Converting Discrete Images to Partitioning Trees

Kalpathi R. Subramanian and Bruce F. Naylor

Abstract—The discrete space representation of most scientific datasets (pixels, voxels, etc.), generated through instruments or by sampling continuously defined fields, while being simple, is also verbose and structureless. We propose the use of a particular spatial structure, the *binary space partitioning tree*, or, simply, *partitioning tree*, as a new representation to perform efficient geometric computation in discretely defined domains. The ease of performing affine transformations, set operations between objects, and correct implementation of transparency (exploiting the visibility ordering inherent to the representation) makes the partitioning tree a good candidate for probing and analyzing medical reconstructions, in such applications as surgery planning and prostheses design. The multiresolution characteristics of the representation can be exploited to perform such operations at interactive rates by smooth variation of the amount of geometry. Application to ultrasound data segmentation and visualization is proposed.

The paper describes methods for constructing partitioning trees from a discrete image/volume data set. Discrete space operators developed for edge detection are used to locate discontinuities in the image from which lines/planes containing the discontinuities are fitted by using either the Hough transform or a hyperplane sort. A multiresolution representation can be generated by ordering the choice of hyperplanes by the magnitude of the discontinuities. Various approximations can be obtained by pruning the tree according to an error metric. The segmentation of the image into edgeless regions can yield significant data compression. A hierarchical encoding schema for both lossless and lossy encodings is described.

Index Terms—Partitioning trees, BSP trees, space partitioning, miltiresolution representations, image reconstruction, image coding, scientific visualization, MRI visualization.

1 Introduction

Visualization of scientific data arising from simulations or experimental observations has become an important means to analyze, comprehend, and gain insights, given the massive amounts of data that are generated, the critical nature of these applications, and the availability of computer graphics technology to aid the process. A key capability that is required of this process is the ability to interactively *probe* and examine the data under the control of a user. When the data consists of multiple overlapping structures, as in medical imaging reconstructions, the ability to make objects *transparent* provides a way to understand the complex spatial relationships between different materials and the interfaces between them.

While viewing medical reconstructions at interactive speeds is possible with current technology, what is required is the ability to efficiently perform spatial operations on them. We list three potential applications:

Surgery Planning
Computer assisted virtual surgery allows rehearsals of an upcoming surgery. For this, the portion of the anatomy that relates to the surgery is first imaged (using CT, MRI, etc.) and a model reconstructed using visualization algorithms. The system should then permit the surgeon to create (or select) appropriate scalpels or instruments to perform incisions (cutaways) on the model. The resulting cutaways should be displayed in real time, along with quantitative information, such as depth or width of the incision; also useful would be the ability to record a virtual surgery for later playback. Finally, tools should be provided to evaluate each operation as it is performed.

Prostheses Design
In this application, the system should permit a prosthetic device (or implant) of the required shape and size to be designed. The model into which the device is to be inserted should then be built again, through a medical imaging procedure followed by model reconstruction. The system should then allow the designer to interactively fit the implant into the model. Facilities for evaluating the fit should be provided, which could result in modifications to the implant geometry. A more sophisticated system would relate the geometry of the prosthetic device and the socket parameters to perform stress analysis, say, using finite element modeling techniques.

Nondestructive Testing
The application domain here is industrial inspection of machine parts for detection of cracks or fractures. CT is the primary imaging modality, as hard surfaces are easily visible under x-ray radiation. The surfaces of the part can be reconstructed using a variety of algorithms. As in the previous two examples, the ability to interactively explore the surface for defects would significantly enhance the process, especially when there are interior surfaces that are obscured by the outer surfaces.

- *K.R. Subramanian is with the Department of Computer Science, The University of North Carolina at Charlotte, Charlotte, NC 28223. E-mail: krs@mail.cs.uncc.edu.*
- *B.F. Naylor is with Spatial Labs Inc., 371 Finch Lane, Bedminster, NJ 07921. E-mail: naylor@spatial-labs.com*

For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number 105391.

All of these applications involve interaction with a model generated from the discrete data. In each case, the user of the system needs to use a suitably designed tool (scalpel or the implant in the medical applications) against the model. In the first application, the result of using the scalpel is the generation of a cutaway (whose geometry depends on the path of the scalpel), while in the second, the prosthetic device is being used as a tool to determine the geometry of the cutaway. Both of these cases can be handled by performing a *set* operation between the tool and the model geometry. As these operations can occur in the interior sections of the model containing multiple overlapping layers of different materials, making the model transparent is an effective way to aid and clarify such operations in 3D.

Making objects transparent serves two purposes,

1) allow interior features of a model to be revealed without removing the outer layers, and
2) examination of boundaries or interfaces between adjoining materials.

In the context of probing discrete images/volumes, transparency is a very desirable capability, especially when used in conjunction with cutaways. A probe, such as a transparent cube could be used to explore the model (using a mouse or a suitable input device). When the probe is in contact with the model, the space intersected by the probe could be removed and the user presented with a view of the boundaries of the clipped cavity. Interactively controlling the probe (via affine transformations) makes this process all the more intuitive and provides an environment that could be used for research, instructional or clinical purposes for rapid exploration of medical reconstructions.

1.1 Difficulties with Current Representations

The boundary representation (or brep) used in current graphics systems (polygonal patches, for example), while adequate for hardware assisted rendering and interactive viewing, is inefficient for performing spatial operations, as the model is represented simply as a list of primitives/objects. Set operations between two breps is usually a complicated operation, requiring extensive case analysis. Almost all graphics systems use the z-buffer algorithm to compute the visible surface, primarily because the algorithm is simple and can be implemented in hardware. While all primitives will need to be scan-converted regardless of whether they are visible or not, there is no necessity to order them, unlike most other visible surface algorithms. However, this poses a difficulty in correctly rendering transparent objects, which do require objects to be sorted by depth, increasing the rendering complexity to $O(n \ln n)$.

Integrating transparency into the z-buffer algorithm is difficult. The work of Mammen [20] does propose such a method, requiring several additional buffers for storing pixel attributes and multiple passes to render all transparent pixels correctly. Transparency can also be implemented via ray casting, but this method is expensive and nonrealtime, as each new view requires ordering the objects in depth along each ray. Hierarchical data structures such as octrees [16], k-d trees [14] or BSP trees [27] can be used to accelerate this process, but the inability to exploit conventional graphics hardware makes it unattractive to interactive applications. An alternative to this is to use the alpha channel (that stores the opacity at each pixel) available in middle and high end graphics systems and use compositing methods [29]. Here again, objects need to be maintained in depth order, resulting in the same inefficiencies of ray casting. For instance, Carpenter's alpha buffer method maintains polygonal fragments clipped to each pixel in depth order [5].

With the difficulties in supporting transparency, the alternative is to perform cutaways of reconstructions to reveal interior structures. Current visualization systems permit only *planar* cutaways, which are ineffective and very unsatisfactory for applications such as those listed above or those that require a form of x-ray vision for probing the data. Using a simple object such as a cube with transparency is a visually more effective means to understanding 3D structure than planar cutaways of opaque objects. Ideally, what is desirable is a system or representation that can efficiently perform cutaways of arbitrary geometry, detect collisions between objects, support transparency and affine transformations on the model. All of these can be accomplished through *variants of geometric set operations*.

Discrete space structures such as quadtrees and octrees [35], [34] can be used to augment the brep's lack of structure, but affine transformations on such structures require resampling or tree reconstruction after each operation, curtailing performance. For instance, rotating an object represented by an octree by any angle other than a multiple of 90 degrees will require a new octree to be built to represent the object, as the partitioning planes in the transformed tree will no longer be axis-aligned. As described in detail in [34], the procedure requires deriving the boundary of the octree regions (for instance, via chain codes), transforming the polygonal outlines and then constructing the target octree.

An alternative to this is to use voxelization techniques to represent all of the geometry [43], [13]. This permits simple algorithms for performing set operations between objects. The main difficulty is in performing affine transformations, suffering the same disadvantages of quadtrees/octrees. In addition, voxel representations are verbose and have to contend with aliasing artifacts. Operations that require a model search, as is needed in operations such as object picking or collision detection are inefficient and difficult to perform at interactive speeds.

1.2 The Partitioning Tree Approach

A spatial structure that has been in development since the late seventies is the *binary space partitioning tree*, or simply, *partitioning tree* or *bsp tree*. The earliest use of partitioning trees was in computing the visible surface in polyhedral environments [8]. As the tree provided the means to generate view-dependent orderings of the polygons in back to front order, this was advantageous in interactive 3D viewing applications without the need for a z-buffer. Since then partitioning trees have evolved into a representation for solid models [23] and efficient tree algorithms have been developed for performing geometric computation. Three important capabilities make partitioning trees attractive from a computational standpoint:

1) *Affine transformations on objects represented via partitioning trees can be performed with great ease.*

   A partitioning tree can be transformed by any affine transformation M_{aff} by simply transforming its hyperplanes by $M_{aff}^{-1^T}$, the transpose of the inverse of M_{aff} [9]. Besides being fundamental to viewing, this property is crucial for interactive probing, as is the case for the applications stated at the beginning of this section.

2) *Visibility priority orderings can be generated from any viewpoint.*

   Partitioning tree representations allow priority orderings of objects to be generated in $O(n)$ time, as opposed to $O(n \ln n)$ for breps. In addition to the advantage of visible surface determination without the need for a z-buffer, it also allows transparent objects to be rendered correctly in linear time.

3) *Intersections between objects can be calculated efficiently.*

   Both collision detection [24] and geometric set operations [40] are efficiently calculated by exploiting the tree structure; by representing the objects as individual trees, both operations reduce to merging the two object trees [26]. In general, each view is generated by dynamically merging transformed instances of the object trees prior to rendering. Picking is implemented by casting a ray into the environment and determining the first object intersection, which can be performed in logarithmic time. Clipping also reduces to an intersection operation, between the view volume and the object environment.

Partitioning trees with hyperplanes in their interior nodes and continous functions in their cells (to be described in detail in Section 2) are a *sufficient* representation, not requiring the explicit representation of the boundary. All of the computation is performed within a *single unified representation*, hence, it is sufficient to perform a single representation conversion at the outset, and all spatial operations are efficiently performed within the tree representation. These features of partitioning trees make it a good candidate for probing discrete images and volumes. Hence our motivation to perform a representation conversion from a discrete set (images and volumes) to a partitioning tree.

In this article, we focus on the problem of converting from a discrete space representation to a partitioning tree. This process proceeds by discovering the inherent structure in the image, yielding a type of segmentation into regions containing no significant discontinuities, i.e., that contain only texture. The segmentation provides the opportunity for compression by using more compact representations of the texture. Compression is important whenever large amounts of data need to be transmitted or archived. The tree representation can be used to generate lossless or lossy encodings of the discrete data, depending on the application (for instance, broadcast TV would use lossy encodings for real time transmission, while medical applications would require lossless encodings).

The tree representation can also be a significant aid in certain recognition problems. Matching could be facilitated in applications where texture is not needed (inspection of parts on an assembly line, for example) and only the object boundaries are relevant. Ease of performing affine transformations assists in generation of shapes or templates for such applications, while set operations can be used to recognize the shape of the objects. In this case, the tree can use a highly compressed representation of the image texture, or even ignore it. Further, the global nature of our conversion scheme makes the tree representation highly tolerant to noise; boundaries can be extracted with high confidence even in the presence of considerable noise (refer to Section 5.3), that can result in the introduction of false edge points and / or removal of small features. This allows boundaries to remain sharp without the necessity of excessive smoothing. This can be exploited in segmenting clinical ultrasound images; these tend to be low contrast noisy images, possessing a variety of artifacts. Traditional segmentation methods, such as region or edge based methods and clustering techniques do not work well with ultrasound images [33].

In addition, our method of constructing the partitioning tree provides a kind of multiresolution representation in which various levels are selected by pruning the tree according to an error metric. This can be exploited for interactive viewing by using the simpler coarse level representations for selecting a view quickly, followed by a more lengthy but higher quality rendering at a higher resolution (using scan-conversion or ray-tracing). This is somewhat analogous to the adaptive refinement in [1] except that we are varying the amount of geometry involved rather than the complexity of the shading calculations. Combined with the capability of performing intersections, a "probe" or "microscope" modeled by a simple transparent polyhedron (e.g., a cube) can be employed to interactively select sections of the data for high resolution rendering with transparency, while the remainder is at a coarse resolution and opaque. Such combinations of analytic objects with sampled data requires no additional effort since all sets are represented using a single schema (i.e., partitioning trees), obviating the need for hybrid algorithms such as those in [17].

An earlier example of a representation conversion from discrete to continuous space in 3D is the "marching cubes" schema, pioneered in [28] and applied to 3D medical data in [19], where the target continuous space representation is a variety of boundary representations (in particular, a set of triangles specified by vertices). The differences between this and our approach include,

1) our method detects discontinuities, not just contours, and hence represents the entire set; subsets of the function (for instance, contours or 2D slices from 3D volumes) can be generated by only using the relevant boundary points.
2) ours is inherently multiresolution,

A schema much more similar to ours is that of [10], in which a tetrahedral decomposition is used that admits a visibility ordering, and each tetrahedron (analogous to our convex cells) is scan-converted so as to incorporate proper integration

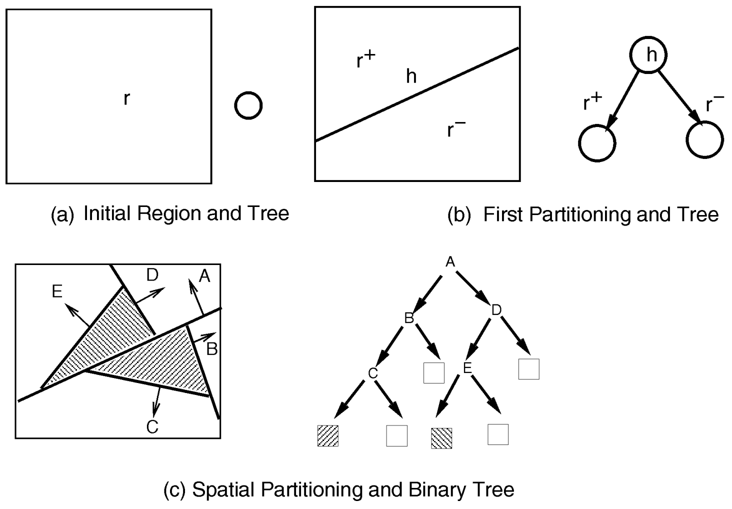

r
r+
h
r−
h
r+
r−
(a) Initial Region and Tree
(b) First Partitioning and Tree
E
D
A
B
C
A
B
D
C
E
(c) Spatial Partitioning and Binary Tree

Fig. 1. Partitioning tree construction.

of transmittance. This approach, as compared to ours, lacks multiresolution rendering as well as set operations.

Partitioning tree schemas have, in fact, been used to represent volume data, although only the axis-aligned variety. In [37], volume rendering via ray-tracing is accelerated by first using a k-d tree to partition the data. Bintrees [17], octrees [42], [15] and pyramidal structures [18], [6] have also been used. However, the generality of partitioning trees permits creating a more adaptive representation when compared to discrete space structures such as octrees; since partitioning trees have no restrictions on hyperplane orientation, discontinuity representation is usually more accurate (a polygon is exactly represented, not discretized). This feature is essential for constructing a multiresolution representation; applying general affine transformations to rotate and translate objects precludes restrictions to axis-aligned hyperplanes.

The schema we will describe was first introduced in [32] and refined in [39] for 2D images. In the work presented here, we extend the methodology to 3D volumetric data (with a new scheme for hyperplane generation) and introduce multiresolution tree pruning using least-squares fit linear approximations of the image within a region of the tree. We also demonstrate for the first time the use of such trees in interactive environments.

2 Partitioning Trees

Binary space partitioning trees [8] are defined via a generating algorithm, and for this only one operation is required: binary partitioning of a region by a hyperplane in a *d*-dimensional continuous space, $d > 0$. Fig. 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, $r^+ = r \cap h^+$ and $r^- = r \cap h^-$, where h^+ and h^- are the positive and negative open halfspaces of *h* respectively. Hence, $r = r^+ \cup r^- \cup r^0 = (r \cap h^+) \cup (r \cap h^-) \cup (r \cap h)$. Also generated is a $(d - 1)$-dimensional region $r^0 = r \cap h$, called a *subhyperplane* (abbr. as *shp*, represented by the bold lines in Fig. 1c). 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 subhyperplanes form a partitioning of the initial region. In Fig. 1, the cells are labeled with numbers and the subhyperplanes with letters. If the initial region is a convex and open set, then all regions of the tree are also convex and open.

Partitioning trees can represent functions whose domain and range are continuous spaces of finite dimensions d_1 and d_2 respectively: $f: X \in S^{d_1} \Rightarrow Y \in S^{d_2}$. The partitioning tree partitions the domain into a hierarchical collection of subdomains. Within each subdomain, a function f_i, which is typically value-continuous, defines the value of f within that subdomain (typically, f_i is defined for all of S^{d_1} as well, although this is not essential). Points in S^{d_1} at which f is value-discontinuous are contained within partitioning hyperplanes. The represented function can be evaluated at any point x by following the path in the tree to the cell c_i that contains the point [22] and evaluating $f_i(x)$. This path following is just the standard method of inserting a point into a search tree, and is commonly called point classification.

3 The Conversion Algorithm

To convert from a discrete space representation of a function to a partitioning tree representation, we need to find points in the domain at which the function is value-discontinuous and then “absorb” them into partitioning hyperplanes. In the context of discrete sets (2D images or 3D volumes), this constitutes segmentation of the image into regions containing no significant discontinuities, but only texture. The first part of the conversion algorithm will determine the points representing discontinuities and associate them with partitioning hyperplanes (lines in 2D and planes in 3D). These hyperplane candidates (along with their point sets) will then be input to a tree construction algorithm. In [40], an algorithm was given, adapted from [8], for converting from the boundary representation of a polytope to a partitioning tree representation. We will use a

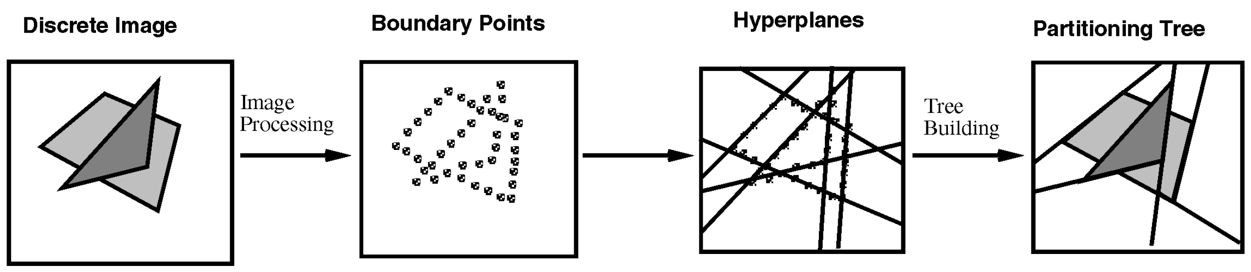


Fig. 2. The conversion process.

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 that all significant discontinuities lie on the subhyperplanes of the tree. Therefore, the hyperplane of any facet (edge in 2D or face in 3D) must be among the set of partitioning hyperplanes if the boundary points of that facet are to be contained in the subhyperplanes. This necessary condition can be met by recursively choosing a partitioning hyperplane and partitioning the candidate hyperplane's point set, as illustrated in Algorithm 1.

```
HpCandidateSet←DiscreteSet_to_HpCandidateSet
                              (DiscreteSet dset,Operator dset_op)
{
    PointSet        boundary_pts;
    Bspt            T;
    HpCandidateSet
                    cand_set;

    boundary_pts =
        GetPoints_DiscreteSet (
            NonMaxSuppress_DiscreteSet (
                Gradient_DiscreteSet (
                    Smooth_DiscreteSet (dset, dset_op), dset_op)),
                                                        dset_op);

    cand_set = Get_HpCandidateSet (boundary_pts, dset);
    T = HpCandidateSet_to_Bspt (cand_set);
}

Bspt←HpCandidateSet_to_Bspt (HpCandidateSet cand_set)
{
    if cand_set == NULL
        T = a cell;
    else
    {
        hp = ChooseCandidate_Hyperplane (cand_set);
        {pos_cand_set, neg_cand_set, on_cand_set} =
                          Partition_HpCandidateSet (cand_set, hp);
        T.faces = on_cand_set;
        T.pos_subtree = HpCandidate
        Set_to_Bspt(pos_cand_set);
        T.neg_subtree = HpCandidate
        Set_to_Bspt(neg_cand_set);
    }
}
```

Algorithm 1: DiscreteSet_to_Bspt: DiscreteSet dset →Bspt T

Since our input domain is a discrete image, we will need to generate something equivalent to a b-rep; using standard image processing operators, we accomplish this by determining the set of points that correspond to discontinuities in the image. These boundary points are then "distributed" among a set of candidate hyperplanes. Two different schemes have been explored to perform this operation,

1) application of the Hough transform and,
2) sorting the gradient vectors of each discrete point and grouping hyperplanes close to each other (gradient vector at each lattice point coupled with the point's location defines a hyperplane).

Once a set of hyperplanes have been generated, Algorithm 1 can be employed to perform the recursive partitioning with the difference that point-sets are partitioned across each hyperplane in contrast to a b-rep. The final step is to calculate attributes (for instance, material color) for each tree cell. Fig. 2 illustrates the general scenario.

3.1 Discontinuity Detection

The first step in the process is the discovery of discontinuities. For this, we identify a subset of the lattice points to be treated as boundary points. This is accomplished by applying standard image processing techniques used for edge detection in 2D images. The processing pipeline is: noise compensation, generating gradients, determining those gradients which are local maxima, and separating edge gradients from texture and noise gradients, as illustrated in Fig. 3.

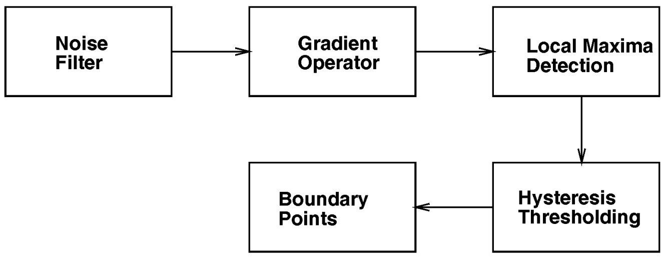


Fig. 3. Image processing operations.

Various strategies have been developed for compensating for noise in data. A common first step is to apply a smoothing operator using a Gaussian filter (with a std. deviation = 1 or 0.75 as needed). However, being a convolution, it also has the undesirable effect of blurring the edges [41], and so we only use it to the extent that the level of noise demands (currently user defined). Because our general methodology is relatively noise tolerant, we can use a sharper filter.

The second step is to produce a gradient at each lattice point by application of a gradient operator to the entire data set. For this we use the Canny edge operator [3], which is a separable operator (i.e., dimension independent, applicable to each dimension separately and in any order). Its form is:

$$\underset{X}{\begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}} \qquad \underset{Y}{\begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix}}$$

Fig. 4. The canny edge detector.

We must now identify those points whose gradient suggests they lie on edges. A standard technique for this is to assume that such points have gradients that are local maxima. Since the gradients arise from first derivatives, this is equivalent to finding the "zero-crossings" of the second derivative, and is called "nonmaximum suppression" [3]. For a given lattice point with gradient $\vec{g}$, two gradient magnitude values are calculated along the direction of $\vec{g}$ by applying linear interpolation to the gradient magnitudes at the neighboring lattice points, as illustrated in Fig. 5. If both of these magnitudes are lesser than the magnitude of $\vec{g}$, then we have identified a zero-crossing, and so a local maximum.

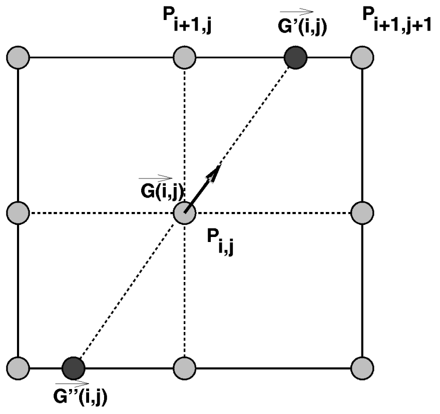
The final step in the image processing pipeline is to separate the local maximum gradients of edge points from texture points. One method of accomplishing this is hysteresis thresholding [4]. This separates the points by their gradient magnitudes into three groups (as illustrated in Fig. 5): strong/accepted edge points, potential edge points, and rejected edge points. Points from the second group will subsequently be accepted if and only if they are topologically connected to some point in the first group. This requires the user to specify two thresholds, interpreted as a percent of the cumulative histogram of the gradient magnitudes, to distinguish the three groups. To do this, one first computes the histogram of the gradients, then the cumulative histogram (integral of the histogram), followed by the conversion of the percent values into actual gradient magnitudes. We can now begin the search process by first marking every lattice point as unvisited. Then the process goes through the set of lattice points, initiating a depth first search at any unvisited point whose gradient magnitude is above the high threshold. Any point visited by this is marked, so as to avoid redundant computation. The thresholds must be selected for each data set by a user driven iterative process; the high threshold is typically 70 percent of the cumulative histogram while the low threshold is around 50 percent. We provide interactive display of the boundary points to facilitate selecting thresholds.

As mentioned earlier, a popular way to study 3D medical datasets is through the visualization of constant density surfaces called isosurfaces. We accommodate this in our representation by generating only the boundary points that correspond to the contour surface. The major difference here is we only represent a subset of the function. Boundary points on the contour surface are determined by performing an intersection of the contour surface with each voxel (similar to the marching cubes algorithm) using linear interpolation to estimate the intersection points. A gradient vector is then computed for each boundary point, again by linear interpolation from the gradients at the two lattice points on the edge containing the boundary point. However, we do not generate polygonal elements at each voxel; instead, the boundary points are directly converted into candidate hyperplanes, as described in the following section.

For generating boundary points corresponding to isosurfaces, the only operations that need to be performed on the volumetric data is smoothing to compensate for noise, followed by the computation of the gradient at the lattice points.

3.2 Generating Candidate Hyperplanes

We now come to the step that provides the bridge between discrete and continuous space: generating hyperplanes from boundary points. We have experimented with two different approaches, (1) using the Hough transform, and



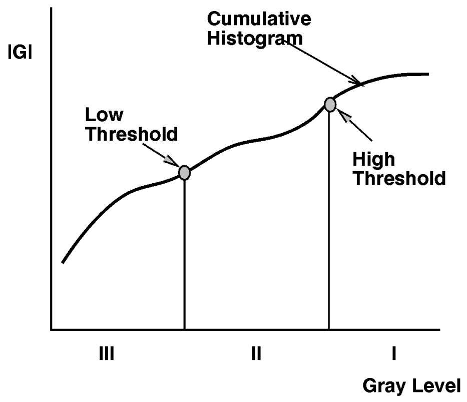


Fig. 5. Nonmaximum suppression.

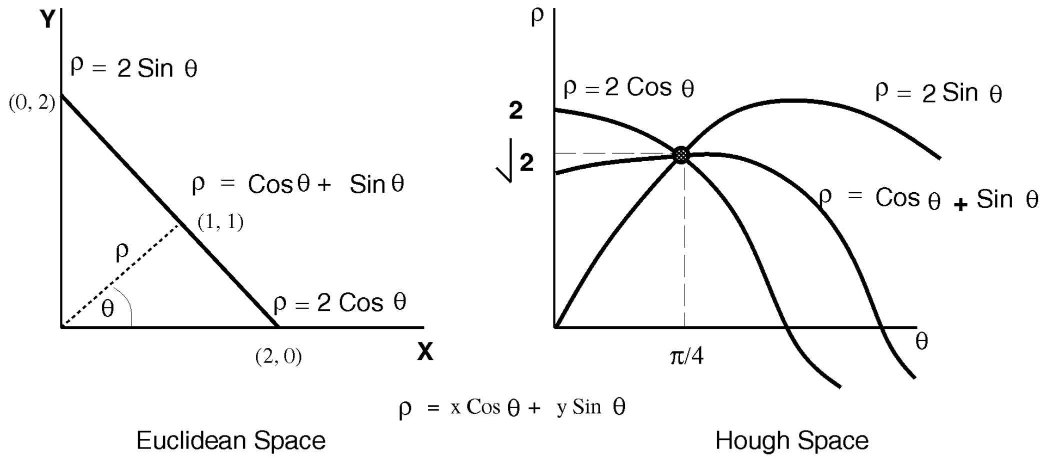


Fig. 6. The Hough transform.

(2) by sorting the boundary points on their gradient vector coefficients, followed by a clustering procedure.

3.2.1 Hough Transform

The Hough Transform, or HT [11], [7], [12] is a search method that uses 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 a unit normal $\vec{n}$ and distance ρ from the origin. The Hough space uses ρ as a parameter/dimension, but $\vec{n}$ is represented instead as angles measured between the normal and coordinate axes. For 3D, there are two angles ϕ, $0 \le \phi < 2\pi$, and θ, $0 \le \theta < \pi$.

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/faces. We could for each possible hyperplane simply go through the list of points and determine coincidence with a dot product. But for any given point $\vec{P}$, 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 $\vec{P}$.

Since the Hough space is represented discretely, we will, in effect, scan-convert the hypersurface corresponding to $\vec{P}$ by stepping through the angles throughout their entire range and determining ρ as a function of $\vec{P}$ and the angles (in 2D, $\rho = x\cos\theta + y\sin\theta$, where (x, y) are the coordinates of $\vec{P}$, as shown in Fig. 6). We have chosen empirically to have ρ be approximately the same as the lattice spacing, and we quantize the angles into 1/2 degree units. Since we want to use the HT to identify hyperplanes containing facets and we have previously computed a discrete gradient for each boundary point, we can improve this process dramatically by limiting the range of angles to lie within a small neighborhood of this gradient, as suggested in [30]. For 2D, this range can be somewhat generous, say as much as ± 10 degrees, but for 3D, it is crucial for this range to be much smaller, say around ± 2 degrees.

For a d-dimensional image, we represent the discrete Hough space by a d-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 a neighborhood operator described below. And finally, for each point we maintain a list of Hough cells to which it contributes (whose use is also described below).

The HT is inherently a global operator. To achieve the fidelity in reconstruction that is required, we have found it necessary to introduce some locality into our schema. For this we use a neighborhood operator. When considering incidence between a point x and a hyperplane h, we first examine the neighborhood of x restricted to h. We want to favor points whose neighborhood lying in h is dense with other boundary points. This prevents boundary points from a facet being considered as incident with a hyperplane that intersects but does not contain the facet. Also, isolated points due to noise are easily identified and eliminated. This allows us to introduce a degree of topological sensitivity in a manner that does not sacrifice noise tolerance. In addition, we will use the neighborhood density as a positive weight for ordering candidate hyperplanes. Those hyperplanes, which have many points with dense neighborhoods, will be favored over those with less density. This will be important in constructing a multiresolution representation.

The neighborhood must be approximated discretely. We use in 2D a neighborhood of five pixels, and similarly in 3D, a 5×5 neighborhood. The larger the neighborhood the more accurately can we determine whether a boundary point is part of some facet lying on the hyperplane being considered; yet, too large a neighborhood will cause points on features smaller than the neighborhood to be rejected. The lattice points corresponding to the neighborhood lying on the hyperplane are found by "scan-converting" the neighborhood. A measure, used subsequently to order hyperplanes, is computed as a weighted sum over the neighborhood. We currently are using as our weights w = [4 3 0 3 4] to generate a radially symmetric filter. This value is then scaled by the gradient magnitude of the boundary point.

To create the set of candidate hyperplanes to be used during tree construction, we must go through the entire Hough table and extract hyperplanes which have sufficiently large measures; that is, we must look for peaks in the measure defined over the Hough space. Now, in the vicinity of a peak, there will generally be multiple HT cells (hyperplanes) with large measure, due to the inaccuracy of the HT. However, we usually want only a single hyperplane per peak. To facilitate this, we generate hyperplanes by always "removing" the hyperplane from the Hough table whose measure is currently the greatest, accompanied by also removing from the table the measure of every boundary point lying on the removed hyperplane. This means subtracting a point's measure from every HT cell that contains the point. This is why for each point we keep a list of HT cells containing the point, so that this measure-removal process can proceed quickly. The removing of measure not only has the effect of setting to zero the measure of the removed hyperplane, but it also significantly reduces the likelihood of a single peak generating multiple hyperplanes. In addition, rather than going through the entire Hough table for each hyperplane we generate, we first sort by measure all hyperplanes / HT cells whose measure lies above a user specified threshold. When we remove measure from the table, this list is resorted in order to reflect the changes due to the removal. This is done using an insertion sort, since this is very efficient for nearly sorted lists.

The Hough transform can be extended to discover hyperplanes in 3D in a straightforward manner. In 3D, the HT is computed as follows:

$$\rho = x \operatorname{Sin}\theta \operatorname{Sin}\phi + y \operatorname{Cos}\phi + z \operatorname{Cos}\theta \operatorname{Sin}\phi \qquad (1)$$

Similar to the 2D transform, ρ is the distance from the hyperplane to the origin, and ϕ and θ are the angles that orient the hyperplane. Discretizing all three parameters for large discrete sets entails the use of a significant amount of space, and the associated computation to locate the boundary points in the HT buckets. Reducing the size of the table makes the representation coarse, resulting in poor quality reconstructions. Discretization resolution for all three parameters of the Hough Transform have to be predetermined, making the process even more prone to aliasing. Since our scheme maintains lists of hyperplanes with each boundary point and lists of boundary points with each candidate hyperplane (primarily for efficiency), the storage requirements to perform the representation conversion becomes impractical for large 3D volumes.

3.2.2 Hyperplane Sort

To overcome some of these problems in using the Hough transform in 3D, we have started using a scheme that simply sorts the boundary points using the coefficients of the hyperplane associated with each point. Note that the numerical gradient computed at each boundary point and the location of the point can be used to determine a hyperplane (the gradient estimates the hyperplane normal) equation for each point. Two hyperplanes, h_1 and h_2 with normals $\vec{n}_1$ and $\vec{n}_2$ and distances d_1 and d_2 are considered to be coincident if the angle between their normals is within a user specified threshold, $\Delta\theta$, and the distance between them is within a specified threshold ΔD, which can be tested as follows:

$$\left(\left|\vec{n}_1 \bullet \vec{n}_2\right| > \Delta\theta\right)\ AND\ \left|d_1 - d_2\right| < \Delta D \qquad (2)$$

In order to determine points coincident with edges / faces, the boundary points are successively sorted four times, once using each of the three components of the gradient vector as the sorting key and once more using the hyperplane distance. This ensures that all collinear points on each face are properly gathered and reduces the introduction of any bias due to the order in which the hyperplane coefficients are used to determine coincidence between pairs of candidates. A merge sort is used in our implementation, with candidate merging performed using the coincidence check illustrated above. The merge process is recursive; a previously merged candidate hyperplane can be merged repeatedly with its neighbors if it satisfies the closeness criterion.

While the HT scheme performs a bucket sort on the boundary points, the hyperplane sort uses a traditional sorting algorithm ($O(n)$ vs. $O(n \log n)$), we have found the latter scheme to be more practical and simpler for large 3D datasets. Also, the scheme requires fewer user defined constants in the hyperplane generation procedure. In 3D, only the angle resolution and the hyperplane width ($\Delta\theta$ and ΔD) need to be specified prior to the hyperplane sort. Most important of all, it enables us to perform representation conversions of datasets of the order of $256 \times 256 \times 256$, which was not possible with the HT method (given the same computing resources).

3.3 Tree Construction

After all hyperplanes have been generated, each having a list of incident boundary points, we are now ready to construct the tree by applying Algorithm 1. Each candidate hyperplane is partitioned across a chosen partitioning hyperplane by having its boundary points subdivided into negative and positive subsets. Any time this partitioning produces a hyperplane with no boundary points, it is discarded. This is likely since the choice of the partitioning hyperplane is governed by a cost model (described below) which might result in a partitioner placing all of the boundary points on one side of the hyperplane.

Now, since every ordering of the hyperplanes will lead to a different tree, but one representing the same image, the question arises as to which trees are better. Constructing good trees is like many optimization problems, it is too hard to solve exactly, and so heuristics must be employed. We have described our current ideas on this subject in [25]. The key concept is that a good tree is one that represents the data by an ordered set of approximations. To see how a single tree might accomplish this, first observe that each path in a tree from the root to a cell corresponds to a nested sequence of regions which "converges" to the cell. Pruning the path at various points yields a region that "approximates" the cell. Now, for an entire tree, we can for each node v compute an approximation of the function represented by the subtree rooted at v (we use a linear approximation, described in the following section) as well as a measure of the error using some appropriate metric (we use squared error) and store that approximation and measured error at *v*. Then, given an error threshold, the tree can be pruned by turning each subtree whose measured error is below the threshold into a cell whose attributes are an approximation of those in the subtree. Since the tree is finite, this can produce only a finite number of approximations. However, a continuum of approximations can be created by interpolating between the approximation at what is currently a cell of the pruned tree and the approximation of its parent region. For this, we use linear interpolation with an interpolation parameter defined as:

$$t = \frac{E_{global} - E_{cell}}{E_{parent} - E_{cell}} \tag{3}$$

where E_{global} is the current global error threshold E_{cell} and E_{parent} are the errors of the cell and its parent.

We now face the question of how to generate such multiresolution trees. One important technique is provided by the HT measure which assigned greater measure to hyperplanes with a large number of densely spaced points and which have large gradients (for the hyperplane sort scheme, we use the sum of the gradient magnitudes of the points that are associated with the hyperplane). Since the hyperplanes were removed from the Hough table in order of decreasing measure, we could build the tree simply using this ordering. However, we can improve upon this by instead considering a number of candidate hyperplanes, say the top 15 or so from our sorted list. To choose among these, we employ a second technique which associates a quantitative interpretation of "goodness" with low expected cost. Our thesis is that representing a function by a set of approximations will yield low expected cost behavior for various spatial operations. Thus, if tree construction attempts to minimize expected cost, it will tend to produce better multiresolution trees.

To compute the expected cost for a particular operation for a given tree *T*, we can, in effect, insert some geometric entity *x*, treated as a random variable, into the tree. To do this we need, as always, to know how to "partition" *x* at an internal region *r*, and in this case this means we need to know the probability of *x* lying in r^+ and r^-. If we assign a unit cost to the partitioning operation then we have:

$$E_{\cos t}(T) = \begin{cases} 0, & \text{T is a cell} \\ 1 + p^- * E_{cost}(T^-) + p^+ * E_{cost}(T^+), & \text{otherwise} \end{cases} \tag{4}$$

This formula as stated does not directly express any dependency upon a particular operation; those characteristics are encoded in the two probabilities p^- and p^+. Now consider point classification. Then, *x* is a random variable chosen from a uniform distribution over some initial region *R* which is partitioned by *T*. For any internal region *r*, we have

$$p^+ = vol(r^+) \big/ vol(r) \tag{5}$$

$$p^- = vol(r^-) \big/ vol(r) \tag{6}$$

where *vol*(*r*) is the *d*-volume of *r*.

3.4 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. Since we have segmented the image into relatively homogeneous regions, we can approximate the original data lying within a cell by a low degree polynomial; in particular, we have chosen to use constant and linear functions to define these attributes (depending upon the variance). Constant functions correspond to the mean value within a cell, while linear functions are generated using least-squares fit.

After the tree is constructed from the boundary points, we determine the attributes for each cell by inserting all lattice points into the tree using the standard point classification algorithm. If a constant function is used to represent the attributes of a cell, then a running sum of material color (*r*, *g*, *b*, α) is maintained at each cell. During classification, if a point is found to be incident with the hyperplane, then its opacity α is halved (each point's opacity begins with an initial value of 1.0 at the root node) and the point is inserted into both sides of the tree. When all lattice points have been inserted, then the mean intensity is calculated at each cell by dividing each material attribute by the summed α at that node.

When the attributes at each cell is to be represented by a linear function, then a least squares fit of the material attributes within each cell is performed. For this, we maintain, at each cell, two matrices of coefficients, (1) $\vec{M}_{mom}$, the moments of the lattice points, and, (2) $\vec{M}_{col}$, the coefficients of the material color, as shown below:

$$\vec{M}_{mom} = \begin{bmatrix} \sum_{i=1}^{n} x_i^2 & \sum_{i=1}^{n} x_i y_i & \sum_{i=1}^{n} x_i z_i & \sum_{i=1}^{n} x_i \\ \sum_{i=1}^{n} x_i y_i & \sum_{i=1}^{n} y_i^2 & \sum_{i=1}^{n} y_i z_i & \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n} x_i z_i & \sum_{i=1}^{n} y_i z_i & \sum_{i=1}^{n} z_i^2 & \sum_{i=1}^{n} z_i \\ \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} y_i & \sum_{i=1}^{n} z_i & \sum_{i=1}^{n} 1 \end{bmatrix}$$

$$\vec{M}_{col} = \begin{bmatrix} \sum_{i=1}^{n} r_i x_i & \sum_{i=1}^{n} g_i x_i & \sum_{i=1}^{n} b_i x_i & \sum_{i=1}^{n} \alpha_i x_i \\ \sum_{i=1}^{n} r_i y_i & \sum_{i=1}^{n} g_i y_i & \sum_{i=1}^{n} b_i y_i & \sum_{i=1}^{n} \alpha_i y_i \\ \sum_{i=1}^{n} r_i z_i & \sum_{i=1}^{n} g_i z_i & \sum_{i=1}^{n} b_i z_i & \sum_{i=1}^{n} \alpha_i z_i \\ \sum_{i=1}^{n} r_i & \sum_{i=1}^{n} g_i & \sum_{i=1}^{n} b_i & \sum_{i=1}^{n} \alpha_i \end{bmatrix} \tag{7}$$

where (x_i, y_i, z_i, 1.0) is the *i*th lattice point within the cell and (r_i, g_i, b_i, α_i) is the material color associated with the point. *n* is the number of lattice points that are interior to the cell (which will be different for each cell) and used in constructing the linear functions. The above system is then solved to determine the linear functions for each of the four material attributes. The red function, for instance, is determined as follows:

$$\vec{M}_{mom}\vec{X}_r = \vec{R} \tag{8}$$

where X_r is $[A_r\ B_r\ C_r\ D_r]^T$, the coefficients of the linear function for the red primary, and

$$\vec{R} = \left[\sum_{i=1}^{n} r_i x_i \sum_{i=1}^{n} r_i y_i \sum_{i=1}^{n} r_i z_i \sum_{i=1}^{n} r_i\right]^T.$$

In a similar fashion, the remaining attribute functions can be obtained. A 2D example is illustrated in Fig. 7. *X* and *Y* are the spatial dimensions, while *r* represents the red component of the material color. The polygon shows a subset of the plane approximates the red attribute over the region of interest. The shaded circles are the intersection of the points with the plane that approximates the attribute function.

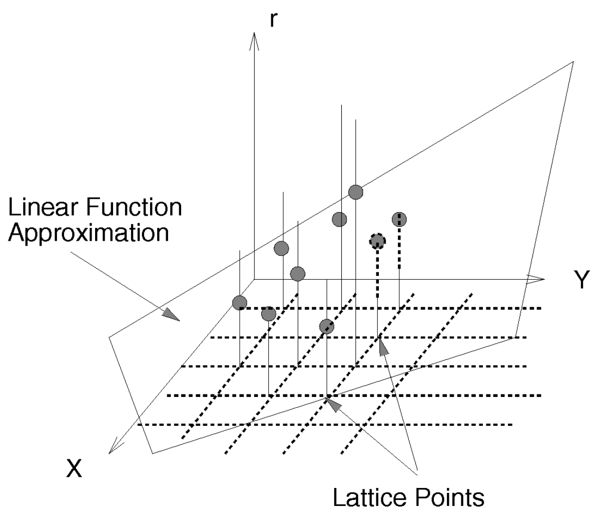


Fig. 7. Linear attribute calculation—2D example.

3.5 Rendering

To render the geometry represented by the partitioning tree, the boundary of the surface is first determined from the partitioning tree and the material attributes, (r, g, b, α), are evaluated at the vertices of each polygon, followed by color interpolation for the interior points. For 2D images, the linear functions are defined over a polygonal domain (x, y, {r, g, b, α}), which can be rendered by linear interpolation. In 3D, the domain of the function is a polyhedron. Rendering involves determining the surface boundary [40](a set of polygons at each node) and evaluating the material color at the vertices, followed by scan conversion.

The above procedure determines the linear attributes at all cells of the partitioning tree. For tree pruning, we also need attributes to be defined at each internal node, as any node could be turned into a cell if its error is below the current error threshold. The least squares coefficients at each internal node (M_{mom} and M_{col}) are obtained simply by adding the corresponding coefficients obtained from its two child nodes. In other words, the parent node's approximation is determined by combining the contributions of the points in its two child nodes. Finally, to render a tree given a particular error threshold, we first determine those nodes of the tree whose error falls below the current error threshold and redefine the subtrees rooted at these nodes as cells of the tree. Next, the cells are rendered by interpolating vertex intensities between each cell and its parent, as per (3). In our implementation, the tree can be pruned interactively by smoothly increasing the error threshold, resulting in a corresponding reduction in the amount of geometry that is rendered.

Attribute calculation reveals another way in which our schema is noise tolerant. First, since we are calculating the attributes from the original data, we can chose to generate as good of an approximation as seems appropriate (linear, quadratic, etc.). Secondly, spurious partitioning hyperplanes generated from noise in the data will have limited effect on the attribute representation.

4 Examples and Results

The representation converter has been implemented on UNIX workstations (Suns and SGIs) and converts 2D and 3D discrete sets to partitioning trees. It works in conjunction with *SCULPT* [24], a solid modeling system based on a partitioning tree representation.

The 2D reconstructions in Figs. 8 and 9 demonstrate the quality that is achievable using this conversion technique. Fig. 8 shows a 2D slice, 256 × 256 pixels, from an MRI dataset of the brain. The left image (Mandrill) in Fig. 9 is of size 512 × 512 pixels, and an example frequently used in graphics and image processing literature. The brain image has been resampled to twice its resolution in each dimension before conversion to a partitioning tree for better reconstruction. A key factor that governs quality is the use of linear color for the tree cells. When compared to the original images, the discontinuities are well represented by the hyperplanes, while the homogeneous regions (represented by the tree cells) exhibit a degree of smoothness. This is more evident in the mandrill example, which is an image with a considerable amount of texture. The converter tends to over-segment the images: The trees in Fig. 8 and 9 have 5,002 and 20,000 internal nodes respectively; pruning these to 2,000 and 10,000 nodes (the leftmost images in Figs. 10 and 11) seems to make very little difference in the quality of the reconstruction. In general, the representation conversion needs to be followed by tree pruning so as to generate not only a good quality reconstruction but also one that is of reasonable size. All of the tree pruning is performed at interactive rates using SCULPT.

The 2D examples (both the brain slice and mandrill) can be generated within about two to five minutes on an Indigo-2 (R4400) workstation. The actual run times do depend on a number of factors, including the size of the final tree, the thresholds used to control the number of hyperplanes used in the tree construction, and the threshold used to limit the number of points that constitute a candidate. Our experience suggests that the tree conversion should be followed by pruning, which provides finer control in generating a good and concise representation. All of the pruning

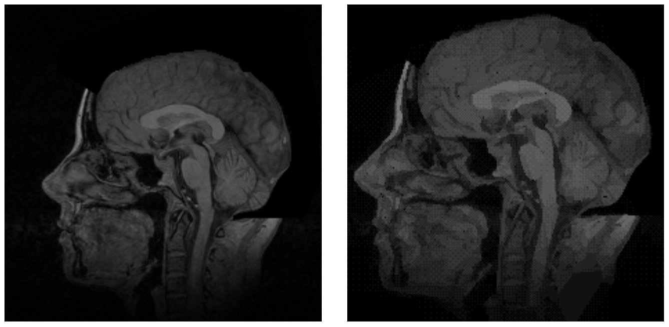


Fig. 8. Brain slice.

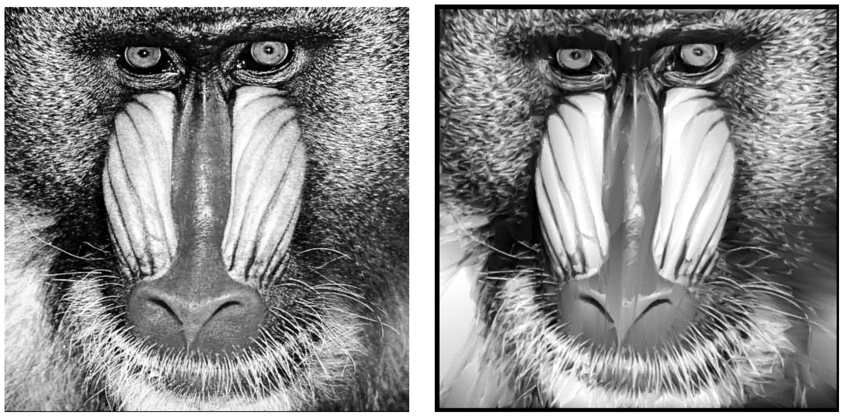

(a) original (b) Tree

Fig. 9. Mandrill.

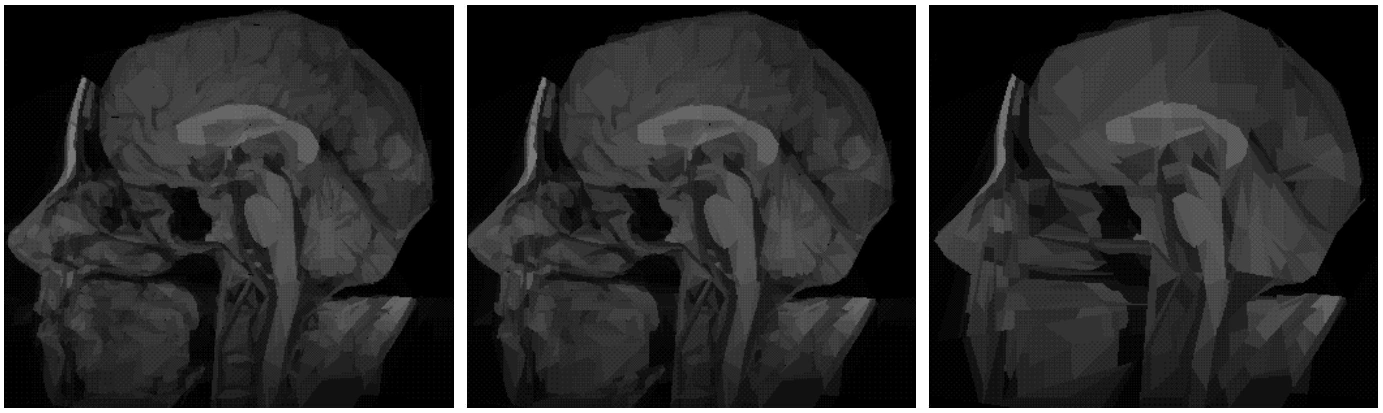

(a) 2,500 nodes (b)1,500 nodes (c) 500 nodes

Fig. 10. Tree size (brain slice).

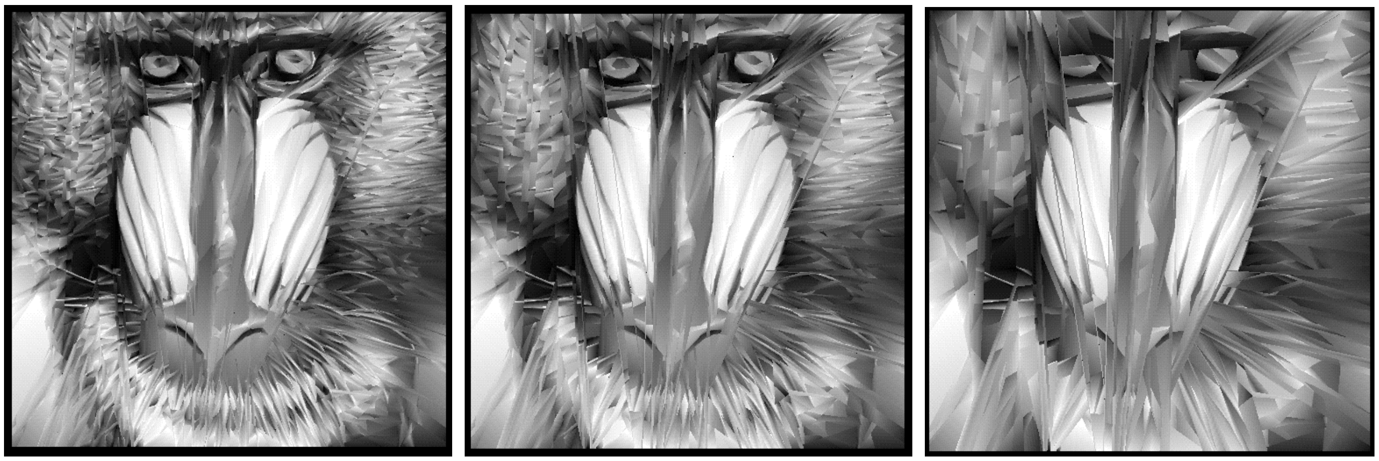

(a) 5,000 nodes (b) 2,500 nodes (c) 1,000 nodes

Fig. 11. Tree size (Mandrill).

is performed at interactive speeds using SCULPT.

Figs. 10 and 11 show the same example images at three different resolutions. Even at the coarsest resolution (500 nodes for the brain slice and 1,000 nodes for the mandrill), the reconstructions are easily recognizable. The biggest advantage of using the coarse resolution trees is the ability to manipulate the images at interactive rates (which will be even more critical in 3D).

The images in Fig. 12 illustrate the tolerance of the representation to noise that can corrupt the data. Noise in images can introduce spurious points or remove points that represent a discontinuity. This could manifest itself in images through weak or missing edges or the loss of small features. In Fig. 12, a subset of the points that represent discontinuities have been selected at random and discarded. The four images (from left to right) represent reconstructions

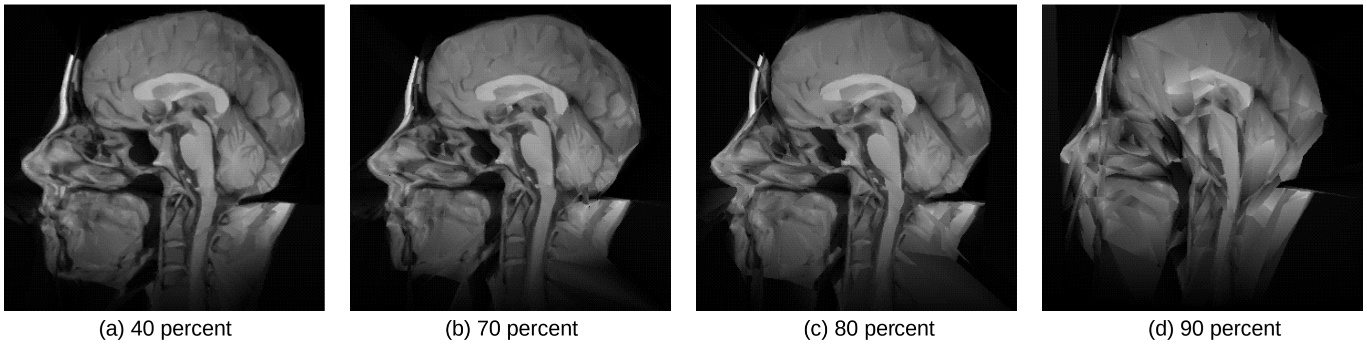


Fig. 12. Boundary point decimation.

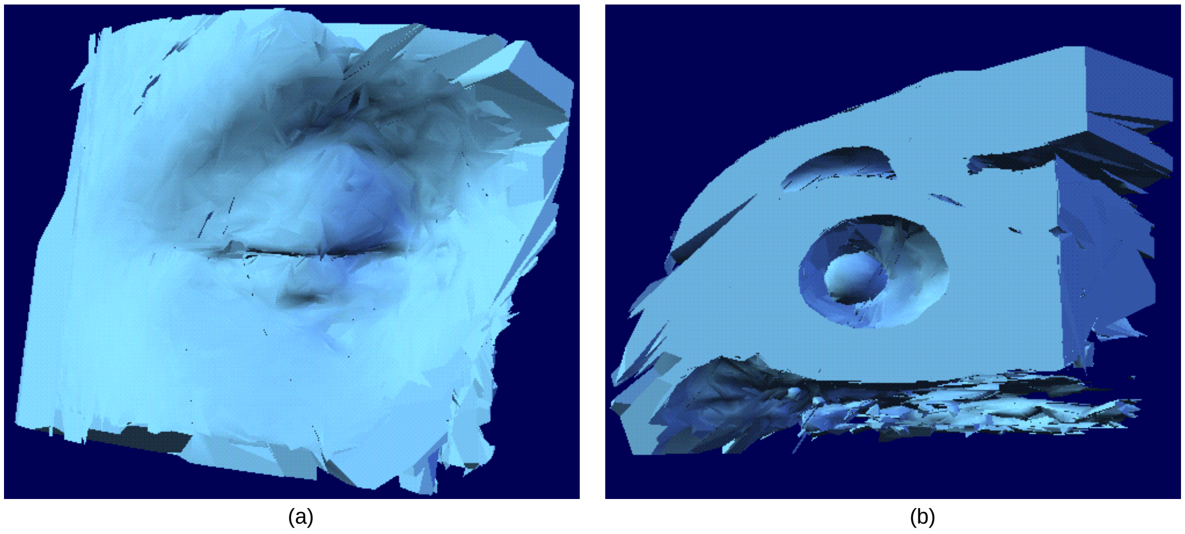


Fig. 13. Eye socket.

performed after discarding 40, 70, 80, and 90 percent of the points that represented the discontinuities in the image. Only the last image shows appreciable degradation in reconstruction quality. Two reasons could account for the high quality of the reconstructions despite the absence of a large fraction of the point set:

1) Our method of constructing hyperplanes is a global technique, and does not require all points on an edge to be present to generate the hyperplane; the neighborhood operator is the only local operation that looks at the density of the neighborhood.
2) It is likely that the discontinuity detection step generates points in excess of what is needed to perform a good reconstruction. However reducing the points at the outset tends to have more of a dramatic effect on reconstruction quality. Its usually safer to generate more points than what is needed (incurring a small penalty in computation) and then prune the constructed tree to reduce the size of the representation, if the image appears to be over-segmented.

Figs. 13 and 14 show two examples of the converter applied to 3D discrete sets. For interactive visualization, isosurfaces are a common way to view medical data; our converter only generates the boundary points corresponding to the isosurface and then converts these points into a partitioning tree. For nonvisualization applications, such as compression or transmission, all of the points representing discontinuities will be input to the converter. In general, we believe that a subset of the function (isosurfaces, arbitrarily oriented 2D slices are examples) are more useful in 3D visualization. Fig. 13 is a reconstruction of a $40 \times 40 \times 40$ subset of an MRI dataset (region around the left eye, at a threshold of 42). Fig. 14 is a $50 \times 50 \times 50$ subset of an engine block dataset. In this example, vertex normals are generated for smooth shading. In 13a, a front view of the left eye is shown, while in 13b, a cutaway of the eye socket is performed (via set intersection with the tree) revealing the socket cavity as well as the cornea. Fig. 14 illustrates reconstruction of the engine block in both opaque and transparent modes. The hyperplane sort scheme was used in both examples to generate hyperplanes from the boundary points.

Finally, Fig. 15 shows the interaction between a transparent cube and the engine block reconstruction. In the left image, the transparent cube acts as a probe to explore different parts of the engine block under user control. Notice that the space intersected between the cube and the engine block is selectively made transparent as the cube is moved through the engine block. In 15a, the cube is just beyond the shaft head, making part of it and a section of its body transparent. In 15b, the volume of the engine block intersected by the cube has been removed (via a set difference operation); the head of the shaft that is visible in the left image now appears clipped in the second image. On an SGI Indigo2 workstation

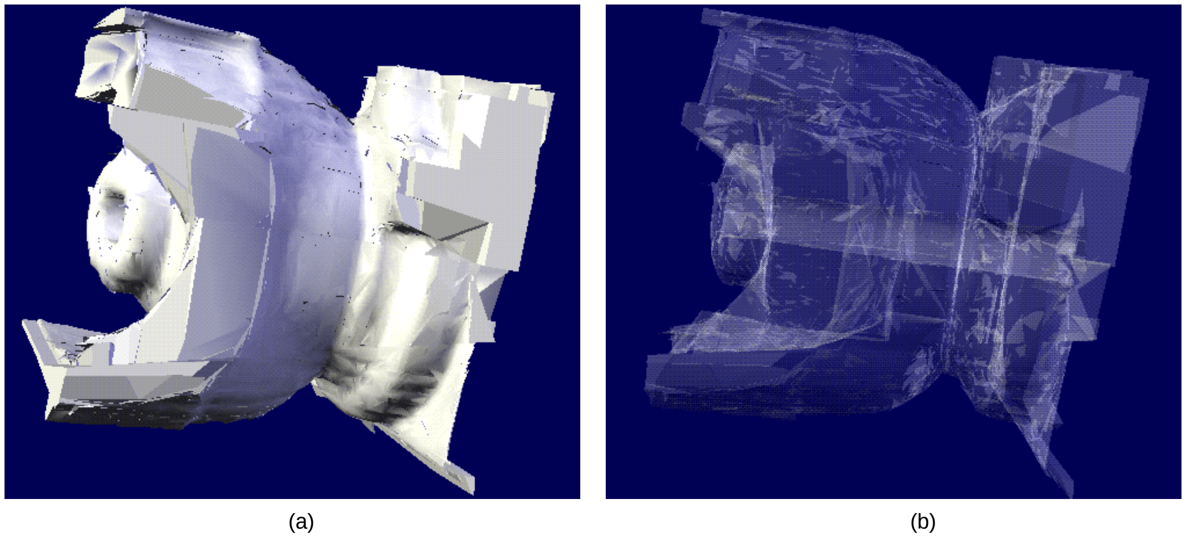


Fig. 14. Engine block.

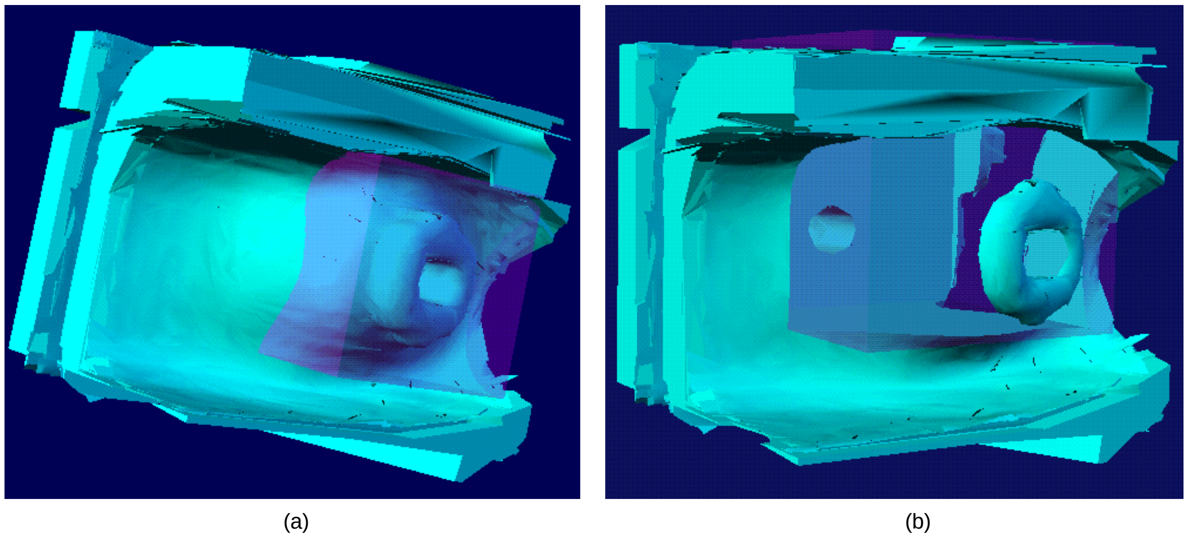


Fig. 15. Interaction between engine block and cube.

with Extreme graphics, the transparent cube can be moved through the engine block model in near real time. In all of these operations, the tree representations of the tool (cube) and the workpiece (engine block) are merged into a single partitioning tree in real time before the view is updated [26].

The 3D reconstructions using our conversion technique are, in general, less detailed when compared to techniques such as the marching cubes method [19]. The primary reason is the difference in the approach to constructing the surface. The global nature of our scheme results in consuming coincident points that span hundreds or thousands of voxels, depending on their orientation. The marching cubes method constructs several triangle elements for every voxel that intersects the isosurface. While our conversion scheme produces surfaces with considerably fewer polygonal facets (the tree representation needs to be converted into polygonal facets prior to rendering, as required of current graphics systems), marching cubes method usually produces very large polygonal models, necessitating the use of decimation algorithms [36].

A second important difference between our scheme and those produced by locally based methods is the degree of smoothness of the generated surface. Local methods have sufficient adjacency information to match faces (common vertices or edges, for instance) across voxels. Global methods, such as ours, have to rely on the discrete approximation of the data to generate accurate hyperplanes that will produce a smooth manifold. The surface that is generated is usually coarse, as evidenced by the transparent engine block in Fig. 14. While the roughness can be reduced at the expense of additional computation, it has been our experience that this requires significant manipulation of parameters that control the reconstruction.

Finally, our conversion schema is targeted at representing the entire function represented by the image/volume, not just a subset. This is evidenced by the 2D examples, where the partitioning tree represents the entire image. This leads to important applications in transmission and compression of image and video, as we will describe later. A large component of 3D visualization of scientific data is through isosurfaces, especially in biomedical applications. This is partly because of the complexity of making sense of the entire set and the difficulties of interacting with the surface containing a large amount of geometry. The partitioning tree representation can ameliorate the problems of spatial interaction (operations are of logarithmic complexity), however, the surface smoothness needs to be improved, which might require some hybrid use of local and global techniques to generate the hyperplanes, or the use of higher order surfaces (cubics, for instance).

5 APPLICATIONS

We discuss three potential applications of using the partitioning tree representation of images and volumes, which we are in the process of pursuing.

5.1 Probing Medical Images

As affine transformations and set operations on objects represented by partitioning trees can be performed efficiently and with great ease, the representation is suited to interacting with medical reconstructions. For instance, clipping a part of an object can be implemented (arbitrarily oriented, not just axis-aligned) by simply performing an intersection operation between the clip volume and the view volume. More generally, one can take a transparent object (shaded differently from the reconstructed volume to make it distinct), such as a cube, and use it as a "probe" to examine the interior of a medical imaging data set. As all objects are represented by partitioning trees, to generate a view, all that needs to be done is to merge the "probe" tree and the 3D image tree [26] (the volume intersected between the probe and image trees is removed, revealing interior features along the boundaries of the cube). As this can be performed at interactive rates (depending on the complexity of the object, if the frame rate gets too slow, tree pruning can be used to improve the rate), this provides a capability to interact with 3D medical images.

As described in the previous section, Fig. 15 illustrates an example of this idea of interactively exploring 3D reconstructions.

5.2 Compression

Compression is important anytime large amounts of data need to be transmitted or archived. Classic examples include broadcast TV (analog), video conferencing/dialtone, fax, etc. In noncritical applications, some loss of data to reduce the transmission time can be tolerated (broadcast TV has few alternatives because of the amount data involved). As our current focus is on medical applications, we are interested in generating lossless encodings, since further degradation of the already discretized images is unacceptable. Another idea that needs to be investigated is how well partitioning trees that have good multiresolution properties compress compared to those that are not. Multiresolution representations are attractive in transmission applications as they permit successive refinement of the image at the receiving end.

To encode a partitioning 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, (ρ, θ) for 2D and (ρ, θ, ϕ) for 3D (all other information can be derived from the tree). The actual number of bits allocated for any given hyperplane can be made dependent upon its place in the hierarchy [31]. In particular, if the region of discrete space partitioned by a specific hyperplane has a maximum linear resolution of x bits, then the hyperplane coefficients need to be encoded only with x bits. However, this requires that these coefficients be defined with respect to a local coordinate system, defined by the smallest axis-aligned bounding box of the region being partitioned. Thus, as one moves further down the tree, fewer bits are needed (as the partitioning regions get successively smaller). Finally, for lossless encoding, as required by medical images, we can encode the residuals; that is, we can encode, within each leaf/cell, the difference between the pixel/voxel values and the linear approximation for that cell. These encoded residuals can then follow in scanline order the encoding of the linear function. The residuals can be variable length encoded using either Huffman or Arithmetic Coding. The variance (or the squared error) for each cell will become part of the encoding, as it determines the number of bits used to encode each residual within that region.

Earlier work on compressing 2D images represented as partitioning trees, proposed in [31], used an optimization scheme to generate partitioning lines, targeted at minimizing the sum of the squared error of the partitioned regions. Results using this scheme on sample images allowed a bit rate anywhere from 0.12 bits/pixel to 0.35 bits/pixel, depending on the mean squared error. However, the method does not take advantage of the hierarchy to optimize the bit allocation and requires an expensive optimization technique to generate the tree.

5.3 Image Segmentation

Segmentation is the process of classifying images into semantically defined objects. It is critical to image understanding and analysis. In medical visualization, they provide an important visual cue for diagnosis and identification of different material or tissue types.

Segmentation is a very difficult problem and to date, no single technique exists that works well for all images [21]. There are a number of methods that can be used to segment images [2]. Three of these methods include edge detection, clustering, and region growing methods.

In Fig. 12, the robustness of the partitioning tree representation was demonstrated for the brain data slice by arbitrarily discarding large fractions of the boundary points and then performing the representation conversion. A potential application of this representation is in segmenting ultrasound images, used widely in OB/GYN, for monitoring the unborn fetus. Ultrasound images, in contrast to CT and MRI, are noisy, of poor contrast and possess a variety of artifacts [33]. Discontinuity detection applied to ultrasound images shows significant loss of boundary points, resulting in weak or missing edges/features. Preliminary work on using traditional segmentation algorithms (region growing methods, for instance) on ultrasound images [38] demonstrate their sensitivity to noise and dropouts of edge points. The tree representation usually can build hyperplanes with only a few of the boundary points (along an edge) present. Further, the gradient measure that was computed and used in the tree construction algorithm can be used to decide if regions neighboring a segmented region belong to the same object. Boundary information such as the gradient measure can make the region growing less sensitive to dropouts in boundary points or spurious edge points, introduced by noise.

Segmentation algorithms will thus operate within the tree representation. A region growing algorithm, for instance, will start from a seed region and recursively merge cells belonging to the same region. The algorithm to perform this is similar to determining the boundary of a polyhedral object represented as a partitioning tree [40]. Such an algorithm, would, in general, be more robust and less sensitive to noise, compared to operating in discrete space.

6 Concluding Remarks

We have presented a scheme that converts 2D and 3D discrete images to a partitioning tree representation. The primary reason to perform this conversion is to exploit the structure of the tree representation to interactively explore, visualize, and quantify objects within such datasets. The partitioning tree representation, being piecewise continuous, facilitates these operations through affine transformations, geometric set operations, and its multiresolution properties. We are currently focused on exploring the applications of the tree representation towards 3D medical visualization, segmentation of fetal ultrasound images, as well as image and video compression.

References

[1] L. Bergman, H. Fuchs, and E. Grant, “Image Rendering by Adaptive Refinement,” *Computer Graphics*, vol. 20, no. 4, pp. 29-37, Aug. 1986.

[2] J.C. Bezdek, L.O. Hall, and L.P. Clark, “Review of MR Segmentation Images Using Pattern Recognition,” *Medical Physics*, vol. 20, no. 4, pp. 1,033-1,048, 1993.

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

[4] J.F. Canny, “A Computational Approach to Edge Detection,” *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 8, pp. 679-698, Nov. 1986.

[5] L. Carpenter, “The A-Buffer, An Antialised Hidden Surface Method,” *Computer Graphics*, vol. 18, no. 3, pp. 103-108, July 1984.

[6] J. Danskin and P. Hanrahan, “Fast Algorithms for Volume Ray Tracing,” *1992 Workshop Volume Visualization*, pp. 91-98, 1992.

[7] R.O. Duda and P.E. Hart. Use of the Hough Transform to Detech Lines and Curves in Pictures,” *Comm. ACM*, vol. 15, pp. 11-15, 1972.

[8] H. Fuchs, Z.M. Kedem, and B.F. Naylor, “On Visible Surface Generation by A Priori Tree Structures,” *Computer Graphics*, vol. 14, no. 3, pp. 124-133, July 1980.

[9] *Graphics Gems*, A.S. Glassner, ed., chapter 10, pp. 539-547. Academic Press, 1990.

[10] P. Hanrahan, “Three-Pass Affine Transforms for Volume Rendering,” *Computer Graphics*, vol. 24, no. 5, Nov. 1990.

[11] P.V.C. Hough, “Method and Means for Recognizing Complex Patterns,” U.S. Patent 306964, 1962.

[12] J. Illingworth and J. Kittler, “A Survey of the Hough Transform,” *Computer Vision, Graphics and Image Processing*, vol. 44, pp. 87-116, 1988.

[13] A. Kaufman, D. Cohen, and R. Yagel, “Volumetric Graphics,” *Computer*, vol. 26, no. 7, pp. 51-64, July 1993.

[14] K.R. Subramanian and D.S. Fussell, “Automatic Termination Criteria for Ray Tracing Hierarchies,” *Proc. Graphics Interface ‘91*, Calgary, Alberta, Oct. 3-7, 1991.

[15] D. Laur and P. Hanrahan, “Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering,” *Computer Graphics*, vol. 25, no. 4, July 1991.

[16] M. Levoy, “Display of Surfaces from Volume Data,” *IEEE Computer Graphics and Applications*, vol. 8, no. 3, May 1988.

[17] M. Levoy, “A Hybrid Ray Tracer for Rendering Polygon and Volume Data,” *IEEE Computer Graphics and Applications*, vol. 10, no. 2, Mar. 1990.

[18] M. Levoy and R. Whitaker, “Gaze Directed Volume Rendering,” *Computer Graphics*, vol. 24, no. 2, Mar. 1990.

[19] W.E. Lorensen and H.E. Cline, “Marching Cubes: A High Resolution 3D Surface Reconstruction Algorithm,” *Computer Graphics*, vol. 21, no. 4, July 1987.

[20] A. Mammen, “Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique,” *IEEE Computer Graphics and Applications*, vol. 9, no. 4, pp. 43-55, July 1989.

[21] H.P. Meinzer, M. Shaffer, G. Glombitza, A. Mayer H. Evers, K. Meetz, and J. Frey, “Segmentation of Medical Images,” *ACM SIGGRAPH ‘94 Course Notes 24, Three-Dimensional Visualization of Medical Data*, pp. 12-27, July 1994.

[22] B.F. Naylor, “A Priori Based Techniques for Determining Visibility Priority for 3-D Scenes,” PhD thesis, The Univ. of Texas at Dallas, May 1981.

[23] B.F. Naylor, “Binary Space Partitioning Trees as an Alternative Representation of Polytopes,” *Computer Aided Design*, vol. 22, no. 4, May 1990.

[24] B.F. Naylor, “Interactive Solid Modeling Using Partitioning Trees,” *Proc. Graphics Interface ‘92*, Vancouver, Canada, May 1992.

[25] B.F. Naylor, “Constructing Good Partitioning Trees,” *Proc. Graphics Interface ‘93*, Toronto, Canada, May 1993.

[26] B.F. Naylor, W. Thibault, and J. Amanatides, “Merging BSP Trees Yields Polyhedral Set Operations,” *Computer Graphics*, vol. 24, no. 4, pp. 115-124, Aug. 1990.

[27] B.F. Naylor and W.C. Thibault, “Application of BSP Trees to Ray Tracing and CSG Evaluation,” Technical Report GIT-ICS-86/03, School of Information and Computer Science, Georgia Inst. of Technology, Feb. 1986.

[28] A. Norton, “Generation and Display of Geometric Fractals in 3D,” *Computer Graphics*, vol. 16, no. 3, pp. 61-67, July 1982.

[29] T. Porter and T. Duff, “Compositing Digital Images,” *Computer Graphics*, vol. 18, no. 3, pp. 253-259, Aug. 1984.

[30] J. Princen, H. Yuen, J. Illingworth, and J. Kittler, “A Comparison of Hough Transform Methods,” *Proc. Int’l Conf. Image Processing and Applications*, pp. 73-77, July 1989.

[31] H.M. Radha, “Efficient Image Representation Using Binary Space Partitioning Trees,” PhD thesis, Columbia Univ., 1992.

[32] H.R. Radha, R. Leonardi, M. Vetterli, and B.F. Naylor, “Efficient Image Representation Using Binary Space Partitioning Trees,” *Visual Communications*, vol. 1, 1991.

[33] G. Sakas, L. Shreyer, and M. Grimm, “Preprocessing, Segmenting and Volume Rendering Ultrasonic Data,” *IEEE Computer Graphics and Applications*, vol. 15, no. 4, July 1995.

[34] H. Samet, *Applications of Spatial Data Structures*. Addison Wesley, 1990.

[35] H. Samet, *The Design and Analysis of Spatial Data Structures*. Addison Wesley, 1990.

[36] W.J. Schroeder, J.A. Zarge, and W.E. Lorensen, “Decimation of Triangle Meshes,” *Computer Graphics*, vol. 26, no. 2, July 1992.

[37] K.R. Subramanian and D.S. Fussell, “Applying Space Subdivision Techniques to Volume Rendering,” *Proc. Visualization ‘90*, San Francisco, Calif, Oct. 23-26, 1990.

[38] K.R. Subramanian, D.M. Lawrence, and M.T. Mostafavi, “Interactive Segmentation and Analysis of Fetal Ultrasound Images,” *Eighth Eurographics Workshop Visualization in Scientific Computing*, Boulogne sur Mer, France, Apr. 1997.

[39] K.R. Subramanian and B.F. Naylor, “Representing Medical Images with Partitioning Trees,” *Proc. Visualization ‘92*, Boston, Mass., Oct. 19-23, 1992.

[40] W.C. Thibault and B.F. Naylor, “Set Operations on Polyhedra Using Binary Space Partitioning Trees,” *Computer Graphics*, vol. 21, no. 4, pp. 153-162, July 1987.
[41] V. Torre and T. Poggio, “On Edge Detection,” *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 8, no. 2, pp. 147-163, Feb. 1986.
[42] J. Wilhelms and A.V. Gelder, “Multidimensional Trees for Controlled Volume Rendering and Compression,” *ACM Symp. Volume Visualization 1994*, Washington D.C., Oct. 1994.
[43] R. Yagel, D. Cohen, and A. Kaufman, “Discrete Ray Tracing,” *IEEE Computer Graphics and Applications*, vol. 12, no. 9, pp. 19-28, Oct. 1992.

Kalpathi Subramanian received a BE (honors) in electronics and communication engineering from the University of Madras in 1983, followed by an MS (1987) and PhD (1990) in computer science at the University of Texas at Austin. Between 1991 and 1993, he was a post-doctoral member of technical staff at AT&T Bell Laboratories in Murray Hill, New Jersey, working under Dr. Bruce Naylor. He has been an assistant professor in the Computer Science Department at the University of North Carolina at Charlotte since 1993. His research interests include the scientific and engineering data visualization, spatial data structures and algorithms, and medical imaging and visualization.

Bruce Naylor received his BA in Philosophy in 1975 from the Unversity of Texas at Austin. He received his MS in 1979 and PhD in 1981 from the University of Texas at Dallas in computer science. He is cofounder and current CEO/CTO of Spatial Labs Inc., Summit, New Jersey, (incorporated 1996). Prior to this, he spent 10 years in research at Bell Labs, Murray Hill, New Jersey. Before joining Bell Labs, Dr. Naylor was on the Computer Science Faculty at Georgia Tech. Dr. Naylor's primary area of reseach has been computational representations of geometry, with binary space partitioning trees being the principal focus of this effort.