Volume Rendering of DCT-Based Compressed 3D Scalar Data

Boon-Lock Yeo and Bede Liu, *Fellow, IEEE*

***Abstract*—This paper proposes a scheme to perform volume rendering from compressed scalar data. Instead of decompressing the entire data set before rendering, blocks of data are decompressed as needed. Discrete cosine transform based compression technique is used to illustrate the method. The data is partitioned into overlapping blocks to permit local rendering and allow easy parallelization. Compression by factor of 20 to 30 produces rendering virtually indistinguishable from rendering using the original uncompressed data. Speedup is obtained by making use of spatial homogeneity detected in the transform domain. Rendering time using the proposed approach is less than that of direct rendering from the entire uncompressed data. The proposed method thus offers an attractive option to reduce storage, computation, and transmission overhead of otherwise huge data sets.**

***Index Items*—3D discrete cosine transform, 3D DCT, volumetric compression, volume rendering from compressed data.**

I. Introduction

Scientific visualization plays an important role in the study of a variety of problems involving large quantities of multidimensional data. However, storage is often needed. The time needed to transmit the huge amount of data stored at possibly remote sites for visualization and the need for large memory are important considerations for the visualization of such data set.

There is thus the interest to explore the use of compression to reduce the raw data size. Much of the attractiveness would be lost if one has to decompress the entire data first before visualization, especially if the data set is much larger than the memory size.

The effect of volumetric compression on the visualization of medical data is explored in [1]. 3D discrete cosine transform (DCT) is used. Surface rendering and oblique angle slices are studied to evaluate the loss introduced by compression. It is reported that a compression ratio of 6 to 15 does not have any adverse visual effect on the rendered output.

Ning and Hesselink [2] describe the use of vector quantization to code small blocks of 3D data. Code-vectors are used to index into the codebook for direct access of data blocks at rendering time. Separate vector training is needed for each desired compression factor. An increase of 5 percent in the rendering time is reported due to the need to perform code book lookup. Fast rendering can be performed on the predefined codebook [3]. For a given viewing direction, volume shading and image generation are performed on each code vector in the codebook. These block projections can then be reused to obtain the rendered image.

Fourier domain volume rendering is investigated in [4], [5], [6], [7], [8]. The Fourier projection-slice theorem is used to obtain fast projection of the volume data in the Fourier domain. Dunne, et al., [4] and Malzbender [5] consider the rendering of attenuation-only images. The approach is generalized [6], [7] to include certain classes of shading and depth cueing which can be performed in the Fourier domain. For a volume data of size $N \times N \times N$, the complexity of rendering is reduced from $O(N^3)$ to $O(N^2 \log N)$. These methods do not handle occlusions. Chiueh, et al., [8] divide 3D data into subcubes of size $M \times M \times M$ and Fourier transform each subcube for compression. Image rendering is performed in the Fourier domain using the projection-slice theorem on each subcube. Occlusion effects are generated by using spatial composition between the resulting subimages. The rendering complexity of this method is $O(N^3 \log M/M)$.

Muraki [9], [10] describes the use of wavelet transform to obtain a unique hierarchical description of an object. The relation between reconstruction errors and number of representative coefficients is also studied. The method requires the entire 3D data to be reconstructed before rendering operations can be performed.

In this paper, we propose a scheme to perform volume rendering from compressed scalar data. Instead of decompressing the entire data set before rendering, blocks of data are decompressed as needed, permitting efficient use of memory and allowing rendering of data set whose size exceeds the memory. We use a DCT based 3D compression scheme to illustrate the method and its effectiveness. Our method is easily parallelizable. Speedup is obtained by making use of spatial homogeneity detected in the transform domain and approximation can be applied without significant loss of quality of rendered image. The proposed method thus offers an attractive option to reduce storage, computation, and transmission overhead.

Volume rendering is briefly reviewed in Section II. In Section III, we present a 3D DCT based compression scheme. Overview of the proposed method is given in Section IV. Ordering of blocks to permit minimal book-keeping of local results is presented in Section V. Arrangement of blocks to enable local rendering is discussed in Section VI. Accelerated rendering by exploiting transform domain information is discussed in Section VII. Finally, compression and rendering results are presented in Section VIII.

Manuscript received Feb. 7, 1995.

The authors are with Dept. of Electrical Engineering, Princeton University, Princeton, NJ 08544; e-mail yeo, or liu@ee.princeton.edu.

IEEECS Log Number V95004.

II. Volume Rendering

Volume rendering is the direct rendering of data sampled in three dimensions. By modeling each voxel (volume element) as both a source and attenuator of light, volume rendering allows the visualization of interior structure and its relations to exterior forms.

There are two basic approaches to volume renderings: the object space approach and the image space approach. In the object space approach [11], [12], [13], [14], [15], contributions of voxels, cells, or kernels are projected onto the image plane, whereas in the image space approach [16], [17], ray tracing to perform the source-attenuation integral is carried out for each pixel on the image plane. Accelerated methods have been proposed in [18], [19], [20], [21], including fast algorithms that build additional data-structure prior to rendering [18], [20], [21] under the assumption that there is sufficient memory to hold the data-structure.

We will consider the image space approach of ray tracing in this paper. Each voxel is mapped to some predefined color and opacity by *shading* and *classification* [16]. Simple shading and classification may involve only a mapping from the voxel value $x(l, m, n)$. More sophisticated mappings take into account lighting model for shading and which materials is of interest [16], [11], [22]. These mappings of $x(l, m, n)$ to the color c and opacity α involve voxel $x(l, m, n)$ and its gradient $\nabla x(l, m, n)$. A common model for α is

$$\alpha(l, m, n) = \left|\nabla x(l, m, n)\right| g\big(x(l, m, n)\big) \tag{1}$$

where $g()$ could be a piecewise scalar linear function.

In practice, $\nabla x(l, m, n)$ is typically estimated using discrete differences [16], [11], [22]:

$$\nabla x(l, m, n) = \frac{1}{2}\begin{pmatrix} x(l+1, m, n) - x(l-1, m, n) \\ x(l, m+1, n) - x(l, m-1, n) \\ x(l, m, n+1) - x(l, m, n-1) \end{pmatrix}. \tag{2}$$

The projected image is then formed by ray tracing: the act of shooting rays through the volume from a given viewpoint and computing the contribution of each voxel along the ray. The value of the ray is the integral of all the light formed along its path, appropriately scaled by the optical distance from the ray to the projected image. Based on low albedo assumption in the models developed by Blinn [23] and Kajiya and Herzen [24], the intensity of a pixel $\vec{x}$ on the projected image due to a ray at direction $\vec{w}$ can be described by

$$I(\vec{x}, \vec{\omega}) = \int_0^\infty e^{-\int_0^t \sigma(s)ds} c(t)dt, \tag{3}$$

where $c(t)$ is the differential intensity scattered at $\vec{x} + t\vec{\omega}$ and $\sigma(s)$ is the differential attenuation at $\vec{x} + s\vec{\omega}$.

Discretizing (3) leads to the recursions:

$$A_{n+1} = A_n + \alpha_n(1 - A_n) \tag{4}$$

$$I_{n+1} = I_n + c_n(1 - A_n). \tag{5}$$

where c_n and α_n are sampled color and opacity along a ray in the direction $\vec{\omega}$. The above recursions can be evaluated using the *over* notation introduced by Porter and Duff [25] for image composition.

Because the point where samples along a ray is taken need not necessarily be on a voxel, interpolation has to be performed. Typically, tri-linear interpolation is used. Evaluation of (4) and (5) can be done via the pseudocode of Fig. 1.

```
rgba RAYTRACE $(\vec{x}, \vec{\omega})$
1 $c \leftarrow (0,0,0,0)$
2 **for** $i \leftarrow$ first$(\vec{x}, \vec{\omega})$to last$(\vec{x}, \vec{\omega})$ **do**
3     $s \leftarrow$ sample$(\vec{x} + i\vec{\omega})$
4     $c \leftarrow c$ over $s$
5 **return** $c$
```

Fig. 1. Pseudocode for ray tracing.

The functions first() and last() return the index for the first and last voxel on the volume data along the ray in the direction $\vec{\omega}$, while function sample() returns interpolated values of color and opacity.

III. 3D Compression

Our goal is to organize and compress the 3D data set such that local volume rendering can be performed without the need to first decompress the entire data set. This method, in principle, allows one to render arbitrarily large data set.

We choose a block-based approach to allow rendering of *any* arbitrary block. The order of rendering is not dictated by the order in which the blocks are compressed. This permits minimal book-keeping of intermediate results. Section V explains this in greater details. The 3D data set is first divided into overlapping *macro-blocks* of size 32 × 32 × 32, of which only one is kept in memory and processed at a time. Each macro-block is comprised of 8 × 8 × 8 *block*. Each 8 × 8 × 8 block in a macro-block is then compressed using 3D DCT. The details of the overlapping which is needed to allow local volume rendering are described in Section VI.

A. Compression Technique

This section describes how an 8 × 8 × 8 DCT-transformed block is coded.

Our compression scheme extends JPEG's [26] scheme of lossy image compression to 3D. In addition, side information is stored to aid in random access of blocks during rendering time. The terminology used in this section follow those used in [26].

The N-point 1D DCT of $\{x_n\}$ is given by

$$f_i = \sqrt{\frac{2}{N}} \sum_{l=0}^{N-1} k(i) x_l \cos\frac{(2l+1)i\pi}{2N}, \tag{6}$$

where

$$k(i) = \begin{cases} \frac{1}{\sqrt{2}}, & i = 0; \\ 1, & \textit{otherwise}. \end{cases} \tag{7}$$

A 3D DCT is obtained by taking the 1D DCT in succession in each of the coordinate axis. The voxels will be denoted by $\mathbf{X} = \{x_{i,j,k}\}$ and the transform coefficients denoted by $\{f_{i,j,k}\}$. In particular, the DC coefficient is $f_{0,0,0}$ while the rest are the AC coefficients. We shall use $N = 8$ for compression purpose from this point on.

We perform scalar quantization on the 3D DCT coefficients. Each quantized coefficient is given by

$$f^q_{i,j,k} = round\left(\frac{f_{i,j,k}}{s \times q_{i,j,k}}\right). \tag{8}$$

where s is a scaling factor and $q_{i,j,k}$ is the quantization step size.

For simplicity, we choose $q_{i,j,k}$ to be dependent only on the sum $i + j + k = c$. The rate of increase of $q_{i,j,k}$ with c is chosen to match the rate of increase with $i + j = c$ for the default quantization steps $q_{i,j}$ used in JPEG. The values used in this paper, shown in Table I, have been found to work well with data sets in practice.

TABLE I
$q_{i,j,k}$ VALUES

c	0	1	2	3	4	5	6	7	8	9	10
$q_{i,j,k}$	16	12	12	14	17	26	34	55	64	80	85
c	11	12	13	14	15	16	17	18	19	20	21
$q_{i,j,k}$	90	100	100	100	100	100	100	100	100	100	100

A.1. Differential Coding of DC Coefficients

Each DC coefficient is coded as the difference between the current value and the previous one. Since it is desired that the information be kept as local as possible, we store the DCs together and reconstruct all the DC coefficients first. Even though we retain in memory and process one macro-block at a time, we can afford to keep in memory all the DCs because they occupy only a small fraction of the entire data.

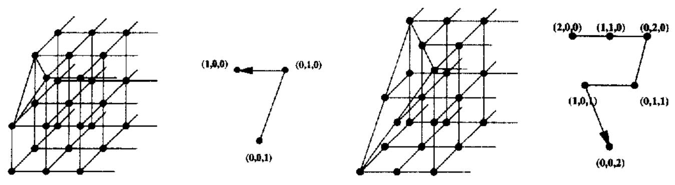


Fig. 2. Planes with $i + j + k = 1$ and $i + j + k = 2$ for zigzag traversal.

A.2. 3D Zigzag Traversal of Quantized AC Coefficients

After quantization, a significant fraction of the higher frequency terms $f^q_{i,j,k}$ would be zeros. A 3D zigzag traversal, a natural extension of the 2D zigzag traversal, also used in [8], of the quantized AC coefficients is performed to remove correlations between AC coefficients. The traversal is such that the planes $i + j + k = c$ are visited in increasing order of c and a 2D zigzagging is performed within each plane. There is a total of 22 planes for $N = 8$. A diagram illustrating two planes is shown in Fig. 2. Such traversal of the quantized coefficients from low-to-high frequency allows efficient packings of the coefficients via run-length encoding and Huffman coding [27].

We shall denote the AC coefficients as ordered in this manner: $f^r_1, f^r_2, \ldots, f^r_{511}$.

A.3. Encoding

From the 3D zigzag traversal of the quantized coefficients, each nonzero term is represented in combination with the run-length, which is the consecutive number of preceding zero-valued terms. Each pair of run-length and coefficient will be referred to as a (*run*, *value*) pair. The last (*run*, *value*) is the (0, 0) pair, which denotes the End of Block, i.e., a consecutive run of zero-valued terms to the end of the block.

The coding of (*run*, *value*) pairs is discussed in detail in [26]. Essentially, each (*run*, *value*) pair is then represented by two pieces of information: (*run*, *size*) and *amplitude*. Each value can be classified into one of the predefined ranges in the form

$$[2^n, \ldots, 2^{n+1} - 1] \cup [-(2^{n+1} - 1), \ldots, -2^n]$$

and *size* indicates which range the value falls into. Thus the range gives the most significant bit of the signed integer encoding. *Amplitude* is the signed integer encoding of the remaining bits of value. Similar technique is used for the differentially coded DC coefficients into *size* and *amplitude*, except that no run-length coding is performed.

A Huffman table is then constructed for the AC's (*run*, *size*) symbols and DC's *size* symbols. In practice, the size of the table is in the order of 0.1 percent of the compressed data. So, the Huffman table can be included with the compressed data with negligible overheads. There is, however, an overhead at compression time because two passes through the data set are required: the first pass is required to construct the Huffman table and the second pass makes use of the table to perform the actual compression.

B. Storage Issues

To further facilitate random access of macro-blocks, we also store the length of the differentially coded DC coefficients and each AC macro-block. Using 32 bits for the length of DC coefficients and 16 bits for each AC macro-block, it is found that the overhead for storing these lengths is in the order of 0.1 percent of the compressed data. For example, with 256 × 256 × 256 data used for experiments in this paper, this overhead is $9^3 \times 2 + 4 = 1462$ bytes.

Before rendering, an index table can be rapidly constructed from the length of DC coefficients and AC macro-blocks. Each entry of this table contains the offset from the beginning of the file to the start of a macro-block. This permits random access to any macro-blocks.

The general format for storing the compressed data set is as follows:

```
< Header>< DC Huffman Table>< AC Huffman Table>
< Length of DC>[··· Coded DCs ···]
< Length of Mblock 1>[··· Coded ACs of Mblock 1 ···]
< Length of Mblock 2>[··· Coded ACs of Mblock 2 ···]
                    •
                    •
                    •
```

IV. Proposed Method of Rendering

Fig. 3 illustrates the steps in volume rendering of data set compressed with the method described in the previous section. There are two stages: 1) preparing of table information for decompression and random access of macro-blocks and 2) local rendering of each macro-block.

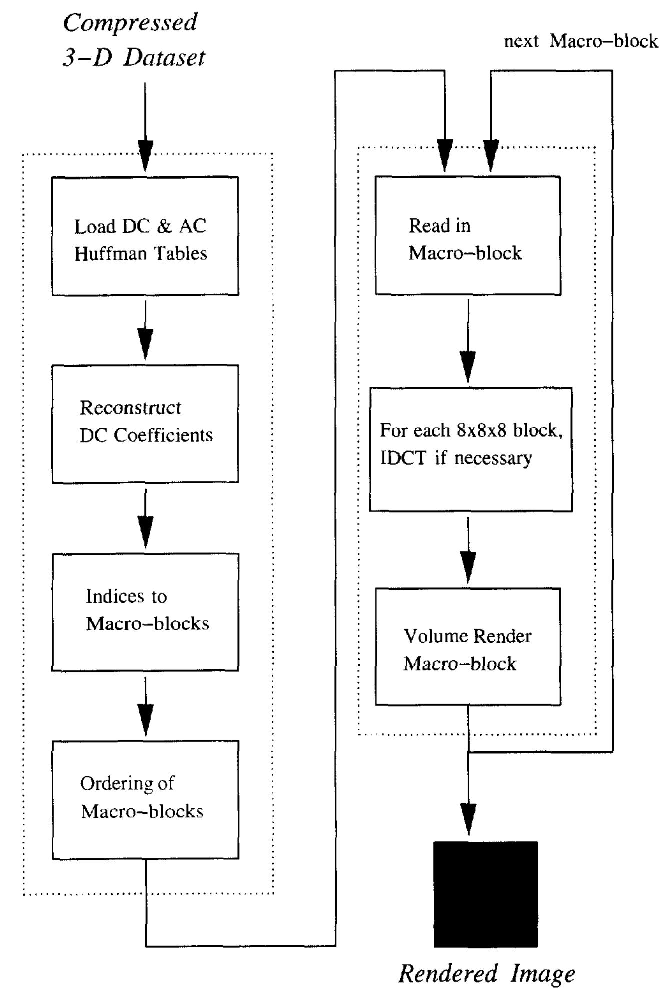

Compressed 3-D Dataset
next Macro–block
Load DC & AC Huffman Tables
Reconstruct DC Coefficients
Indices to Macro–blocks
Ordering of Macro–blocks
Read in Macro–block
For each 8x8x8 block, IDCT if necessary
Volume Render Macro–block
Rendered Image

Fig. 3. Block diagram for rendering steps.

The first stage reads the relevant header information from the compressed data. These include the Huffman table for differentially coded DC and run-length encoded AC coefficients and length of each macro-block. DC coefficients are reconstructed from its differentially coded forms. An index table is also constructed to enable random access to individual macro-block in the compressed file. In addition, given a viewing matrix, the macro-blocks are ordered so that minimal book-keeping is necessary when combining the rendered results from each macro-block. Section V explains this in more detail.

The second stage iteratively reads in a macro-block according to the order computed at the first stage. Random access of macro-block in the compressed file is possible through the index table created. For each such macro-block, each $8 \times 8 \times 8$ block is examined in the transform domain and Inverse DCT (IDCT) is performed when necessary. Approximation techniques described in Section VII evaluate the run-length coded AC coefficients to test if IDCT is necessary. Finally, the macro-block is rendered. The arrangement of macro-blocks to permit all information needed for classification and shading to be local in a macro-block is described in Section VI.

V. Ordering of Macro-Blocks

Volume rendering is performed locally for each macro-block. Such block-wise approach is equivalent to segmenting a ray from a given view-point such that segmented piece will be associated with the macro-block in which it passes. This method of data division and ray segmentation is similar to the methods described in [28], [29], [30] for distributed volume rendering on parallel computers. Working with a macro-block at a time permits one to work with a small amount of memory at a time. This is crucial especially when the volume data is huge and cannot fit into the main memory.

If the macro-blocks are rendered in the order they are stored in the file, then a lot of book-keeping is necessary in order that the locally rendered images can be combined to form the final image. In addition, because the order a ray is sampled is no longer in order of increasing distance from the image plane, adaptive termination of ray tracing [18] cannot be implemented in this framework.

However, it is not hard to see that a viewing matrix induces an ordering (not necessarily unique) $B_1, \ldots, B_m$ on the collections of macro-blocks such that for any ray and the sequence of macro-blocks $B_{k_1}, \ldots, B_{k_n}$ it visits in order of depth,

$$k_1 < k_2 < \ldots < k_n$$

always holds. This ordering depends only on the viewing matrix and is independent of the data set.

Fig. 4 illustrates an ordering for a given viewing direction in 2D. The order of visit of macro-block in this figure is 1, 2, ..., 12. It is easy to see that in this case, the sequence of macro-block visited by any ray in the view direction is numbered in increasing order. Order of visit of slices of macro-blocks is increasing along the coordinate axis most perpendicular to the viewing direction.

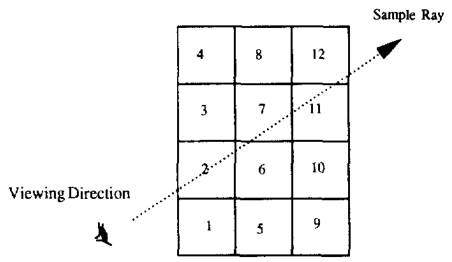


Fig. 4. Ordering of macro-blocks: ray visits blocks 2, 6, 7, 11, 12.

By visiting the macro-blocks in such order, we only need to maintain a color and opacity for each pixel of the image plane throughout the entire rendering process. For any given pixel, each time a macro-block that will affect it is visited, the old values of the color and opacity are used as initial values to the recursions (4) and (5), and the resulting values become the new values. The rendered image is simply the value of the color after all the macro-blocks have been traversed. No further manipulations or book-keeping are necessary. In addition, adaptive termination of ray tracing [18] can be easily implemented in such frameworks. The little overhead (less than 0.1 percent of the compressed data) in keeping the length of DC coefficients and AC macro-blocks permits such random access.

Under this framework of reordered macro-blocks, the pseudocode in Fig. 1 becomes that shown in Fig. 5. Here, the last color and opacity values for a given pixel is used as the initial values. The functions first() and last() include also the macro-block of interest as an argument.

```
rgba RAYTRACE $(\vec{x}, \vec{\omega}, c\_prev, mblock)$
1 $c \leftarrow c\_prev$
2 **for** $i \leftarrow$ first$(mblock, \vec{x}, \vec{\omega})$ to
        last$(mblock, \vec{x}, \vec{\omega})$ **do**
3     $s \leftarrow$ sample$(\vec{x} + i\vec{\omega})$
4     $c \leftarrow c$ over $s$
5 **return** $c$
```

Fig. 5. Pseudocode for blockwise ray tracing.

When the macro-blocks in Fig. 4 are visited and rendered in the order as shown, the intensity, $c(\vec{x})$ at pixel $\vec{x}$ is formed by the sequence of call to **RAYTRACE** shown in Fig. 6. $\vec{x}$ is the point from which the ray in Fig. 4 starts. $c(\vec{x})$ is initialized to (0, 0, 0, 0). After all the macro-blocks have been visited, $c(\vec{x})$ is the value of the rendered image at $\vec{x}$. No further manipulation is necessary by virtue of the way the macro-blocks have been visited.

[in Macro – block 2]

$c(\vec{x}) \leftarrow \mathbf{RAYTRACE}(\vec{x}, \vec{\omega}, c(\vec{x}), \text{Macro – block - 2})$

[in Macro – block 6]

$c(\vec{x}) \leftarrow \mathbf{RAYTRACE}(\vec{x}, \vec{\omega}, c(\vec{x}), \text{Macro – block - 6})$

⋮

[in Macro – block 12]

$c(\vec{x}) \leftarrow \mathbf{RAYTRACE}(\vec{x}, \vec{\omega}, c(\vec{x}), \text{Macro – block 12})$

Fig. 6. Order of update to $c(\vec{x})$.

VI. ARRANGEMENT OF MACRO-BLOCKS

To perform ray tracing in a block-wise fashion involves sampling along a ray in each block and computation of $\nabla x(l, m, n)$ defined in (2). We need values of each voxel and of adjacent voxels to be available locally in each block.

A. 1-Space-Filling Block Arrangement

We first consider the simpler case of shading and classification using only $x(l, m, n)$ alone. Consider sampling along a ray in each block, and assume tri-linear interpolation which makes use of the eight neighboring shaded and classified voxels. We then require that the construction of block in a way that the collection of blocks fill the entire volume. Fig. 7 illustrates a conventional way of dividing a 2D image into blocks. A ray is shown to pass through two disjoint blocks. The marks on the ray represents sampling points. The point, labeled 'a' is between the two blocks, so it cannot be handled by the local ray tracing in either block alone. This problem will be especially severe if a ray happens to pass entirely through the region between two blocks. A black line would then result in the projected image. To overcome this difficulty, we divide the data set into blocks that share boundaries as illustrated in Fig. 8.

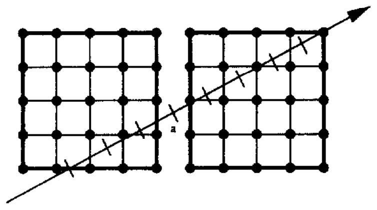


Fig. 7. Non-overlapping blocks.

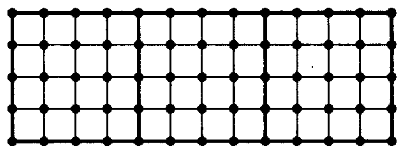


Fig. 8. 2D example of 1-space-filling block.

In 3D, the outer-most layer of voxels in each block is shared with its neighboring blocks. We shall call such arrangement a 1-space-filling arrangement (in general, an n-space-filling arrangement is one in which n layer is shared between two adjacent blocks).

This approach, however, will necessarily incur extra storage. To investigate the increase in data size, we proceed as follows. Let the size of the data set be $L_x \times L_y \times L_z$ and size of a block $N_x \times N_y \times N_z$. Then the number of blocks required in the 1-space-filling arrangement is

$$\left\lceil \frac{L_x - 1}{N_x - 1} \right\rceil \times \left\lceil \frac{L_y - 1}{N_y - 1} \right\rceil \times \left\lceil \frac{L_z - 1}{N_z - 1} \right\rceil. \quad (9)$$

We tabulate in Table II for various values of data set sizes and block sizes and the corresponding increase in data size. There are significant increases in data size when the 1-space-filling blocks do not fill the entire data set exactly. This actually corresponds to the case when $\frac{L_x-1}{N_x-1} \neq \left\lceil \frac{L_x-1}{N_x-1} \right\rceil$ or $\frac{L_y-1}{N_y-1} \neq \left\lceil \frac{L_y-1}{N_y-1} \right\rceil$ or $\frac{L_z-1}{N_z-1} \neq \left\lceil \frac{L_z-1}{N_z-1} \right\rceil$ in (9).

However, while the 1-space-filling block arrangement solves the problem of permitting arbitrary sampling ray, it still does not permit shading and classification that make use of $\nabla x(l, m, n)$. We shall treat the more general problem and proceed to consider the case when gradient computation is needed in the next section.

TABLE II
DIFFERENT DATA SET, BLACK SIZES, AND INCREASES IN DATA SIZE FOR 1-SPACE-FILLING ARRANGEMENT

	Data Size	128^3	256^3	512^3
Block: 8^3	% Increase	67.5	54.6	48.4
Block: 16^3	% Increase	42.4	20.0	30.8
Block: 32^3	% Increase	95.3	42.4	20.0

B. 3-Space-Filling Block Arrangement

It is clear from Fig. 8 that if we need to shade and classify the voxels using $\nabla x(l, m, n)$ on the common layer between two adjacent blocks, then we need a 3-space-filling block arrangement. The outermost three layers of voxels in each block are shared by their adjacent blocks. The 1D counterpart is shown in Fig. 9 with total of 14 points and each block is six points ($L = 14$, $N = 6$). The point marked with $\times$ is the extra point needed to fill up the last block. The block containing the last few voxels together with non-zero number of extra voxels will be called *boundary block*. With such arrangement, only the inner voxels in each block are considered for the local blockwise ray tracing and the outer voxels (corresponding to the outer-most layer) are used for computation of $\nabla x(l, m, n)$ of the inner voxels.

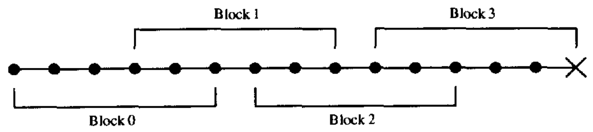


Fig. 9. 1D example of 3-space-filling arrangement.

Analogous to (9) and using the same notation as before, the number of blocks required in the 3-space-filling arrangement is

$$\left\lceil \frac{L_x - 3}{N_x - 3} \right\rceil \times \left\lceil \frac{L_y - 3}{N_y - 3} \right\rceil \times \left\lceil \frac{L_z - 3}{N_z - 3} \right\rceil. \tag{10}$$

The increase in data size for various values of data set and block size are tabulated in Table III. It can be seen that the increase in data size tends to be smaller for larger block sizes.

TABLE III
DIFFERENT DATA SET, BLOCK SIZES, AND INCREASES IN DATA SIZE FOR 3-SPACE-FILLING ARRANGEMENT

	Data Size	128^3	256^3	512^3
Block: 8^3	% Increase	281	305	305
Block: 16^3	% Increase	95.3	95.3	95.3
Block: 32^3	% Increase	95.3	42.4	42.4

One way to reduce the growth of data size is to use larger blocks for DCT. There is however a tradeoff between using larger blocks and achieving good compression since block transform takes advantage of localization of information to achieve energy compaction. It is reported in [31] that there is no improvement in average energy compaction for 2D block sizes above 8×8 for a wide range of pictures. Use of larger blocks also means a higher cost in terms of decompression: an 8-IDCT requires 40 operations while a 16-IDCT requires 112 operations [32].

Instead, we group 64 8×8 DCT blocks into a *macro-block* of size $32 \times 32 \times 32$ and set up the 3-space-filling arrangement on these macro-blocks. In addition, because of the nature of DCT-based compression and the ways encoding are carried out, if voxel repetitions are performed at the boundaries of the data set in order to fit the size of the block, such redundancies will allow us to encode these boundary blocks with significantly fewer bits. Coupled with the fact that voxels at the last few layers of a data set typically do not contain important information and have values close together, boundary blocks can then be encoded with very few bits. This provides a lower bound on the number of blocks:

$$\left\lfloor \frac{L_x - 3}{N_x - 3} \right\rfloor \times \left\lfloor \frac{L_y - 3}{N_y - 3} \right\rfloor \times \left\lfloor \frac{L_z - 3}{N_z - 3} \right\rfloor. \tag{11}$$

The lower bound on the increase of data size is tabulated in Table IV which show some significant decreases of overhead of data size in comparison with Table II.

TABLE IV
LOWER BOUND ON THE INCREASES IN DATA SIZE FOR 3-SPACE-FILLING ARRANGEMENT

	Data Size	128^3	256^3	512^3
Block: 8^3	% Increase	281	281	293
Block: 16^3	% Increase	42.4	67.5	81.0
Block: 32^3	% Increase	0	0	19.9

C. Adaptive Block Sizes

One further way to reduce the increase of data size is to use smaller block sizes to contain those voxels in the last few layers that do not fit exactly into the macro-block. For instance, if we have a 128^3 data set and a 32^3 macro-block, then along each dimension, there will be five blocks. Take for example voxels from $[115, \ldots, 127] \times [0, \ldots, 31] \times [0, \ldots, 31]$. Rather than using a 32^3 block to contain these boundary voxels and using voxel repetitions, we could use a $16 \times 32 \times 32$ block to minimize voxel repetition subjecting to the constraint that each dimension of the block size remains a multiple of eight. Such adaptation of block sizes at the boundaries is used in actual implementations.

To compare the effect of compression on non-overlapping block and that of 3-space-filling block arrangement with adaptive block sizes, we tabulated in Table V the compression ratio of two $256 \times 256 \times 256$ data, cadaver and angiogram, at a different scaling factor s of quantization step sizes. We see that with compression, the boundary block tend to need fewer bits to code so that the 42.4 percent predicted in Table III is much less. This drop in compression ratio is acceptable and is the price to pay to keep extra information to allow local-blockwise rendering of the 3D data set.

One alternative method to cope with boundary voxels is to use an n-point DCT, where $n < 8$. However, the use of an 8-point DCT allows simplicity and permits use of hardware for fast computation. The regularity of a 2-point butterfly used in deriving fast algorithms does not carry over to non-radix-2 computation. Furthermore, use of a non-cubic block means

TABLE V
COMPRESSION RATIOS FOR NON-OVERLAPPING BLOCK APPROACH AND 3-SPACE-FILLING WITH ADAPTIVE BLOCK SIZE FOR 256^3 CADAVER AND ANGIOGRAM DATA

s	Cadaver			Angiogram		
	Non-overlapping block	3-space-filling (adaptive)	% decrease	Non-overlapping block	3-space-filling (adaptive)	% decrease
1	18.1	13.7	24.3	32.5	25.0	29.2
2	26.8	20.3	24.3	51.4	39.6	23.0
4	40.0	30.3	24.3	85.8	66.4	22.6
8	60.6	45.9	24.3	152.1	118.4	22.2

that AC coefficients groupings and quantizations have to be also adapted according to n. The overhead with voxel repetition to allow 8-point DCT is very small because the boundary voxels typically have values close to zero.

D. Random Voxel Access

Here we develop properties of 3-space-filling block arrangements that facilitate random access of particular voxels. Conversions between data set voxel position to block number and voxel position within the block are developed. To this end, we label a block by a 3-tuple (i, j, k) and a voxel within a block by (r, s, t), where each variable starts from zero. Such a voxel will be called a $(<i, r>, <j, s>, <k, t>)$ voxel. This voxel will also have a global position (x, y, z) with respect to the entire data set. This section develops properties concerning the conversion between the global (x, y, z) and the block-local $(<i, r>, <j, s>, <k, t>)$ representation.

Recall that voxels on the boundaries of a block do not contribute to the ray tracing. They are there only for the purpose of ray-sampling and shading/classification. We illustrate in Fig. 10 the same 1D example as in Fig. 9 with the voxels within each block that contribute to the rendering process of that block appropriately grouped. Selected voxels are also labeled using the block-local representation. Voxels that are shaded/classified will be called *useful* in that block. In each block, only the boundary voxels (except the first and last block) are not *useful*. In this example, five voxels are *useful* in block 0: <0, 0>, <0, 1>, <0, 2>, <0, 3>, and <0, 4>. <0, 5> is not *useful* in block 0. The *useful* voxels in block 1 are <1, 1>, <1, 2>, <1, 3>, and <1, 4>. <1, 0> and <1, 5> are only used to shade/classify <1, 1> and <1, 4>, respectively.

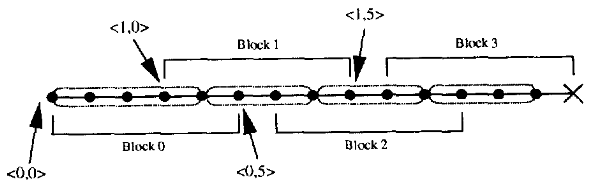


Fig. 10. 1D example of 3-space-filling arrangement.

Because each coordinate can be treated independently, we state some properties in the 1D case. Assume that there are L voxels and N voxels per block.

1) There are
   a) $N - 1$ *useful* voxels in the block 0;
   b) $N - 2$ *useful* voxels in block 1, ..., $\lceil \frac{L-3}{N-3} \rceil - 2$, and
   c) $L - \left(\lceil \frac{L-3}{N-3} \rceil - 2\right) \times (N-3) - (N-2)$
   $= L - (N-3)\lceil \frac{L-3}{N-3} \rceil + N - 4$
   useful voxels in the last block.

   If $\frac{L-3}{N-3} = \lceil \frac{L-3}{N-3} \rceil$, the last block contains $N-1$ *useful* voxels.
2) To convert from block-local position $<i, r>$ to global position x, we have

$$x = \begin{cases} r, & i = 0; \\ i(N-3) + r, & i > 0. \end{cases} \tag{12}$$

   Of course, x should be in the range 0, ..., $L - 1$. For example, <0, 3> and <1, 0> in Fig. 10 are both mapped to $x = 3$.
3) To convert from global position x to block-local position $<i, r>$, we have $<i, r> =$

$$\begin{cases} <0, x>, & \\ & x < N-2; \\ \left\langle \left\lfloor \frac{x-N+2}{N-3} \right\rfloor + 1, \right. & \left. (x - N + 2) \bmod (N-3) + 1 \right\rangle \\ & x \geq N-2. \end{cases} \tag{13}$$

   This mapping is made such that only *useful* voxels are obtained. In addition, because between adjacent blocks (1D), there will be one overlapping *useful* voxel, we choose to unambiguously assign such voxels to the block with higher block index. Such unique assignment is necessary at rendering time because we do not want to compute the contribution of such voxels more than once if it happens that a ray is sampled at such voxels.[1] For example, consider again Fig. 10. If $x = 5$, then (13) gives <1, 2>. If $x = 7$, then it is a *useful* point in both block 1 and 2; (13) gives <2, 1>—the assignment is made to the block with higher block index.
4) For block i, the first voxel starts at global position $i(N - 3)$.

[1] However, such overlapping voxels are still needed for interpolation purposes to both the adjacent blocks. Thus they are still *useful* to both blocks.

5) For block i, the range of global positions for the *useful* voxels is

$$\begin{cases} 0, \ldots, N-2 & i = 0; \\ i(N-3)+1, \ \ldots, (i+1)(N-3)+1, & i = 1, \ldots, \left\lceil \frac{L-3}{N-3} \right\rceil - 2; \\ i(N-3)+1, \ \ldots, L-1, & i = \left\lceil \frac{L-3}{N-3} \right\rceil - 1 \end{cases} \tag{14}$$

E. Alternative Methods

Previous sections deal with the construction of blocks to permit local rendering. Operations like interpolation and shading/classification require values of neighboring voxels. The need to keep all the needed information local leads to the 3-space-filling voxel arrangement. The penalty is about 25 percent decrease in compression ratio, as indicated in Table V. We discuss other possibilities to achieve the same goal.

The need to compute (2) for shading/classification leads us to consider the 3-space-filling instead of a 1-space-filling arrangement. One possibility is to compress ∇x, as is done in [3]. However, even if we assume that there is no extra overhead in using a 1-space-filling arrangement, it is unlikely that 25 percent overhead can be used to compress ∇x, which contains three times the information compared to the data set. For a given compression ratio, it is most likely that one has to sacrifice the quality of the data set (i.e., use large compression ratio) to accommodate a more accurate representation of ∇x. The relative importance of x and ∇x to shading and classification and thus to the rendered image needs to be studied more carefully to shed light on how bits should be allocated to x and ∇x for better compression of these quantities.

An alternative method is to use a multiple block approach, where there is no longer voxel sharing between blocks. Instead, adjacent blocks are reconstructed when voxels in these blocks are needed. Consider a 1D example shown in Fig. 11. Block A consists of eight points, labeled 'e' to 'l.' To permit shading (using ∇x) of points in a local blockwise fashion without any voxel sharing, points from at least one neighboring block must be available. To see also that one neighboring block is sufficient, we reason as follows: suppose we only make use of the previous adjacent block, then points 'e' to 'k' in the block A can be shaded. Point 'l,' however, cannot be shaded using values from the current block only. But at the next block (points 'm' onward), and using the current block, point 'l' can be shaded. This chain of reasoning leads to the condition that in 1D, we can perform shading on a block, with at most one other block loaded.

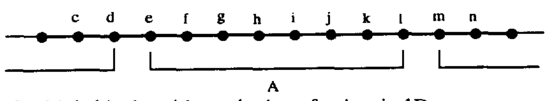


Fig. 11. Multiple blocks without sharing of points in 1D.

Extending the argument to 3D, it is easy to see that with three extra blocks, one does not need to have voxel sharing block arrangement to locally shade all voxels. There is, however, a problem with interpolation near corner voxels, as is illustrated in Fig. 12 for 2D. Assume appropriate shading of pixels in Block A using information from the two neighboring blocks. The value at 'm' cannot be obtained via interpolation because pixel 'b' cannot be shaded. Even ignoring such effects for the moment, there is the need to load three extra blocks into memory at a time. This means that extra transfer of data and computation in terms of decompression would be necessary. If we denote by f_d the percentage of the total rendering time spent on decompression, then using three extra blocks incur an increase of $3f_d$ percent in the total time. In our experimentation f_d ranges typically from 10 to 15. So an increase of about 30 percent in processing time is incurred even when effect of interpolating near corner voxels is ignored. This is clearly an issue of tradeoff between compression ratio and processing time in the ways blocks are arranged and loaded into memory.

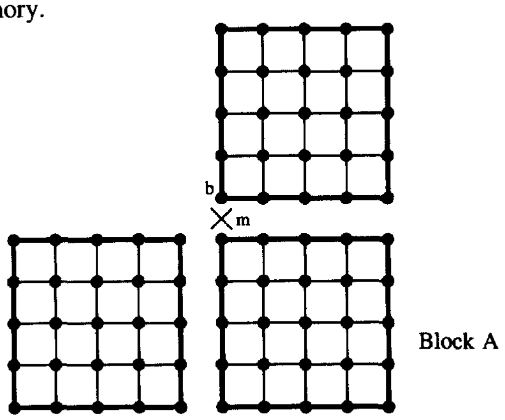


Fig. 12. Multiple block without sharing of pixels in 2D.

VII. Accelerated Rendering Via DCT Information

In this section, we will show how DCT information can be used to permit speedup in rendering without significant loss of image quality. Under certain conditions, a block need not be decompressed. A near-homogeneity measure eliminates IDCT. From the compressed data, we can assess whether a block is nearly homogeneous. If so, replacing it by a constant value will lead up speedup without perceptible distortion error.

We will work directly from the run-length encoded data. This means we will deal with the (*run*, *value*) pairs, as described in Section III.A.3. The value will be the quantized coefficient $f^q_{i,j,k}$. In the following, for simplicity, the equations will be given in terms of $f_{i,j,k}$, which is related to $f^q_{i,j,k}$ via $f_{i,j,k} = f^q_{i,j,k} q_{i,j,k} s$.

A. Average Value of a Block

The DC coefficient

$$f_{0,0,0} = 16\sqrt{2}\left[\frac{1}{512}\sum_{l=0}^{7}\sum_{m=0}^{7}\sum_{n=0}^{7} x_{l,m,n}\right] \tag{15}$$

is the average value or the DC value of the block scaled by $16\sqrt{2}$. Denoting this average by $E(\mathbf{X})$, we have

$$E(\mathbf{X}) = \frac{1}{16\sqrt{2}} f_{0,0,0}. \qquad (16)$$

Note that this value is immediately accessible from the run-length encoded stream. In particular, if all the values in **X** are the same, i.e., all $x_{i,j,k} = E(\mathbf{X})$, then $f_{i,j,k} = 0$ for $(i, j, k) \neq (0, 0, 0)$ and the run-length encoding would contain a run of zeros to the end of block immediately after the DC term. Such knowledge would thus facilitate speedup in processing of such blocks. Regions of emptiness, typical in medical data, have such characteristics.

B. Fluctuations of Values in a Block

A useful measure of fluctuations of values of **X** is the variance

$$\mathrm{Var}(\mathbf{X}) = \frac{1}{N}\sum_{i,j,k} x_{i,j,k}^2 - (E(\mathbf{X}))^2. \qquad (17)$$

The standard deviation is then $sd(\mathbf{X}) = \sqrt{\mathrm{Var}(\mathbf{X})}$. That this operation could be performed directly from transform coefficients follows from the fact that the 1D DCT is a unitary transform. It can be shown that

$$\mathrm{Var}(\mathbf{X}) = \frac{1}{512} \sum_{(i,j,k)\neq(0,0,0)} f_{i,j,k}^2. \qquad (18)$$

Due to the energy compaction properties of the DCT [33], there will be fewer non-zero $f_{i,j,k}$s than non-zero $x_{i,j,k}$s. As a consequence, (18) can be more efficiently evaluated in the transform domain. We have also the added advantage of knowing precisely where the non-zero terms are from the run-length encoded streams.

While the variance found in (18) is useful as an indication of the fluctuation of values in a block, it is probably more useful to determine the range of values in a block. If we could ascertain that a block contains values that are all very close to its mean value, then operations like classification and shading can be well approximated without performing an IDCT on the block.

In this light, a good measurement of the homogeneity of a block would be how small $sd(\mathbf{X})$ is. Typically, with high probability, the range of values would fall within

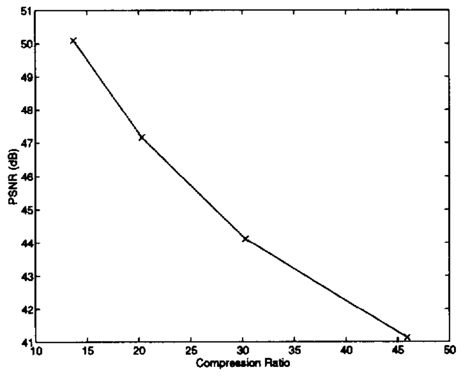
$$[E(\mathbf{X}) - n \times sd(\mathbf{X}),\ E(\mathbf{X}) + n \times sd(\mathbf{X})],$$

for some appropriately chosen n.

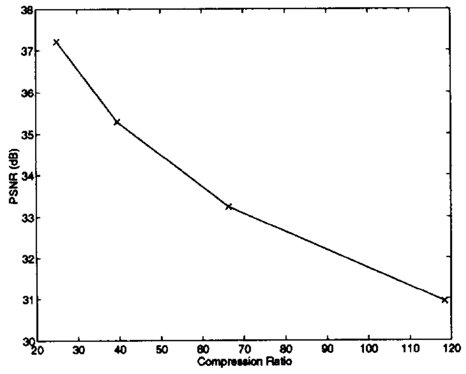
If a DCT block is considered nearly homogeneous, we could simply replace the values of the voxels in the block by its DC value, computed from (16). We shall call such a replacement *DC approximation.* Such an approximation reduces the computation times. It may be justified as follows: since volume rendering tends to highlight regions of high spatial variations (for example, assigning a high opacity value to voxels with large values of $|\nabla x(l, m, n)|$), blocks with small spatial variations will make little difference to the final rendered image. In the particular model in (1), it is clear that all voxels except those on the boundaries in a DC-approximated block will be assigned zero opacity. Thus, in addition to not requiring IDCT on such blocks, gradient computations and shading/classification are not required on all voxels except those on the boundaries.

VIII. Results

We experimented with two data sets: a CT scan of a cadaver-head and a CT angiogram of a human head. The first data set is provided by Softlab Software Systems Laboratory, Department of Computer Science, University of North Carolina at Chapel Hill, and is commonly used for volume rendering in computer graphics literature. The second data set is provided by Siemens Corporate Research, Princeton, New Jersey, and is a result of an automatic computed tomography angiography technique [34] that subtracts two volumetric data sets, one containing contrast agents and the other not, for visualization of blood vessels. This angiogram data set contains appreciable amount of subtraction artifacts due to uncanceled non-rigid motions. Both data are 256 × 256 × 256. Each voxel is 12 bits.



(a)



(b)

Fig. 13. PSNR of rendered image vs. compression ratios: (a) cadaver head, (b) angiogram.

A. Compression Results

Compression is performed on an SGI Challenger. The data set is stored on a local hard disk (i.e., no data transfer through the network). Fast DCT algorithms described in [32] are used. The compression process is divided into four steps: loading of data set into memory, first pass through the data set, Huffman table generation for DC and AC coefficients and second pass through the data set. $s = 1$ is used in deriving the table. The time in the first pass includes DCT and generation of frequency tables for Huffman coding; the time in the second pass includes also DCT and the time to write the compressed bitstream out to file. A breakdown of the compression time is shown in Table VI. In the current implementation, DCT is performed twice on each block, once in each pass. The repetition can be saved if the DCT results are stored in memory. But this means requiring at least twice the memory size. In that case, second pass reduces to generation of compressed bitstreams. The timing in Table VI could then be cut by about 50 seconds. The compression speed of the current scheme permits one to change the compression ratios easily.

TABLE VI
BREAKDOWN OF COMPRESSION TIME (SEC) FOR 256 × 256 × 256 DATA SET

Dataset	Loading Time	First Pass	Huffman Table Generation	Second Pass	Total Time
Cadaver	10	55	0.1	67	132.1
Angiogram	10	55	0.1	56	121.1

The performance of compression is evaluated by the signal-to-noise ratio (SNR)[2], given by

$$SNR = 10\log_{10}\frac{\sum_{i=1}^{N} x_i^2}{\sum_{i=1}^{N}(x_i - \hat{x}_i)^2} \tag{19}$$

where N is the size of 3D data set, x_i and $\hat{x}_i$ are the original and reconstructed voxels, respectively. The compression ratio, bits/voxel (bpv) and SNR performance for s = 1, 2, 4, 8 is tabulated in Table VII.

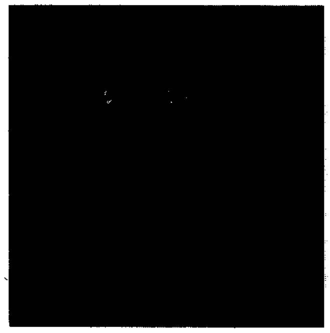
TABLE VII
COMPRESSION PERFORMANCE FOR 256^3 CADAVER AND ANGIOGRAM DATA

	Cadaver			Angiogram		
Scaling factor	Compression Ratio	Bits/voxel (bpv)	SNR (dB)	Compression Ratio	Bits/voxel (bpv)	SNR (dB)
1	13.7	0.860	40.1	25.0	0.48	27.8
2	20.3	0.591	36.8	39.6	0.30	25.0
4	30.3	0.396	33.7	66.4	0.18	22.4
8	45.9	0.261	30.6	118.4	0.10	19.7

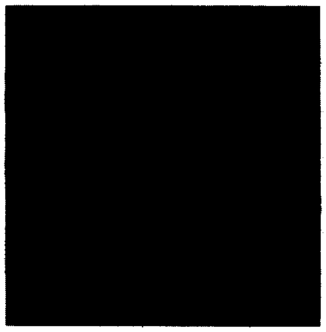
We show selected 2D slices of the original and the decompressed cadaver head data in Fig. 14 and angiogram data in Fig. 15. In the cadaver head data, little distortion is observable except near the edges. In the angiogram data, more blockiness can be seen due to the presence of more higher frequency terms inherent in the data.

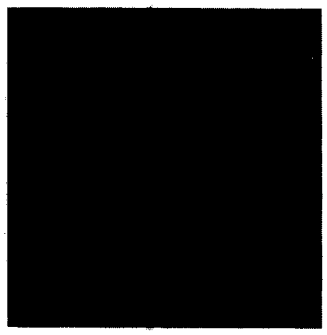
B. Rendering Results

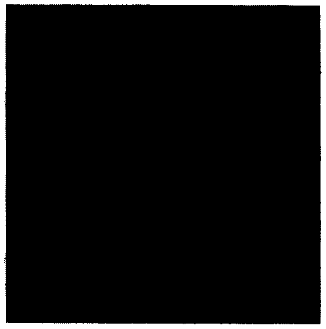
To evaluate the performance of blockwise rendering from compressed data, we tabulate in Table VIII the run-time (in seconds) of volume rendering of the entire uncompressed data and blockwise rendering from compressed (s = 1) data. The compression ratios, as seen from Table VII are 13.7 and 25, respectively. We use an SGI Challenger under two different memory configurations for comparison: one has sufficient memory to load the entire original data and hold additional relevant information while the other does not. Rendering at 256 × 256 image resolution of compressed cadaver and angiogram data is made. To illustrate the effect of compression on the rendered images, we use two compression ratios. The smaller ratio yields rendered images that are virtually indistinguishable from those generated from the original ones. Slight distortion can be observed at the rendered images from the data compressed with larger ratio. The views generated are those shown in Figs. 16 and 17. The code is not optimized and implements shading and opacity classification as described in [16]. In both rendering, a ray is terminated when the opacity exceeds 0.95. The complexity is $O(N^3)$. We show in Fig. 18 an enlarged portion of the rendered angiogram. Here, some blockiness from compression can be observed.


(a)

(d)


(b)


(e)


(c)

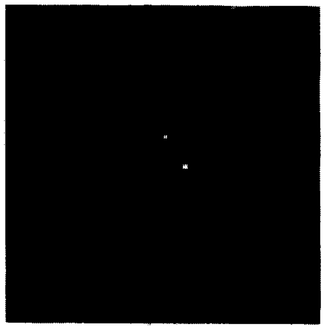

(f)

Fig. 14. 2D images of cadaver head: (a) original, (b) compressed 20.3 times, (c) compressed 45.9 times, (d) enlarged portion of (a), (e) enlarged portion of (b), and (f) enlarged portion of (c).

[2] Peak SNR or PSNR = $10\log_{10}\left(P^2/\frac{1}{N}\sum_{i=1}^{N}(x_i-\hat{x}_i)^2\right)$, where P is the peak signal value, is a popular objective measure for 8-bit images in image processing literatures; SNR is chosen here because for 12-bit data, the PSNR can be misleadingly high.

TABLE VIII
TIMING (SEC) FOR RENDERING OF ORIGINAL/COMPRESSED DATA UNDER DIFFERENT MEMORY SIZE

Dataset	Memory: 256 Mbytes			Memory: 64 Mbytes		
	Original	Compressed	Speedup	Original	Compressed	Speedup
Cadaver	217	172	1.26	668	172	3.88
Angiogram	206	156	1.32	500	156	3.21

The time to render the compressed data is shorter than for the original data even when there is sufficient memory (256 M) and despite the time needed to perform some decoding. This is due to several factors: the presence of regions of emptiness in the data, a lower disk-to-memory transfer cost for the compressed data, and a better memory cache utilization for using only a small amount of memory. Speedup up of about 30 percent in rendering time is obtained under sufficient memory conditions.

It is clear that with limited memory (but with sufficient swap space), as will be the case when the data size gets very large, the proposed blockwise rendering from compressed data would outperform direct rendering of the uncompressed volume data.

To illustrate the size of data set and the effect of compression on the transmission cost, we tabulate in Table IX the time taken to load the uncompressed and compressed cadaver data set into memory. Two cases are considered: the data set resides on a local hard disk (i.e., no network transfer) and the data set resides in a remote hard disk (i.e., network transfer via Ethernet). For this experiment, the data set has been compressed 13.7 times ($s = 1$). For larger compression ratio, the gain would be even larger. The speedup of transmission cost through compression is obvious.

TABLE IX
AVERAGE TIME TAKEN TO LOAD DATA SET INTO MEMORY

Local Storage			Remote Storage		
Compressed	Uncompressed	Speedup	Compressed	Uncompressed	Speedup
0.35 sec.	10 sec.	28.5	1.8 sec.	49.6 sec.	27.6

While the compression ratio does not factor in directly into the complexity of volume rendering (i.e., shading/classification and ray tracing), the compression gain results in significant speedup in transmission costs. However, larger compression means that more DCT blocks become homogeneous so that DC-approximation can be much more effective.

We measure the effect of compression on the quality of rendered image by considering multiple views and different shading/classification conditions. For each shading/classification condition, 64 views are generated. The peak SNR (PSNR), averaged over all the views generated, is used as an objective measure here. The images rendered from the compressed data set are compared to those rendered from the uncompressed one. For the cadaver data set, three shading/classification models to highlight bones alone, skin alone and both bone and skin are used. One model is used to highlight blood vessels in the angiogram data set. The results are plotted in Fig. 13. The very high PSNR of the rendered cadaver images means that they are hardly distinguishable from the those rendered from uncompressed data. The rendered images of the angiogram data have lower PSNR due to the presence of subtraction artifacts in the data set and the effect of compression on such artifacts.

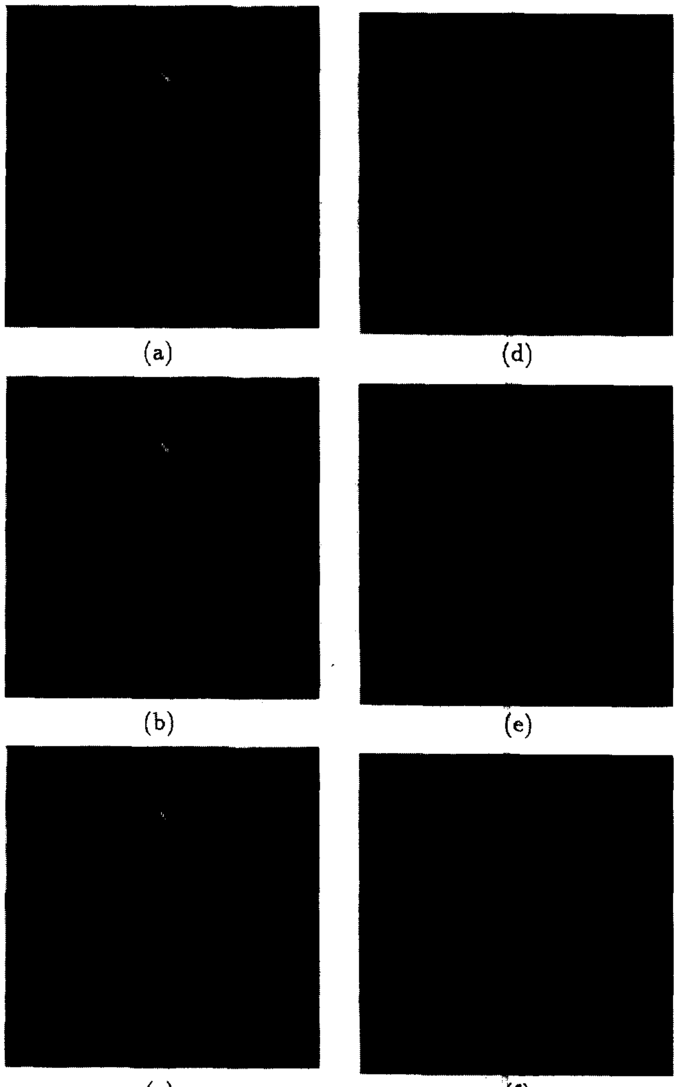


Fig. 15. 2D image of angiogram: (a) original, (b) compressed 25.0 times, (c) compressed 66.4 times, (d) enlarged portion of (a), (e) enlarged portion of (b), (f) enlarged portion of (c).

C. Accelerated Rendering Results

We approximate a block **X** by its DC value if the following two conditions are satisfied for given n and sd,

1) $\min_l \{ i : f_i^r = 0,\ i \geq l \} \leq n$, and

2) $sd(\mathbf{X}) \leq sd$.

Condition 1) implies that $f_n^r = f_{n+1}^r = \cdots = 0$, i.e., the higher frequency coefficients are all zeros. Such an (n, sd) approximation tends to preserve blocks with high frequency components; and since volume rendering techniques as described in [16], [22] tend to highlight regions with high gradient magnitude, most details are preserved at rendering. Note that if criterion 2) is used solely, then we may replace a block containing very high frequency terms but yet having a small variance with a constant block.

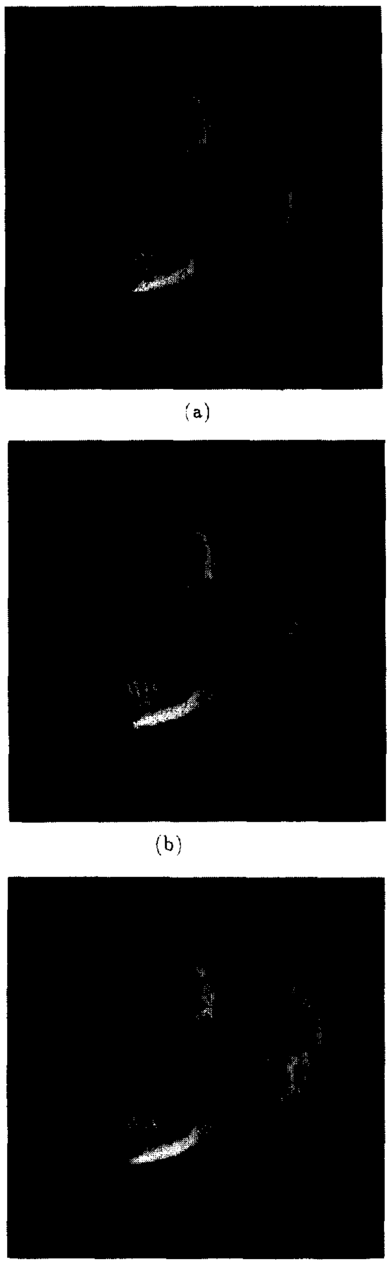


Fig. 16. Rendered cadaver head: (a) original, (b) compressed 20.3, and (c) compressed 45.9 times.

TABLE X
RENDERING TIME AND PERCENTAGE OF DC BLOCK FOR DC-APPROXIMATIONS OF DATA SET

	Cadaver		Angiogram	
n	1	200	1	40
sd	-	200	-	40
Time (sec)	147	116	148	104
% DC Block	33	75	4	65

To compare against rendered results shown previously, we perform approximation on the cadaver compressed 45.9 times and the angiogram compressed 66.4 times. The (*n*, *sd*) parameters are chosen such that the rendered results start to show noticeable distortion. The timing and the percentage of DC blocks are shown in Table X. The table also includes timing when no approximations are made. Note that for the cadaver data, 33 percent of its DCT blocks is constant, whereas for the angiogram data, only 4 percent is constant. The small percentage of emptiness in the angiogram is due to the presence of subtraction artifacts. However, as the table also implies, many high frequency coefficients are quantized to zero. The rendered results of (200, 200.0) approximation of the cadaver head and (40, 40.0) approximation of angiogram are shown shown in Fig. 19. Slight artifacts can be seen at such approximations.

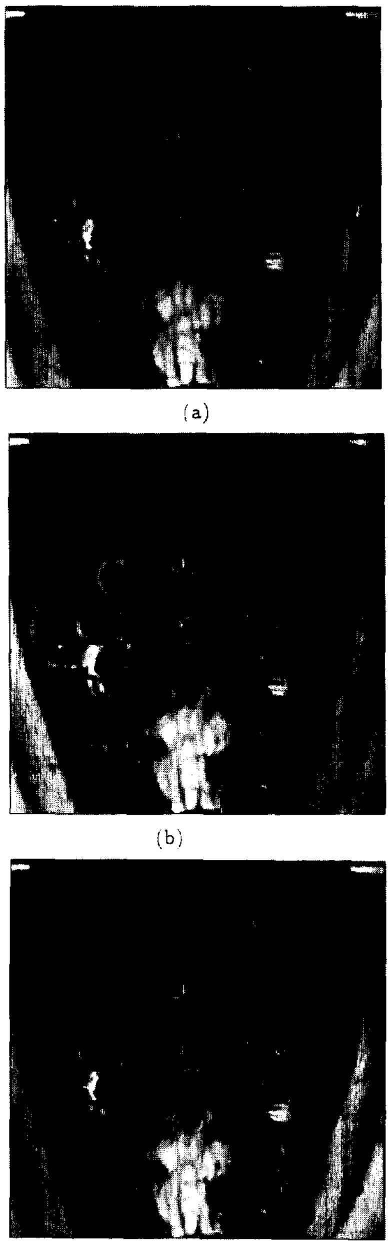


Fig. 17. Rendered angiogram: (a) original, (b) compressed 25.0 times, and (c) compressed 66.4 times.

Fig. 18. Enlarged portion of rendered angiogram: original (left), compressed 25.0 (center), and compressed 66.4 times (right).

The gain from DC-approximation comes from three sources:

1) IDCT is not required for a DC-approximated block. All the voxels' values are equal to the DC value. Only $1 - \gamma$ of the DCT blocks need to be decompressed, where γ is the fraction of DC-approximated blocks.
2) For all voxels except those on the boundary of a DC-approximated block, shading/classification is not needed in the model described by (1). All these voxels will be transparent, i.e., with zero opacity. Since there are $7^3 = 343$ interior voxels, $1-\frac{343}{512}\gamma \approx 1-0.67\gamma$ of the voxels need to be shaded/classified.
3) When the sample points along a ray lie inside a cell with eight transparent voxels, no trilinear interpolation is needed. Such cell populates the interior of a DC-approximated block. A reduction of 67γ percent in interpolation cost is achieved.

Timing obtained from rendering experiments indicate that the above analysis of speedup is accurate. The time taken to raytrace the volume, however, does not change significantly. This is due to the inherent $O(N^3)$ complexity of the raytracing process. Further reduction in rendering time may be possible if preprocessing steps exploiting coherence in the volume such as those used in [18], [21] are performed. Since DC approximation gives constant blocks, such preprocessing could potentially be performed faster.

IX. Conclusions and Discussions

In this paper, we propose a scheme to perform volume rendering from compressed scalar data. Instead of decompressing the entire data set before rendering, only one subblock needs to be decompressed at a time prior to rendering. The compression is 3D DCT-based and is fast, permitting one to experiment with compression parameters easily.
Overlapping macro-blocks are used to permit rendering to be performed in a local fashion. Even with data overlapping, no compression artifacts are noticeable in 2D slices or rendered images at compression ratios of 20 to 30 times. Rendering time on the compressed data is less than direct rendering from the entire data. In situations when memory is limited, this scheme has an obvious advantage. The nature of partition imposed upon the data set also allows easy parallelization of the rendering process.

(a)

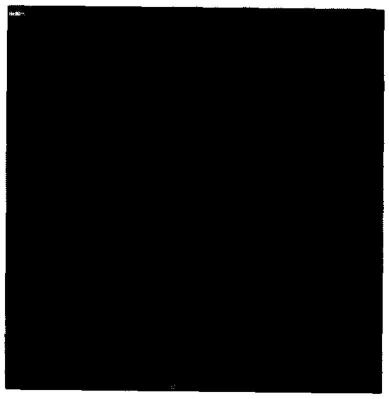

(b)

Fig. 19. (a) Rendered (200, 200.0)-approximation of cadaver head, (b) rendered (40, 40.0)-approximation of angiogram.

Adding only a very small percentage (about 0.1 percent) of overhead in the compressed data permits random access of the compressed macro-blocks. This minimizes book-keeping of intermediate results from each macro-block. No final composition of intermediate results is necessary. In addition, adaptive ray termination [18] can be implemented.

DC-approximation allows adaptive replacement of a homogeneous DCT block by a block equal to its DC value. Decompression is then performed on the significant blocks. The decision whether a block is significant is made entirely in the transform domain, in particular, on the run-length encoded AC coefficients. Speedup in rendering is achieved without significant loss of rendered image quality.

The data set used in this paper is 256 × 256 × 256. The advantages of the proposed rendering method would become more pronounced with even larger data size. For instance, loading a 1024 × 1024 × 1024 16-bit data set into the main memory would be virtually impossible for most work-stations. While the proposed rendering method can deal with arbitrarily large data set by block-wise rendering, compression also reduces the demand for storage space and cuts down transmission overheads.

Even though volume rendering based on (3) is used in this paper, the framework of overlapping block-wise rendering of compressed blocks generalizes to other form of projection operations which can be performed in a piece-wise fashion, such as the maximum intensity projection. Object space approach such as those described in [13], [14] could also be used as a substitute for raytracing approach described in this paper.

Other methods of compressions such as interframe technique [35], [36], subband and wavelet should be explored to evaluate their effectiveness for joint decompression-rendering purposes. Combining these compression schemes with hierarchical methods may also permit better compression and rendering speedup.

Acknowledgments

The authors would like to thank Ming-Yee Chiu and Indranil Chakravarty for helpful suggestions. Special thanks go to the anonymous reviewers for their constructive criticism and suggestions. We thank Professor A. Kaufman for providing reference [8].

This work was suppported in part by an IBM Graduate Fellowship and in part by Siemens Corporate Research, Princeton, N.J. The computing facilities used in the research are supported by the U.S. National Science Foundation under Grant CDA-9216171.

References

[1] K.K. Chan, C.C. Lau, K.S. Chuang, and C.A. Morioka, "Visualization and volumetric compression," *Image Capture, Formatting, and Display*, vol. SPIE 1444, pp. 250–255, 1991.

[2] P. Ning and L. Hesselink, "Vector quantization for volume rendering," *1992 Workshop Volume Visualization*, pp. 69–74, Oct. 1992.

[3] P. Ning and L. Hesselink, "Fast volume rendering of compressed data," *Visualization 1993*, pp. 11–18, Oct. 1993.

[4] S. Dunne, S. Napel, and B. Rutt, "Fast reprojection of volume data," *Proc. First Conf. Visualization Biomedical Computing*, pp. 11–18, 1990.

[5] T. Malzbender, "Fourier volume rendering," *ACM Trans. Graphics*, vol. 12, pp. 233–250, July 1993.

[6] M. Levoy, "Volume rendering using the Fourier projection-slice theorem," *Proc. Graphics Interface '92*, pp. 61–69, May 1992.

[7] T. Totsuka and M. Levoy, "Frequency domain volume rendering," *Computer Graphics*, vol. 27, pp. 271–278, Aug. 1993.

[8] T.-C. Chiueh, T. He, A. Kaufman, and H. Pfister, "Compression domain volume rendering," Tech. Report TR.94.01.04, State Univ. of New York at Stony Brook, 1994.

[9] S. Muraki, "Volume data and wavelet transforms," *IEEE Computer Graphics & Applications*, pp. 50–56, July 1993.

[10] S. Muraki, "Multiscale 3D edge representation of volume data by a DOG wavelet," *Symp. Volume Visualization 1994*, Oct. 1994.

[11] R.A. Drebin, L. Carpenter, and P. Hanrahan, "Volume rendering," *Computer Graphics*, vol. 22, pp. 65–74, Aug. 1988.

[12] C. Upson and M. Keeler, "V-BUFFER: Visible volume rendering," *Computer Graphics*, vol. 22, pp. 59–64, Aug. 1988.

[13] L. Westover, "Footprint evaluation for volume rendering," *Computer Graphics*, vol. 24, no. 4, pp. 367–376, 1990.

[14] J. Wilhelms and A.V. Gelder, "A coherent projection approach for direct volume rendering," *Computer Graphics*, vol. 25, no. 4, pp. 275–284, 1991.

[15] D. Laur and P. Hanrahan, "Hierarchical splatting: A progressive refinement algorithm for volume rendering," *Computer Graphics*, vol. 25, no. 4, pp. 285–288, 1991.

[16] M. Levoy, "Display of surface from volume data," *IEEE Computer Graphics and Applications*, vol. 8, pp. 29–37, May 1988.

[17] P. Sabella, "A rendering algorithm for visualizing 3D scalar fields," *Computer Graphics*, vol. 22, pp. 51–58, Aug. 1988.

[18] M. Levoy, "Efficient ray tracing of volume data," *ACM Trans. Graphics*, vol. 9, pp. 245–261, July 1990.

[19] M. Levoy, "Volume rendering by adaptive refinement," *The Visual Computer*, vol. 6, pp. 2–7, Feb. 1990.

[20] J. Danskin and P. Hanrahan, "Fast algorithms for volume ray tracing," *1992 Workshop Volume Visualization*, pp. 91–98, Oct. 1992.

[21] P. Lacroute and M. Levoy, "Fast volume rendering using a shear-warp factorization of the viewing transformation," *Computer Graphics*, vol. 28, pp. 451–458, July 1994.

[22] D. Ney, E. Fishman, D. Magid, and R. Drebin, "Volume rendering of computed tomography data: Principles and techniques," *IEEE Computer Graphics and Applications*, pp. 24–32, Mar. 1990.

[23] J.F. Blinn, "Light reflection functions for simulation of clouds and dusty surfaces," *Computer Graphics*, vol. 16, pp. 21–29, July 1982.

[24] J.T. Kajiya and B.P.V. Herzen, "Ray tracing volume densities," *Computer Graphics*, vol. 18, pp. 165–174, July 1984.

[25] T. Porter and T. Duff, "Compositing digital images," *Computer Graphics*, vol. 18, pp. 253–259, July 1984.

[26] G.K. Wallace, "The JPEG still picture compression standard," *Comm. ACM*, vol. 34, pp. 30–44, Apr. 1991.

[27] D.A. Huffman, "A method for the construction of minimum redundancy codes," *Proc. IRE*, vol. 40, pp. 1098–1101, 1962.

[28] W.M. Hsu, "Segmented ray casting for data parallel volume rendering," *1993 Parallel Rendering Symp.*, pp. 7–14, Oct. 1993.

[29] K.L. Ma, J.S. Painter, C.D. Hansen, and M.F. Krogh, "A data distributed, parallel algorithm for ray-traced volume rendering," *1993 Parallel Rendering Symp.*, pp. 15–22, Oct. 1993.

[30] E. Camahort and I. Chakravarty, "Integrating volume data analysis and rendering on distributed memory architectures," *1993 Parallel Rendering Symp.*, pp. 89–96, Oct. 1993.

[31] A.N. Netravali and B.G. Haskell, *Digital Pictures: Representation and Compression*, Plenum Press, 1988.

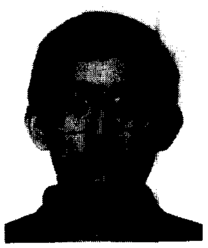
[32] C. Loeffler, A. Ligtenberg, and G. S.Moschytz, "Practical fast 1D DCT algorithms with 11 multiplications," *ICASSP*, pp. 988–991, 1989.

[33] A.K. Jain, *Fundamentals of Digital Image Processing*, Prentice Hall, 1988.

[34] M.M. Yeung, B.L. Yeo, S.P. Liou, and A. Banihashemi, "Three-dimensional image registration for Spiral CT Angiography," *Computer Vision and Patten Recognition*, pp. 423–429, June 1994.

[35] H. Lee, Y. Kim, A.H. Rowberg, and E.A. Riskin, "Statistical distributions of DCT coefficients and their applications to an interframe compression algorithm for 3D medical images," *IEEE Trans. Medical Imaging*, vol. 12, pp. 478–485, Sept. 1993.

[36] A. Nosratinia, N. Mohsenian, M.T. Orchard, and B. Liu, "Interslice coding of magnetic resonance images using deformable triangular patches," *IEEE Int'l Conf. Image Processing*, pp. 892–902, Nov. 1994.



Boon-Lock Yeo received the BSEE degree in electrical engineering (with highest distinction) from Purdue University, West Lafayette, Ind., in 1992, and the MA degree from Princeton University in 1994. He is currently working toward the PhD in electrical engineering at Princeton University.

Yeo was a Singapore Technologies Overseas Scholar in 1989-1991 and is currently a recipient of an IBM Graduate Fellowship. He served in the Singapore Armed Forces from 1986 to 1989. Yeo worked at C-Cube Microsystems, San Jose, Calif., in the summer of 1990, and at Siemens Corporate Research, Princeton, N.J., in the summer of 1993.

Yeo's research interests include designing algorithms and techniques for image processing on compressed videos, multi-dimensional data compression, visualization, and computer vision.

Bede Liu (S'55, M'62, F'72), a professor of electrical engineering at Princeton University, holds a BSEE degree from the National Taiwan University and an MEE degree and a DEE degree from the Polytechnic Institute of Brooklyn. Prior to joining the Princeton faculty in 1962, he was with Bell Laboratories, Allen B. DuMont Laboratory, and Western Electric Company. He has also been a visiting faculty member at the National Taiwan University (Taipei), the University of California at Berkeley, the Imperial College (London), and the Jiao Tong University (Shanghai).

Dr. Liu served as the technical program chairman for the 1978 IEEE International Symposium on Circuits and Systems, the president (1982) of the IEEE Circuit and Systems Society, and the IEEE Division I Director (1984, 1985). He is the recipient of an IEEE Centennial Medal, the 1985 Technical Achievement Award of the IEEE Acoustics, Speech, and Signal Processing Society, the 1988 Education Award of the IEEE Circuits and Systems Society, and the Best Paper Award for Video Technology (1994). In 1988 he was named an honorary professor by the Academia Sinica.

Dr. Liu edited the research volume *Digital Filters and the Fast Fourier Transform* (DHR, 1975) and coauthored the senior level textbook *Digital Signal Processing: Theory, Design, and Implementation* (Wiley, 1976). His current research interests lie mostly in the areas of image and signal processing.