Applying Space Subdivision Techniques to Volume Rendering

K. R. Subramanian †‡ Donald S. Fussell ‡

†Center for High Performance Computing
The University of Texas at Austin
10100 Burnet Road, Austin, Tx-78758

‡Department of Computer Sciences
The University of Texas at Austin
Austin, Tx-78712

Abstract

We present a new ray-tracing algorithm for volume rendering which is designed to work efficiently when the data of interest is distributed sparsely through the volume. A simple preprocessing step identifies the voxels representing features of interest. Frequently this set of voxels, arbitrarily distributed in three dimensional space, is a small fraction of the original voxel grid. A *median-cut* space partitioning scheme, combined with bounding volumes to prune void spaces in the resulting search structure, is used to store the voxels of interest in a *k-d* tree. The tree is then efficiently ray-traced to render the voxel data. The *k-d* tree is view independent and can be used for animation sequences involving changes in positions of the viewer or positions of lights. We have applied this search structure to render voxel data from MRI, CAT Scan and electron density distributions.

1 Introduction

An increasingly important application of computer graphics technology is in providing visualization tools to help scientists in a number of fields understand massive amounts of data. Some of these fields include medical imaging, molecular modeling, computational fluid dynamics, seismology, weather models and oceanography. In most of these applications, the data generated are usually too large to interpret directly in their raw form. Also, the data models usually contain a large number of features which are difficult to study all at once. A visual representation of these features, either individually or in some reasonable combination, is desirable for a better understanding of the underlying phenomena.

Many of these data-sets are scalar or vector fields of functions sampled in three spatial dimensions. For example, medical imaging data consists of scalar density values at each vertex of a three dimensional grid. These 'voxel' data are either input directly to a rendering program, in which case the visualization procedure is termed 'volume rendering', or converted to an intermediate representation, for example a surface model, before rendering.

One advantage of creating a surface model from a volumetric representation is that it is often a compact encoding of the characteristic that is being visualized. A surface model, built of polygonal primitives, for instance, can be rendered efficiently with special purpose graphics hardware. The model needs to be created only once and can be viewed from any direction. If the number of surface primitives is not too large, rendering can often be performed in real time, a very useful feature in scientific visualization. A popular method for creating surface models from voxel data is the Marching Cubes method [21].

CH2913-2/90/0000/0150/$01.00 – 1990 IEEE

Authorized licensed use limited to: University of North Carolina at Charlotte. Downloaded on May 27,2023 at 22:42:09 UTC from IEEE Xplore. Restrictions apply.

Most often, creating a surface model involves making a binary classification decision of whether a surface passes through a voxel or not. This can lead to aliasing problems. Also, it may not make sense to create surfaces for certain kinds of volume data. In these cases, alternate solutions involving the direct rendering of volumetric data are preferred. Direct volume rendering techniques are generally based either on a ray-casting approach [7][15][16][17] [5] or on the projection and compositing of preprocessed voxels onto the image plane [5][6].

Ray-casting techniques sample points along the path of each ray (as often as needed) that is cast into the volume. A weighted sum of the contributions of all these points is projected onto the view plane. The images obtained using this method typically have fewer aliasing artifacts than surface modeling methods because no binary classifications need to be made,although the point sampling involved can also be a source of aliasing. Also the grid is sampled at a greater level of detail, thus providing a more accurate picture of the volume data. Similar advantages can be obtained with projection and compositing methods.

Unfortunately, direct volume rendering is generally frequently more time consuming than surface rendering. Such methods, in general, do not take advantage of standard graphics hardware. Though the images produced are of higher quality than those generated from surface models, each image is more expensive to compute. It can be quite expensive to generate animation sequences, which are often a key to understanding scientific phenomena.

Surface modeling approaches to visualizing volumetric data take advantage of the fact that very often the features of interest to a scientist are contained in only a small portion of the original voxel data. By suitably identifying this subset of voxels and representing them as a set of surfaces, considerable savings in computation and space can sometimes be obtained. A surface may be an effective way to visualize data in cases in which a single type of real surface is being identified in the volume data. Frequently, however, either many different surfaces are of interest or the data actually contains no real surfaces. In these situations, direct volume rendering may provide a more useful way for investigators to understand the information of interest in the data.

In this paper, we present a volume rendering algorithm which takes advantage of the lack of interesting information in a large fraction of voxel data in many applications without representing the interesting voxels as surfaces. Our goal is to achieve the benefits of the relatively fast rendering and compact representations of surface models while retaining the ability to effectively represent data which are poorly suited to surface modeling.

Our strategy is to first identify the voxels which do not contain interesting data and remove them, rather than attempting to identify a set of interesting voxels which can constitute a surface. After this initial step, we are left with clumps of interesting voxels distributed throughout the original volume. We incorporate this data into a *k-d tree*, which we build using a ***median-cut*** space partitioning scheme [10][19] with bounding volumes in the interior nodes of the hierarchy. Interesting voxels are recursively partitioned by axis-aligned planes along their medians, resulting in a balanced binary tree. Bounding volumes are computed at nodes of the hierarchy to help in reducing void space created by the partitioning. Ray tracing method is used to render the data contained in the *k-d* tree.

The motivation for applying space subdivision techniques to volumetric rendering comes their success in accelerating the ray tracing of surface models [11][13][18][12][1][8]. Levoy [16] and Meagher [14] have used octrees for rendering volume models. Our own studies of different characteristics of ray tracing hierarchies [20] have shown that for surface models, the ***kd*** tree is a more adaptive and flexible data structure, principally because it combines the advantages of pure space partitioning structures such as the octree and of bounding volume hierarchies.

An important advantage of the $k-d$ tree is its view independence. Animation sequences involving changes in viewer locations or positions of lights require no change in the search structure. Slicing the volume data to look at the internals of a feature

Authorized licensed use limited to: University of North Carolina at Charlotte. Downloaded on May 27,2023 at 22:42:09 UTC from IEEE Xplore. Restrictions apply.

also can use the same search structure with a minor modification in the preprocessing step. We have used this structure to efficiently render animation sequences of a human heart (from an MRI data-set) and scalar fields of electron density distributions.

The remainder of the paper is organized as follows. First, we survey some major existing surface rendering and ray-tracing techniques for volume data in order to identify techniques for identifying and processing interesting voxels which we will also make use of. We then describe some important search structures used to ray-trace surface models. Subsequently, we present a detailed description of our algorithm, and then examine implementation results on the data mentioned above.

2 Visualizing Volumetric Data

Among the different methods of visualizing volumetric data, two techniques are commonly used in medical imaging and molecular modeling applications. The first is the marching cubes method, which outputs polygonal surfaces of a certain density threshold value. The second method is based on ray tracing and directly samples the voxel grid.

The marching cubes method builds triangle models of constant data value surfaces from 3D scalar fields. A threshold density value (or surface constant) is first selected. This value is compared with the density values at the eight corners of each voxel. If it falls within the density range of any of the edges of the voxel, then the surface intersects that edge. The intersection points, determined by linear interpolation from the edge densities define one or more planar surfaces. These surfaces are then triangulated. Before rendering, a unit normal is computed for each triangle vertex. For a constant data value surface, the gradient vector is normal to the surface. The gradient at each vertex (i, j, k) of the grid is computed using central differences. Let this be $\nabla f(\vec{x_i})$. The unit normal is given by

$$N(\vec{x_i}) = \frac{\nabla f(\vec{x_i})}{|\nabla f(\vec{x_i})|}$$

Normals at the vertices of the triangles are calculated by linear interpolation from the corner gradients. Once the normals for the vertices of all the triangles are computed, the triangles can be rendered on any standard graphics workstation.

In the ray tracing approach, rays are cast into the voxel grid, through an imaginary projection plane. Each of these rays is sampled at equal intervals along its path through the grid. This can exploit the use of incremental techniques [9][4]. At each sample location, the voxel density, opacity and local gradient is determined. The voxel density is usually obtained by linear interpolation from the corner density values. The opacity can be a made a function of the density, although this is not necessary. Theoretically, the opacity can be any meaningful function. For instance, if it were a step function peaking at a certain density value, then we end up with iso-surfaces, as in the marching cubes method. A method proposed by Levoy [15] to compute opacity is as follows:

$$\alpha(\vec{x_i}) = \alpha_v * \begin{cases} 1 & \textbf{if } |\nabla f(\vec{x_i})| = 0 \\ & \quad \textbf{and} \\ & f(\vec{x_i}) = f_v \\ t_i & \textbf{if } |\nabla f(\vec{x_i})| > 0 \\ & \quad \textbf{and} \\ & f(\vec{x_i}) - r|\nabla f(\vec{x_i})| \\ & \quad \leq f_v \leq \\ & f(\vec{x_i}) + r|\nabla f(\vec{x_i})| \\ 0 & \textbf{otherwise} \end{cases} \quad (1)$$

where

$$\begin{aligned} \vec{x_i} &= i \text{ th sample location.} \\ f_v &= \text{surface threshold constant.} \\ \alpha_v &= \text{opacity of voxels having density of } f_v. \\ f(\vec{x_i}) &= \text{density at sample } \vec{x_i}. \\ \nabla f(\vec{x_i}) &= \text{local gradient vector.} \\ r &= \text{voxel thickness of the transition region.} \\ \alpha(\vec{x_i}) &= \text{opacity at sample } \vec{x_i}. \\ t_i &= 1 - \frac{1}{r}\left|\frac{f_v - f(\vec{x_i})}{|\nabla f(\vec{x_i})|}\right|. \end{aligned}$$

What the above equation does is make the opacity maximum when the density value is the selected threshold. At nearby density values, the opacity falls

Authorized licensed use limited to: University of North Carolina at Charlotte. Downloaded on May 27,2023 at 22:42:09 UTC from IEEE Xplore. Restrictions apply.

off at a rate inversely proportional to the local gradient vector.

A unit normal is computed as in the marching cubes method. A lighting model is then applied at this sample location to compute a color. A running sum of the accumulated opacity is maintained and used to weight the color of each sample location. Processing terminates when the accumulated opacity reaches unity or there are no more voxels left to process. The sum of all the weighted sample colors is the final color for the ray.

The total intensity for each ray cast into the grid is given by

$$I = \sum_{i=0}^{k-1} I(\vec{x_i})\alpha(\vec{x_i}) \prod_{j=0}^{i-1}(1-\alpha(\vec{x_j})) \qquad (2)$$

where

$$\begin{aligned} \vec{x_i} &= i \text{ th sample.} \\ I(\vec{x_i}) &= \text{Intensity at sample } \vec{x_i}. \\ \alpha(\vec{x_i}) &= \text{opacity at sample } \vec{x_i}. \\ k &= \text{total number of samples along the ray.} \end{aligned}$$

3 Building the k-d Tree

As indicated in the introduction, our goal is to create a data structure which represents interesting volume data in a way that supports fast ray tracing. We do this in two steps.

In the first step, we design a culling function that identifies voxels that can be eliminated from any consideration since they do not contribute to the current view. This step depends on the opacity function used. We demonstrate a culling function to be used when equation 1 computes the opacity. The output of this step is a list of voxels representing the characteristic that will be visualized.

Next we build the *k-d* tree using the median-cut scheme The partitioning is highly flexible in adapting the partitioning planes to the distribution of the voxels in the hierarchy. For early detection of rays that do not intersect any interesting voxels, bounding volumes are stored at nodes of the hierarchy to make the data structure even more compact.

The *k-d* tree will be used to efficiently access the voxels of interest during ray tracing. The remainder of this section is devoted to a detailed description of the building of the data structure. The next section describes its use in ray tracing.

3.1 Identifying 'Relevant' Voxels

The inequalities at the right of equation 1 are the key to determining a culling function for identifying voxels of zero opacity. The culling function is given by

$$\{(f_{max}(vox_{i,j,k}) \quad + \quad r|\nabla f_{max}(vox_{i,j,k})|) < f_v\}$$
$$\textbf{OR}$$
$$\{(f_{min}(vox_{i,j,k}) \quad - \quad r|\nabla f_{max}(vox_{i,j,k})|) > f_v\}$$

where

$$\begin{aligned} f_{min}(vox_{i,j,k}) &= \text{min. voxel density at } (i,j,k). \\ f_{max}(vox_{i,j,k}) &= \text{max. voxel density at } (i,j,k). \\ \nabla f_{max}(vox_{i,j,k}) &= \text{max. voxel gradient at } (i,j,k). \end{aligned}$$

If the above function is **TRUE** for a voxel at (i,j,k), it is discarded; otherwise, it is added to the list of relevant voxels.

Since linear interpolations are used for determining densities as well as gradients within each voxel, the range of densities within each voxel as well as the maximum density gradient of each voxel can be determined. This immediately lets us design a culling function which checks to see if any density in a voxel is within the range of density values that could possibly contribute to the final image. That is exactly what the inequalities in the above function test for. A voxel is irrelevant if its range of densities does not contain the surface threshold density and its density and gradients are such that the opacity function lies completely outside its range. The two parts to the function (on either side of the OR operator) consider the density ranges on either side of f_v, the selected threshold.

Authorized licensed use limited to: University of North Carolina at Charlotte. Downloaded on May 27,2023 at 22:42:09 UTC from IEEE Xplore. Restrictions apply.

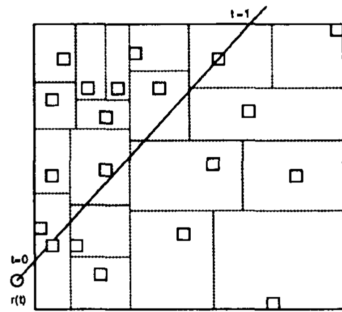


Figure 1: **A median-cut subdivision**

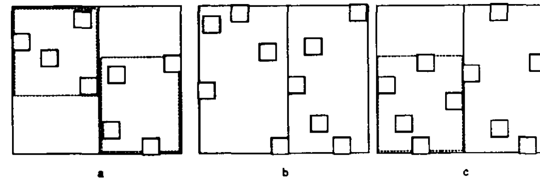


Figure 2: **Using Bounding Volumes**

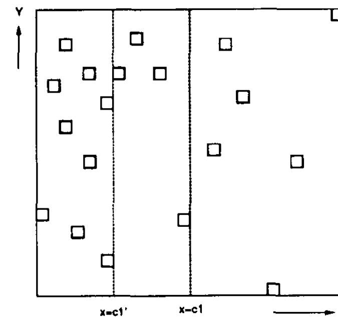


Figure 3: **Optimizing the preprocess**

This process is repeated for each voxel in the grid and a list of 'relevant' voxels is recorded. All other voxels are not of interest until a different feature or characteristic needs to be studied. The output from the preprocessing step is a list of voxels representing the characteristic of interest.

3.2 Building the Median-cut Hierarchy

Having identified the voxels of interest, the next step is to build a hierarchy to store them. The process begins by determining the three dimensional extent of the voxels. This is easily computed from the knowledge of the locations of all the voxels. A binary search is conducted next to determine the plane that best balances the number of voxels on both sides of the partition. This is done in all three dimensions. The criterion or figure of merit (fom) for the plane choice is simply

$$fom = |l_{cnt} - r_{cnt}|$$

where l_{cnt} and r_{cnt} are the voxel counts on either side of the plane. Once the plane is determined, the voxels are partitioned into two subsets, on either side of the plane. These subsets of voxels are recursively partitioned in a similar fashion until each region contains exactly one voxel. An example is shown in Fig. 1 with five levels of partitioning. The dotted lines are partitioning planes and the rectangles represent voxels.

Each time we partition a set of voxels, we need to determine if bounding volumes are required to cull void space. A 2D example is shown in Fig. 2. In 2a, bounding volumes are required on both sides of the plane. In 2b, no bounding volumes are required as this results in no reduction in void space, whereas in 2c only the left side needs it. The reasoning behind using bounding volumes is that a lot of the rays will intersect the void areas without intersecting the surface areas, in which case they can be quickly eliminated from further consideration with a simple bounding volume intersection test.

Some important optimizations in this preprocessing step include the following:

When we are dealing with regular grids (common in medical imaging and molecular modeling applications), partitioning planes need be located only on voxel boundaries, which means the entire search can be performed using integer arithmetic.

Since the partitioning planes are axis-aligned and there are only a finite number of locations for each of them, voxels that fall in each of these locations can be summed up. These partial sums can be used

Authorized licensed use limited to: University of North Carolina at Charlotte. Downloaded on May 27,2023 at 22:42:09 UTC from IEEE Xplore. Restrictions apply.

to determine the best plane choice. Thus, when we need to determine a plane that best balances the voxels in a region, the partial sums are first computed parallel to the orientation of the plane. These sums can be used to determine the best plane as the binary search is conducted. In Fig. 3, vertical columns of voxels are added up when we are determining a plane orthogonal to the horizontal axis. In 3D, if we are searching along the X axis, voxel counts in YZ plane will be summed up at each possible location of the partitioning plane.

During the binary search, when we move the plane towards a location that tends to give a better balance of voxels, the side we are moving away from need not have its voxels examined any further. Thus the total number of voxels in this region just needs to be remembered and taken into consideration when the figure of merit is computed. This is explained with an example in Fig. 3. Here the plane $x = c_1$ results in a partition with more voxels to its left ($l_{cnt} > r_{cnt}$). The next plane choice is at $x = c'_1$, midway between c_1 and the left bound. To determine the number of voxels on either side of $x = c'_1$, its enough to examine the voxels between the left bound and $c1$. The number of voxels to the right of c_1 is r_{cnt} and just needs to be remembered. A similar argument holds if $l_{cnt} < r_{cnt}$.

The median-cut hierarchy built is a binary tree, also called a *k-d* tree [2][3]. Each node in this tree represents a set of voxels, stored in its subtree. One advantage of such a structure is that its height is smaller than an unbalanced tree, making it less expensive to reach the leaf nodes where the voxels are stored. Secondly, our studies have shown that using bounding volumes in the internal nodes of the hierarchy is critical to its success. In this configuration, it is a very compact encoding of the original data for rendering purposes.

4 Ray-tracing the k-d Tree

A ray with arbitrary origin and direction can be traced efficiently using the *k-d* tree. The partitioning planes and the bounding volumes in the hierarchy help determine a set of voxels ordered along the path of the ray. More important, only voxels close to the path of the ray are identified, thus ignoring the bulk of the voxels in the tree. This is done with the help of bounding volume and partitioning plane intersections described as follows.

Given a ray and the root of the tree, the ray is intersected with the bounding volume stored at the root node. If there is an intersection, then we need to determine if this is an internal node or a leaf node. If it is a leaf node, then the voxel at this node needs to be sampled and its contribution to the color of the ray is computed as described in section 2. On the other hand, if it is an internal node, we have two different cases:

1. The ray lies entirely on one side of the partitioning plane.
2. The ray crosses the partitioning plane.

These two cases can be easily determined by intersecting the ray against the partitioning plane and comparing its parametric intersection value with those obtained from intersecting the region's bounding volume. For case 2, the direction of the ray helps define the order in which the two regions need to be processed. It must be pointed out that no coordinates need be computed during this process.

For case 1 we need to search the region containing the ray segment (the ray has been clipped to the bounding volume of this node). For case 2, we need to potentially examine both regions. However the region closer to the ray origin is processed first, since the accumulated opacity might reach unity while examining this region. This would make it unnecessary to examine the farther region. This process is recursively applied, until either the opacity accumulates to unity or there are no more regions (and hence, voxels) to examine. Figure 4 illustrates this. Here $r1(t)$, $r2(t)$ and $r3(t)$ are three different rays. $t1$ and $t2$ are the parametric intersection points with the region, and t, the intersection with the partitioning plane $x = c_1$. The top ray visits only one of the 2 regions, which is recognized by the fact that $t1 < t$ and $t2 < t$. The actual region visited by the ray is determined by the x component of the

Authorized licensed use limited to: University of North Carolina at Charlotte. Downloaded on May 27,2023 at 22:42:09 UTC from IEEE Xplore. Restrictions apply.

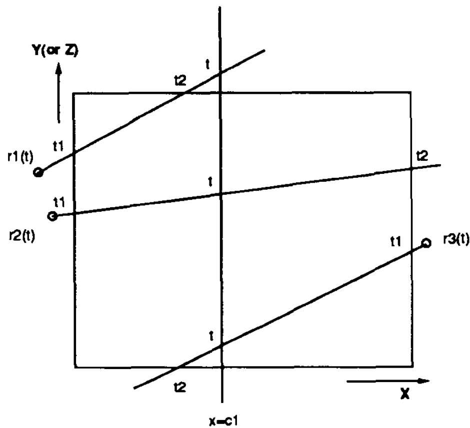


Figure 4: **Ray Traversal.**

Scene	Total Voxels	Relev. Voxels	%Redn.
Heart	2097152	94874	95.50
Heart(sliced)	2097152	51713	97.53
SOD at 80	1091444	63533	94.17
SOD at 100	1091444	27427	97.48
HIPIP	262144	24628	90.61

Scene	Time(min.)			Memory (Mbytes)
	Preprocess		Render	
	Cull	Build		
Heart	2.93	0.49	5.00	4.5
Heart(sliced)	2.96	0.27	5.56	2.5
SOD at 80	1.61	0.34	11.72	3.0
SOD at 100	1.44	0.13	8.13	1.3
HIPIP	0.40	0.12	4.78	1.2

Table 1: **Experimental Results**

ray direction. For the top ray, direction along the x axis is positive, i.e. x increases along the path of the ray, thus identifying region 1. Otherwise, the region 2 will be visited. The middle and bottom rays visit both regions, indicated by $t1 < t$ and $t2 > t$. The x component of the ray direction determines the traversal order. For the middle ray, the direction is positive, so the order is region 1 followed by 2, while it is 2 followed by 1 for the bottom ray because its x direction is negative. A similar strategy holds for identifying the traversal order when the partitioning dimension is y or z.

5 Implementation and Results

We have implemented our algorithm in C on an Ardent Titan 1500 running Ardent UNIX[1] 2.1.1 and tested the implementation on three different test cases. The first case is an MRI data-set of a cadaver heart that has been autopsied; during autopsy, cuts were made into the ventricles. The heart was in a bucket of preservative and was imaged from three orthogonal directions, 24 slices in XY, 20 slices in YZ and 16 slices in XZ plane. The slices were assembled and tri-linearly interpolated to obtain a 128x128x128 data cube. Color plate 1 shows two different views of the heart. Next the heart is sliced vertically (by a plane) and the voxels in front of the slicing plane are removed so as to get a better view of the heart chambers. Plate 2 shows two views after slicing.

The next example is an electron density map of the active site of superoxide dismutase (SOD) enzyme as determined by x-ray crystallography at 1.8 angstrom resolution. The data-set consists of 116 slices, each of size 97x97. Plate 3 shows one frame of an animation sequence when centered around a threshold of 80. At this level, teardrop shapes and clumps of atomic density are seen. At a level of 100, we start seeing individual atoms.

The last example is a quantum mechanical calculation of one electron orbital of a four-iron, eight-sulphur cluster found in many natural proteins. This particular data-set is a high potential iron protein (HIPIP). The data represents the scalar field of the wave-function at each point. The resolution of the data is 64x64x64. Scientists are interested in seeing 'nodal' surfaces, where the data value crosses zero. Plate 4 attempts to demonstrates this.

Table 1 illustrates the reduction in voxel data after the preprocess. In all cases, less than 10% of the total voxels are relevant to the final image. This demonstrates the importance of culling irrelevant voxels and working with only voxels of interest. In

[1] UNIX is a trademark of AT&T Bell Laboratories

Authorized licensed use limited to: University of North Carolina at Charlotte. Downloaded on May 27,2023 at 22:42:09 UTC from IEEE Xplore. Restrictions apply.

the heart images, slicing causes further culling of the data. All timings are for a resolution of 640x480, with one ray cast per pixel. In this implementation, we have not taken advantage of either vectorization or parallelization to optimize performance.

We are presently adapting this technique to be useful for supercomputer users located remote from our center. Our strategy is to preprocess the data, whereby the voxels representing a given characteristic is identified. This set of voxels is then transmitted over the network to the researcher's workstation for rendering. In filtering the voxels of interest from the original three-dimensional grid, a large reduction in the number of voxels transmitted over the network will result a natural compression of the original dataset. Since building the *k-d* tree search structure usually takes a fraction of the rendering time, it can be performed remote from the supercomputer center, thus making it unnecessary to transmit the search structure over the network.

6 Conclusions

We have presented an algorithm which uses space partitioning techniques to support efficient volume rendering. A cull function prunes voxels from the original data that are irrelevant to the characteristic being studied. A *k-d* tree based on median-cut space partitioning with bounding volumes at selected nodes of the hierarchy is used as a data structure for storing the relevant voxels so that they may be accessed efficiently during rendering. This data structure has important advantages for rendering volumetric data:

1. It provides a compact representation of voxels. The partitioning planes in the hierarchy help in determining a traversal order that identifies only voxels close to the path of each ray. The bounding volumes used in the internal nodes of the hierarchy help in identifying rays that do not intersect any relevant voxel by simple bounding volume intersection tests.
2. Since the tree can be traversed along the path of a ray, the ray trace can be terminated once the accumulated opacity reaches unity.
3. The search structure is view independent. With the help of the partitioning planes, a traversal order can be determined for any ray with arbitrary origin and direction. Changes in viewing or lighting parameters require no change in the search structure.
4. The choice and location of the partitioning planes is flexible. The plane that best balances the voxel counts on either side of the partition is chosen. The plane can be aligned with any of the three dimensions.
5. Since partitioning planes need to be located only on voxel boundaries, the entire plane search can be performed using integer arithmetic.

Surface modeling approaches to visualizing volumetric data work best when few surfaces are involved. Existing direct volume rendering approaches exploit the regular nature of three-dimensional grids through the use of incremental techniques for volume traversal. These techniques are most efficient when the voxels of interest are distributed sufficiently densely through the volume. In the data that we have worked with, this has not been the case. Visualizing multiple features increases the number of voxels participating in the view, but there is a point beyond which visualizing multiple characteristics (thereby increasing the number of relevant voxels) only makes it increasingly difficult to interpret the data. Our approach is targeted at producing images of intermediate complexity, i.e. those which may contain more than one type of surface but not so much relevant data that the majority of the volume is involved. The technique should thus be complementary to existing direct volume rendering approaches which work best on very dense data and surface modeling techniques which work best on data with few surfaces of interest.

7 Acknowledgements

The heart data was provided by Dr. Raleigh F. Johnson, Jr. and Dr. Donald G. Brunder, University of Texas Medical Branch, Galveston. The SOD data was due to Duncan McRee, Scripps Clinic, La

Authorized licensed use limited to: University of North Carolina at Charlotte. Downloaded on May 27,2023 at 22:42:09 UTC from IEEE Xplore. Restrictions apply.

Jolla, California. The electron orbital data was provided by Louis Noodleman and David Case, Scripps Clinic, La Jolla, California. Our thanks to the CHPC Visualization Laboratory staff for their help and comments. Our thanks to Don Speray for his useful insights on volume visualization. Lastly, our thanks to Dr. James Almond and Dr. Matthew Witten for making time available to perform this research.

References

[1] James Arvo and David Kirk. Fast ray tracing by ray classification. *Computer Graphics*, 21(4):269–278, July 1987.

[2] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9), September 1975.

[3] Jon Louis Bentley. Data structures for range searching. *Computing Surveys*, 11(4), December 1979.

[4] John G. Cleary and Geoff Wyvill. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *Visual Computer*, 4(2):65–83, July 1988.

[5] C.Upson and M.Keeler. Vbuffer:visible volume rendering. *Computer Graphics*, 22(4), August 1988.

[6] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. *Computer Graphics*, 22(4), August 1988.

[7] D.S.Shlusselberg and W.K.Smith. Three dimensional display of medical image volumes. *NCGA 86 Conf. Proc. NCGA, Fairfax, Virginia*, 1986.

[8] Henry Fuchs, Gregory D. Abram, and Eric D. Grant. Near real-time shaded display of rigid objects. *Computer Graphics*, 17(3):65–72, July 1983.

[9] Akira Fujimoto, Takayuki Tanaka, and Kansei Iwata. Arts: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, 6(4):16–26, April 1986.

[10] Donald Fussell and K. R. Subramanian. Fast ray tracing using k-d trees. Technical Report TR-88-07, Department of Computer Sciences, The University of Texas at Austin, March 1988.

[11] Andrew S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, October 1984.

[12] Jeff Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, pages 14–20, May 1987.

[13] Michael R. Kaplan. The uses of spatial coherence in ray tracing. *ACM SIGGRAPH Course Notes 11*, July 1985.

[14] D. Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19(2):129–147, June 1982.

[15] M.Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3), May 1988.

[16] M.Levoy. Design for a real-time high-quality volume rendering workstation. In *Chapel Hill Workshop on Volume Visualization*, pages 85–92. Computer Science Dept. of the University of North Carolina, 1989.

[17] M.Levoy. A hybrid ray tracer for rendering polygon and volume data. *IEEE Computer Graphics and Applications*, 10(2), March 1990.

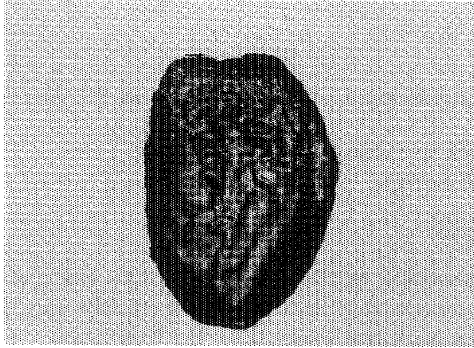
[18] Steven M. Rubin and Turner Whitted. A 3-dimensional representation for fast rendering of complex scenes. *Computer Graphics*, 14(3):110–116, 1980.

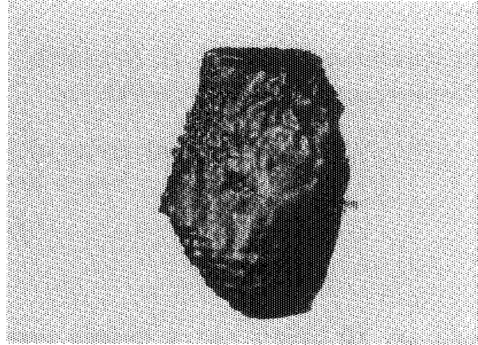
[19] K. R. Subramanian. Fast ray tracing using k-d trees. Master's thesis, Department of Computer Sciences, The University of Texas at Austin, December 1987.

[20] K. R. Subramanian. Factors affecting performance of ray tracing hierarchies. Technical Report TR-90-21, Department of Computer Sciences, The University of Texas at Austin, July 1990.

[21] W.E.Lorensen and H.E.Cline. Marching cubes: A high resolution 3d surface reconstruction algorithm. *Computer Graphics*, 21(4), July 1987.

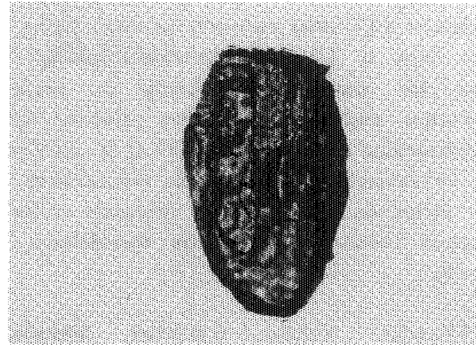
Authorized licensed use limited to: University of North Carolina at Charlotte. Downloaded on May 27,2023 at 22:42:09 UTC from IEEE Xplore. Restrictions apply.





(Color Plate 61, page 470) *(Color Plate 62, page 470)*

Plate 1. Two Views of the Heart.



(Color Plate 63, page 470) *(Color Plate 64, page 470)*

Plate 2. The Heart After Slicing.

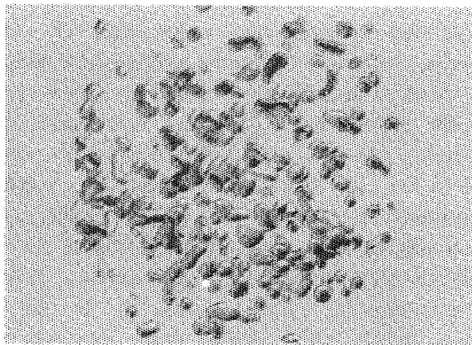


Plate 3. SOD at 80.
(Color Plate 65, page 470)

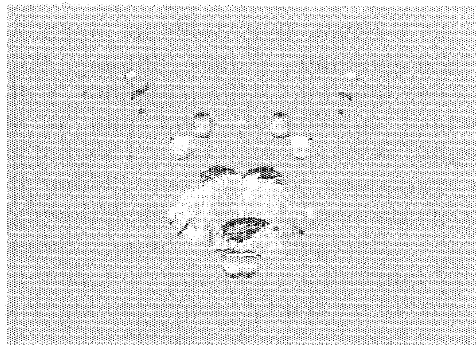


Plate 4. HIPIP Nodal Surfaces.
(Color Plate 66, page 470)

Authorized licensed use limited to: University of North Carolina at Charlotte. Downloaded on May 27,2023 at 22:42:09 UTC from IEEE Xplore. Restrictions apply.