figure 2 illustrates the general scenario.



Major steps in the conversion process
Figure 2

Edges are then approximated by boundary points. However, we need not construct edges explicitly from boundary points, but rather we can generate hyperplanes from boundary points. This is a simpler task, since it is not based upon the topological operation of discovering connected components, a process which is sensitive to noise. The only topological operation we use is a local fixed-width neighborhood operation for determining whether a given boundary point should be treated as lying on a given hyperplane, as opposed to some other hyperplane.

The schema illustrated in Figure 2 was first introduced in [Rahda et al 91]. In the work present here, we demonstrate the application of this methodology to medical images as well as providing several improvements. The Hough transform has been augmented with an iterative least-squares fit and a neighborhood operation that together produce higher fidelity hyperplanes, leading to better quality trees. In addition, we have incorporated ideas concerning the generation of good trees based on cost models for the expected case [Naylor 92], which we use in Choose_Edge_Hyperplane() in Algorithm I. Unlike previous partitioning tree construction heuristics, the intrinsic hierarchical nature of trees have been exploited to provide an overtly multi-resolution form. This yields better trees in terms of size, cost of spatial operations, and degree of compression, and it reduces or eliminates the need for the pyramidal based conversion scheme in [Rahda et al 91].

Discrete Space Operations

The first step in the process is the discovery of discontinuities in the geometric set. This takes the form of generating a finite set of boundary points, each located on the lattice upon which the discrete data is defined. Currently, we accomplish this by applying standard image processing techniques used for edge detection in 2D images. The processing pipeline is: initial noise compensation, gradient generation, finding gradients which are local maxima, and separating edge gradients from texture and noise gradients (Figure 3).

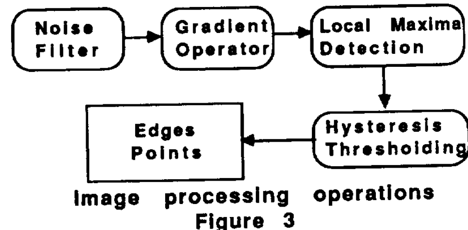


Image processing operations
Figure 3

Various strategies have been developed for compensating for noise in data. Since no stage of the process can eliminate noise induced artifacts, each stage must in fact be noise tolerant. However, a common first step is to apply a smoothing operator using, for example, a Gaussian filter, that does no more than distribute the "energy" due to noise about its local neighborhood. However, being a simple convolution, it also has the undesirable effect of blurring the edges [Torre and Poggio 86], and so we only use it to the ex-

tent that the level of noise demands. Because our general methodology is very noise tolerant, we need a weak filter.

The second step is to apply a gradient operator to the entire data set, resulting in a gradient defined at each lattice point. For an analytically defined function, the gradient is the set of partial derivatives, and so is well defined. But for a discrete space representation, this operation must be approximated, and there are a number of ways of doing so. We have tried the difference operator, the central difference operator, the Sobel operator (see, for example, [Ballard and Brown 82]) and Canny's edge operator [Canny 83]. Each has its advantages and disadvantages, and we have no definitive opinion yet about which to use, but we are currently using Canny's.

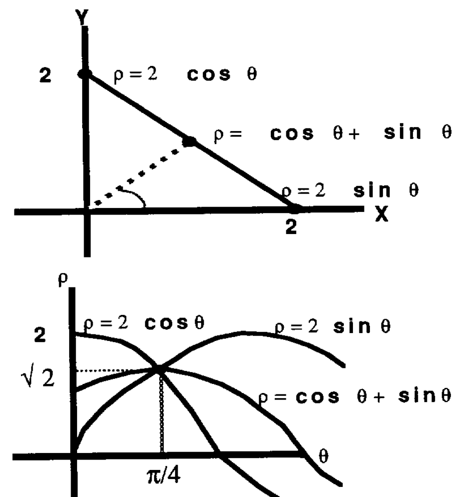
From the discrete representation of the gradient, we must identify those points whose gradient suggests that they lie on edges. A standard technique is to assume that such points have gradients that are local maxima. Since the gradients arise from 1st derivatives, this is of course equivalent to finding the zeros of the second derivative. The operation for doing this is called "non-maximum suppression" (see [Canny 83]). It simply examines the local neighborhood of a gradient to determine whether it is a local maximum or not.

The final step in the image processing pipeline is to separate the remaining local maximum gradients of texture from the edge points. One method for accomplishing this is called *hysteresis thresholding* [Canny 86]. The idea is separate the points by their gradients initially into three groups: accepted edge points, potential edge points, and rejected edge points. Points from the second group will subsequently be accepted if and only if they are connected to some point in the first group. To do this classification, one computes a histogram of the gradients, followed by the selection of two thresholding values for separating the three groups. The lower threshold is typically 80% of the cumulative histogram (integral of the histogram), while the high threshold is typically 2-3 times the low threshold.

Hough transform

We now come to the step that provides the bridge between discrete and continuous space: generating hyperplanes from boundary points. The principal method for this is the Hough Transform, or HT [Hough 62] [Duda and Hart 72] (see [Illingworth and Kittler 88] for a survey containing 136 references). This is a search method using a finite discrete space to represent all hyperplanes that may be incident with boundary points; that is, points in Hough space correspond to hyperplanes in image space. Discretizing and bounding the Hough space means that only a finite number of hyperplanes are considered, which is crucial to the technique. In image space, hyperplanes are commonly represented by the unit normal $\mathbf{n}$ and distance ρ from the origin. The Hough space uses ρ as a parameter/dimension, but $\mathbf{n}$ is represented instead as angles measured between the normal and coordinate axes. In 2D, this is the angle θ, $0 \le \theta < \pi$, measured from the x-axis. The quantization chosen for these coordinates is correlated to the quantization and noise of the image space [Brown 83]. We chose ρ to be approximately the same as the lattice spacing, and we quantize the angles into 1 or 1/2 degree units.

The idea of the HT is to count how many image space points lie on any given image space hyperplane, with the anticipation that hyperplanes with many points are ones containing edges. We could for each possible hyperplane simple go through the list of points and determine coincidence with a dot product. But for any given point, it is known *a priori* that it is not coincident with most hyperplanes. So a less expensive approach is to go in the reverse direction: for each point enumerate all hyperplanes containing the point. If the Hough space was continuous, then this "enumeration" would be equivalent to the following: a single point $\mathbf{x}$ in image space maps to a hypersurface in Hough space; and conversely, this hypersurface contains every Hough space point corresponding to an image-space hyperplane incident with $\mathbf{x}$. Since the Hough space is represented discretely, we will, in effect, scan-convert the hypersurface corresponding to a particular $\mathbf{x}$ by stepping through the angles throughout their entire range and determining ρ as a function of $\mathbf{x}$ and the angles. In 2D, $\rho = x \cos \theta + y \sin \theta)$ (Figure 4). We have extended this process slightly by incorporating a hyperplane width 2ω; that is, we vary the value of ρ over a small interval, $\rho \pm \omega$, where $1< \omega < 2$. The motivation for this is that the discrete space operators generate "fat" edges with width usually > 1. Without a hyperplane width, only a subset of these points would be treated as incident with the face's hyperplane. Thus ρ can have a relatively high resolution to achieve good positioning without boundary points being missed. This also smooths somewhat the quantitization of the Hough space.



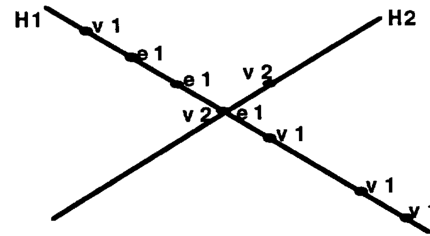
The Hough Transform
Figure 4

Since we want to use the HT to identify hyperplanes containing faces, and we have previously computed a discrete gradient for each boundary point, we can limit the range of angles to lie within some neighborhood of this gradient, as suggested in [Princen et al 89]. However, since this discrete gradient can be rather poor in quality, we found it prudent to use a relatively large neighborhood of almost 45 degrees, instead of the much smaller neighborhood of ~22 degrees advocated in [Princen et all 89] in order to avoid unintentionally excluding a point from being coincident with its "true" hyperplane.

We are able to use this much more conservative gradient culling because we use in addition a *neighborhood operator*. Image processing uses frequency as its primary metaphor and convolution as its primary operator. In geometry, many properties are defined in terms of the structure of the ε–neighborhood surrounding a point. This defines whether a point is in the interior, exterior or on the boundary, and if a boundary point, then whether it lies in the relative interior of a face, edge or vertex (in general, a k-face). When considering incidence between a point **x** and a hyperplane **h**, we first examine the neighborhood of **x** restricted to **h**. This allows us to introduce a degree of topological sensitivity without sacrificing noise tolerance, since drop-outs will have a limited effect, and noise induced isolated points can be detected and eliminated.

If we consider an edge **e** as an open set relative to its hyperplane of support **e.h**, then all boundary points comprising **e** should have a dense ε–neighborhood, for some ε, in **e.h**. Thus points with a sparse neighborhood in some arbitrary **h** could not be edge points, and so should not, for the purposes of edge recognition, be treated as incident with **h**. This prevents boundary points from an edge being considered as incident with a hyperplane that only intersects but does not contain **e**. Also, isolated points due to noise are easily identified and eliminated, and so there is less need for the Gaussian blurring mentioned above as the first step in the image processing pipeline. In addition to eliminating points, we will, for retained points, use the density of their neighborhood as a positive weight when selecting hyperplanes. Consequently, those hyperplanes which have many points with dense neighborhoods will be favored over those with less density.

The neighborhood must of course be approximated in discrete space. We use a width 5 neighborhood and its lattice points are found by scan-converting the neighborhood. A measure, used subsequently to order hyperplanes, is computed as a weighted sum over the neighborhood, similar to convolutions. We currently are using [1 2 4 2 1] as our weights. However, we can also perform certain kinds of pattern recognition. For example, the vertices of an edge, being shared by other edges, constitutes an area in which the quantization seriously degrades the ability to reconstruct the geometry using boundary points within that area. However, the neighborhood operator can often recognize such vertex points, by for example detecting that one or more of the points in the neighborhood are missing (assuming no dropouts). The vertex points can then be excluded from the reconstruction process. Currently we define a vertex point to be one whose neighborhood along a given hyperplane contains less than 3 other boundary points. In figure 5, points labeled with v1 or e1 are respectively vertex and edge points with respect to hyperplane H1, and similarly v2 are vertex points with respect to H2. Note that the classification of boundary point lying at the intersection of H1 and H2 is different for each hyperplane. We can afford to ignore small features since all k-faces, k < d-1, are defined in the partitioning tree by the intersection of hyperplanes, and thus, we only need to recognize edges.



Boundary point classification by neighborhood operator.
Figure 5

The discrete Hough space is represented by a 2-dimensional array, and for each point in Hough space, we maintain a list of the boundary points which are coincident with the corresponding image space hyperplane. We also maintain a measure of "goodness" that is the sum of the measurements produced by the neighborhood operator. While it has been common when using the Hough transform to use the Hough coordinates for the image space hyperplane, we want greater fidelity than these quantized parameters can provide. Consequently, we apply a least squares fit, using floating point to the boundary points which mapped to a single Hough point, to generate an image space hyperplane with floating point coordinates.

After all hyperplanes have been generated, we are now prepared to create the input to the tree construction algorithm in the form of an ordered set of sub-hyperplanes each containing incident boundary points. Initially, the sub-hyperplanes will represent the same set as the hyperplanes, but during tree construction, the sub-hyperplanes will be subjected to partitioning, an operation not defined on hyperplanes (see [Naylor, Amanatides and Thibault 90] for representation and generation of sub-hyperplanes). First, the sub-hyperplanes are ordered by using the measure from the Hough array. This ordering favors sub-hyperplanes with large numbers of boundary points with dense neighborhoods. Then the boundary points are distributed among the sub-hyperplanes by partitioning the entire set of boundary points in the order induced by the measure and associating a boundary point with the first sub-hyperplane with

which it is incident. Any sub-hyperplane containing no boundary points is immediately discarded.

Algorithm I can now be applied to this ordered list of sub-hyperplanes, instead of b-rep edges, where partitioning of a sub-hyperplane entails partitioning its boundary points into negative and positive subsets as well. Any time this partitioning produces a sub-hyperplane with no boundary points, it is discarded.

Attribute generation

One of the advantages of partitioning trees over traditional b-reps is the explicit representation of d-dimensional cells with which we can associate attributes and so represent functions. After the tree is constructed from the boundary points, we must determine the attributes for each cell. To do this, we insert all lattice points into the tree using the standard point classification algorithm which determines the cell in which each point lies. For each cell, its attributes are, currently, the average of the attributes of lattice points lying in that cell. This is implemented in the usual way by keeping a running sum of the values along with the number of points contributing to the sum. After all lattice points have been inserted, the average values for each cell can be computed in a final pass through the tree. This, of course, introduces loss of information. Better "fits" could be obtained with more sophisticated methods; a simple alternative would be to use least-squares fit to produce a linear approximation.

Compression

Compression can be important if one wishes to store a library of images or reduce transmission costs of images. To encode a tree, we first recall that any binary tree can be linearized by a preorder traversal. In this linearized format, one needs to distinguish between internal and leaf nodes; for this, the first bit of each record will do. The only information required at internal nodes are the hyperplane coefficients $[\rho\ \theta]$, and at leaf nodes the value of the image within that cell. Given the resolution of our source images (e.g. 256x256), 9-bits for each coefficient is sufficient. The record for each leaf node would occupy 9-bits, the first bit set to 1 and the remaining 8-bits being the value at that node. Thus, this encoding would require 19 bits for each internal node and 9 bits for each leaf node for an average of 14 bits per node (since the number of leaf nodes = 1 + the number of internal nodes).

Further compression could be obtained by exploiting the tree hierarchy. For each internal node **v** corresponding to a region **r**, we can easily compute the mean value of the image in **r**. This is the weighted sum of the values at the leaf nodes of the subtree rooted at **v**, each weight being the area of a cell (normalized to the area represented by the root of the tree). However, instead of storing this average, we instead store the difference between a node's value and its parent's value. A small exponent can be associated with these delta values, either explicitly or implicitly as a function of the size of the region and/or depth in the tree. Similarly, the hierarchy can be used to compress the hyperplane coefficients; for example, the ρ of a hyperplane **h** can be normalized to the size of the region partitioned by **h**. Altogether, this could reduce the size of the encoding to an <u>estimated</u> 10 bits per node. (We have <u>not</u> yet implemented an encoder.)

A decoded version suitable for interactive display is most easily accomplished using a boundary representation of the image, i.e. a list of colored convex polygons each as a list of vertices. These can be synthesized easily from the basic tree representation simply by classifying an initial polygon corresponding to the image domain. This can be done with an algorithm given in [Thibault and Naylor 87].

Examples

In the color plates, we show pictures of three different data sets all obtained through UNC Chapel Hill. For the first two pairs of pictures, the original discrete data is on the left and the partitioning tree version on the right. Picture 1 knee slice and Picture 2 head slice are courtesy of Siemens Medical Systems. In Picture 3, we have a horizontal cross-section through the nasal passages of a head courtesy of North Carolina Memorial Hospital (due to space limitiations, only the partitioning tree representation is shown).

Finally, we have illustrated our claim of robustness by taking the set of boundary points for the MRI head (#2), and prior to the HT, randomly removing 20% percentage of points. While this injects some visible degradation, we were still able to generate a quite acceptable tree as shown in Picutre 2c.

The estimated compression for these examples is given below.

Picture	Name	resolution	type	~size	bits/pixel
1	knee	256x152	MRI	7120	1.1
2	head	256x256	MRI	6150	0.9
3	brain	256x256	CT	3000	0.5

While these rates are attainable by alternative techniques, such as the Discrete Cosine Transform and/or Vector Quantization, what we have presented here represents fairly early stages in the development of our methodology. For example, we have from the outset intended to exploit the hierarchical nature for multi-resolution representations, but have yet to fully develop this. Assuming this is successful, then our coding should be compared to other multi-resolution methods. Also no encoder/decoder has been implemented so we do not really know what the compression will be. Additionally, our numbers are for 256x256 images. We predict, and have some data to indicate, that tree size will grow linearly with increase in linear resolution. If so, we should get twice the compression rates with 512x512 images. However, one is getting more than compression, since the image is being segmented and represented with a search structure, i.e by a tree. This should be crucial for computer vision applications that attempt some form of automated image analysis.

Comparison to Quadtrees

Our method provides a schema for encoding images in continuous space that is an alternative to quadtrees, which are essentially discrete space entities. There discrete space nature is evident when applying affine transformation. Transforming quadtrees entails transforming the cells as if they were continuous space entities, resampling them at lattice points, and then constructing a new quadtree from the resulting discrete space representation. In contrast, the structure of partitioning trees are unaffected by affine transformations, as they are true continuous space representations. The time required to transform a partitioning tree is typically much less than that required by quadtrees or array representations, and does not require maintaining both an original and a transformed instance in order to avoid accumulation of quantization errors. And as noted earlier, enlarging the image does not blur edges (or alternatively generate huge pixels).

Another difference is the presence in partitioning trees of arbitrarily oriented hyperplanes. This allows one to represent an edge exactly rather than by a discrete approximation, i.e. as a set of pixels (quadtree cells). This leads to partitioning trees that are in general smaller than the corresponding quadtree. Similar arguments can be made when comparing k-d tree based approaches, such as [Subramanian and Fussell 90], to partitioning trees. For example, a quadtree for the three data sets are of approximate size 87k, 77k and 33k respectively, yielding compression factors of 8.3, 7.3 and 3.1 bits/pixel, assuming 1-byte per leaf node and one bit per internal node. However, this is a lossless encoding, which is not the function we have represented with partitioning trees. So if we instead generate the quadtree for exactly the same function, the respective numbers are about 32k, 28k and 8k for compression factors of 3.0, 2.7 and 0.8. Thus partitioning trees appear to be 3 to 1.5 times better on these examples.

References

[Ballard and Brown 82]
Ballard and Brown, **Computer Vision**, Prentice Hall (1982).

[Brown 83]
C.M. Brown, "Inherent Bias and Noise in the Hough Transform", **IEEE Transactions on Pattern Analysis and Machine Intelligence**, vol. 5, pp. 87-116, (1983).

[Canny 83]
J.F. Canny, "Finding Edges and Lines in Images", Artificial Intelligence Lab, MIT Technical Report 720, 1983.

[Canny 86]
J.F. Canny, "A Computational Approach to Edge Detection", **IEEE Transactions on Pattern Analysis and Machine Intelligence**, Vol. 8, pp. 679-698, (November 1986).

[Duda and Hart 72]
R.O. Duda and P.E. Hart, "Use of the Hough Transformation to Detect Lines and Curves in Pictures", **CACM**, Vol. 15, pp. 11-15, (1972).

[Fuchs, Kedem, and Naylor 80]
H. Fuchs, Z. Kedem, and B. Naylor, "On Visible Surface Generation by a Priori Tree Structures", **Computer Graphics**, Vol. 14(3), pp. 124-133, (June 1980).

[Hough 62]
P.V.C. Hough, "Method and Means for Recognizing Complex Patterns", U.S. Patent 3069654 (1962).

[Illington and Kittler 88]
J. Illington and J. Kittler, "A Survey of the Hough Transform" **Computer Vision, Graphics, and Image Processing**, vol. 44, pp. 87-116, (1988).

[Naylor 92]
Bruce F. Naylor, "Constructing Good Partitioning Trees", unpublished manuscript.

[Naylor, Amanatides and Thibault 90]
Bruce F. Naylor, John Amanatides and William C. Thibault, "Merging BSP Trees Yields Polyhedral Set Operations", **Computer Graphics** Vol. 24(4), pp. 115-124, (August 1990).

[Princen et al 89]
J. Princen, H. Yuen, J. Illingworth and J. Kittler, "A Comparison of Hough Transform Methods", *Proc, of IEEE International Conference on Image Processing and its Applications*, pp. 73-77, (July 1989)..

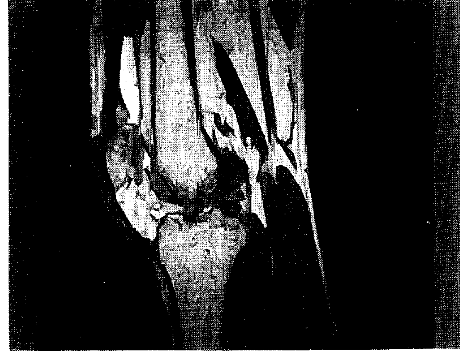
[Radha et al 91]
Hayder Radha, Riccardo Leonardi, Martin Vetterli and Bruce Naylor, "Binary Space Partitioning Tree Representation of Images", **Visual Communications**, Vol. 1, (1991).

[Subramanian and Fussell 90]
K.R. Subramanian and Donald S. Fussell, "Applying Space Subdivision Techniques to Volume Rendering", Proceeding of Visualization '90, (Oct. 1990).

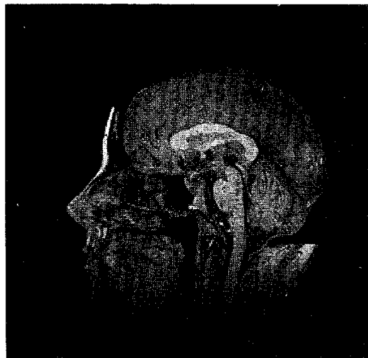
[Thibault and Naylor 87]
W. Thibault and B. Naylor, "Set Operations On Polyhedra Using Binary Space Partitioning Trees", **Computer Graphics** Vol. 21(4), pp. 153-162, (July 1987).

[Torre and Poggio 86]
V. Torre and T. Poggio, "On Edge Detection", **IEEE Transactions on Pattern Analysis and Machine Intelligence**, Vol. 8, pp. 147-163, (February 1986).

Picture 1a: Knee slice.

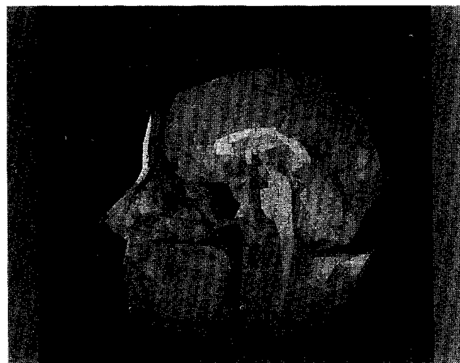


Picture 1b: Knee slice.



Picture 2a: Head slice.

Picture 2b: Head slice.



Picture 2c: Head slice.

Picture 3: Horizontal cross section, nasal passages.

(See color plates, p. CP-18.)