3D RGB Image Compression for Interactive Applications

Chandrajit Bajaj
Department of Computer Sciences
The University of Texas at Austin
U.S.A.

Insung Ihm* Sanghun Park
Department of Computer Science
Sogang University
Seoul, Korea

April 6, 2001

Abstract

This paper presents a new 3D RGB image compression scheme designed for interactive real-time applications. In designing our compression method, we have compromised between two important goals: high compression ratio and fast random access ability, and have tried to minimize the overhead caused during run-time reconstruction. Our compression technique is suitable for applications wherein data are accessed in a somewhat unpredictable fashion, and real-time performance of decompression is necessary. The experimental results on three different kinds of 3D images from medical imaging, image-based rendering, and solid texture mapping suggest that the compression method can be used effectively in developing real-time applications that must handle large volume data, made of color samples taken in three- or higher-dimensional space.

Keywords: 3D volume data, Data compression, Haar wavelets, Random access, Interactive real-time applications, Medical imaging, Image-based rendering, 3D texture mapping

1 Introduction

Volumetric or volume data in computer graphics and scientific visualization are a discrete collection of scalar or vector values sampled in n-dimensional space, where n is typically greater than or equal to 3 [17]. Such data are often produced by volumetric imaging scanners, like CT and MRI, as well as the output of physical simulations. 3D texture maps, that are created by evaluating solid texture functions on a three-dimensional grid, are another example of 3D volumes [7]. Four dimensional space-time volume data appears frequently in computational fluid dynamics and global climate simulations [23]. Sampled light fields or lumigraphs created for image-based rendering are also volume data in 4D [19, 12]. Interactively handling and visualizing such datasets has become increasingly important.

*Currently, on leave at Texas Institute for Computational and Applied Mathematics of The University of Texas at Austin.

Typical volume data are often very large in size, ranging from several hundred megabytes to several dozen gigabytes. Developing interactive real-time applications with such data assumes, implicitly or explicitly, that the entire data can be loaded into main memory for efficient run-time processing. This places enormous burden on in-core storage space as well as transmission bandwidth. One way to alleviate this problem is to store compressed representations. There are several data compression techniques, most of which are geared towards achieving the best compression rate with minimal distortion in the reconstructed images [11, 30] (*High compression rate and visual fidelity*). Such compression methods, however, often impose constraints on the random access ability, which makes them inappropriate for interactive graphics applications especially where it is difficult to predict data access patterns in advance (*Fast decoding for random access.*). For instance, variable-bitrate or differential encoding schemes such as the Huffman or arithmetic coders coupled to block JPEG or MPEG schemes, do not lend themselves to efficiently decode individual data items that are accessed in a random pattern in interactive exploration.

In order to be used in developing real-time or, at least, interactive-time graphics applications, a compression method must satisfy some requirements. In addition to the two aforementioned issues, we consider the following properties, as similarly discussed in [3, 19]:

- *Multi-resolution representation.* It is highly recommended to choose a compression technique that additionally provides a multi-resolution representation. This offers the basis for LOD (Level of Detail) processing of compressed data.

- *Effective exploitation of data redundancy.* In general, n-dimensional volume data exhibit redundancy in all n dimensions. A compression scheme devised for 2D images, for example, could be applied to compress each slice in 3D volumes, however, a good compression technique must be able to fully exploit data coherence in all dimensions to maximize compression performance.

- *Selective block-wise compression.* In some applications like 3D texture mapping, as will be demonstrated in Subsection 4.3, it is more effective to selectively compress a dataset block-wise rather than the entire dataset in totality. It is very desirable that a compression scheme includes this selective compression capability in its encoding algorithm for better compression.

Vector quantization [10], that meets some of the above five properties, has been popular in developing interactive real-time applications mainly because it supports fast random decoding through simple table lookups. Recent

applications of vector quantization in the computer graphics field, include compression of CT/MRI datasets [24], light fields [19], and 2D textures [3].

In an effort to provide a compression method that supports fast decompression to random access as well as achieves fairly high compression ratios, we have developed a compression scheme for 3D volume data whose voxels have associated RGB color (vector) attributes. In this paper, we extend our previously published work on compression of volume data with grey-scale density values [15, 16]. The new method presented in this paper employs a new encoding technique, called *zerobit encoding*, which significantly improves the decompression speeds compared to the previous results. Unlike the previous work on volume compression like [9], it is a lossy compression method, based on a 3D wavelet transform, and offers a multi-resolution representation of volume data in addition to fast decompression to random access.

The rest of this paper is organized as follows: In Section 2, we outline the three steps of our compression scheme. In Section 3, we provide details of a new zerobit encoding technique. Experimental results on three different kinds of 3D images that are found in medical imaging and real-time rendering are reported in Section 4. Finally, we present conclusions and directions for further research in Section 5.

2 Preliminaries

2.1 3D Images, Unit Blocks and Cells

In this paper, a *3D image* refers to a 3D volume dataset, defined on a regular grid, whose voxel values are 24-bit RGB colors. It is naturally constructed from a sequence of gradually varying 2D images, like movies and the Visible Human RGB cryosection images [25], or can be formulated from high dimensional sampled data, such as light fields [12, 19] and space-time volume data. It can also be built by sampling a procedurally defined solid texture in three-dimensional space [7]. In our framework, a 3D image is partitioned into a set of subvolume of size $16 \times 16 \times 16$, called *unit blocks*, which are again subdivided into subblocks of size $4 \times 4 \times 4$, called *cells*. As will be explained in the later sections, cells are the basic unit for our 3D image compression scheme. One of the desirable properties we expect a 3D image to have, is that its RGB colors have some degree of spatial coherence, at least, within cells.

A typical transform coding algorithm consists of three major stages: transform, quantization, and encoding [8, 32]. An input dataset is passed through some transformation to represent it using a different mathematical basis in

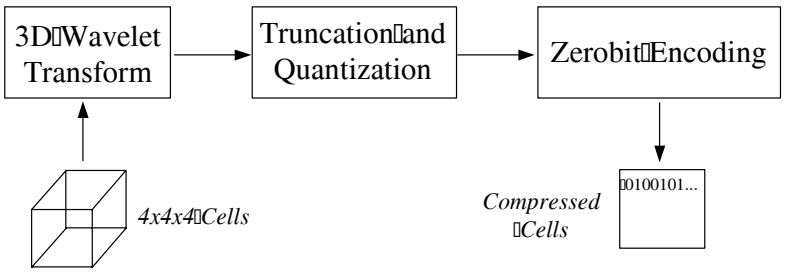


Figure 1: The three Stages of Our Compression Scheme

the hope that this new representation will reveal the correlation that exists in the data. The decorrelated coefficients produced in this stage, are quantized to produce a stream of symbols, each of which corresponds to an index of a particular quantization bin. The last stage encodes the stream of symbols and attempts to losslessly represent it as efficiently as possible. Our compression scheme is along the similar lines as is illustrated in Figure 1. In the remainder of this preliminary section, we briefly explain our adapted solutions for the first two stages, and describe our new encoding scheme, called *zerobit encoding*, for the last stage in Section 3.

2.2 3D Haar Wavelet Transform

In the transform stage, we apply a discrete Haar transform which is the simplest wavelet basis [8, 31, 33]. The Haar wavelet is simple and computationally cheap because it can be implemented by a few integer additions, subtractions, and shift operations. It yields a multi-resolution representation for discrete data in a fashion that is very natural in computer graphics. The potential for using the 3D Haar wavelet in approximation of 3D volumes was discussed earlier in [21, 22]. The basis is quite effective in applications that require fast decomposition and reconstruction though it does not perform as well in terms of filtering quality as other popular wavelet bases, such as Daubechies wavelets. In [35], Westermann tested the Haar and Daubechies bases in approximating the volume rendering integral in multi-resolution spaces for the purpose of reducing the amount of memory needed during the rendering process. As expected, it was shown that the Daubechies wavelets achieved higher compression rates than the Haar wavelet did. Their complexity, however, increased the rendering time to a great extent, implying the Haar basis is a better choice for the interactive applications where the response time is most important.

The 1D Haar wavelet transform is extended naturally into higher dimensions simply by taking tensor products of 1D filters. Consider a $2 \times 2 \times 2$ grid of a 3D image whose eight voxel colors are labeled as $c_{ijk}, 0 \leq i, j, k \leq 1$.

$$
\begin{aligned}
c_{lll} &= (c_{000} + c_{001} + c_{010} + c_{011} + c_{100} + c_{101} + c_{110} + c_{111})/8 \\
c_{llh} &= (c_{000} + c_{001} + c_{010} + c_{011} - c_{100} - c_{101} - c_{110} - c_{111})/8 \\
c_{lhl} &= (c_{000} + c_{001} - c_{010} - c_{011} + c_{100} + c_{101} - c_{110} - c_{111})/8 \\
c_{lhh} &= (c_{000} + c_{001} - c_{010} - c_{011} - c_{100} - c_{101} + c_{110} + c_{111})/8 \\
c_{hll} &= (c_{000} - c_{001} + c_{010} - c_{011} + c_{100} - c_{101} + c_{110} - c_{111})/8 \\
c_{hlh} &= (c_{000} - c_{001} + c_{010} - c_{011} - c_{100} + c_{101} - c_{110} + c_{111})/8 \\
c_{hhl} &= (c_{000} - c_{001} - c_{010} + c_{011} + c_{100} - c_{101} - c_{110} + c_{111})/8 \\
c_{hhh} &= (c_{000} - c_{001} - c_{010} + c_{011} - c_{100} + c_{101} + c_{110} - c_{111})/8
\end{aligned}
$$

Figure 2: A 3D Haar Transform (Decomposition)

The Haar transform in 3D is expressed as in Figure 2, where q_{lll} represents their average, and the remaining seven values on the left side are detail, or wavelet, coefficients, determined by filtering sequences. For example, c_{hlh} is obtained by applying the high-pass or detail filter h, the low-pass or smoothing filter l, then the high-pass filter h, along the three principal axes, respectively. As a result of an application of the 3D Haar transform, the eight coefficients are decomposed into one average and seven detail coefficients.

The original values can be reconstructed by its inverse transform (Figure 3). In our framework, coefficients in the transforms are 3-tuples whose elements, corresponding to red, green, and blue channels, respectively, are represented as three unsigned characters. Hence a vector addition/subtraction in the inverse transform is efficiently implemented in three integer addition/subtraction operations. There is a lot of redundancy among arithmetic operations in the eight reconstruction formulae. For instance, the subexpression $q_{lll} + c_{llh}$ appears four times in computing c_{000}, c_{001}, c_{010}, and c_{011}, and $c_{lll} + c_{llh} + c_{lhl} + c_{lhh}$ appears twice in restoring c_{000}, and c_{001}. By avoiding recomputing such common subexpressions, the inverse transform can be performed in 24 vector addition/subtraction operations.

Now consider a $4 \times 4 \times 4$ cell C of a 3D image. When the 3D Haar transform is applied to each of eight $2 \times 2 \times 2$ subblocks in C, eight sets of transformed coefficients, consisting of an average value and seven details, are generated. By repeating the transform to the eight averages, the cell C is further decomposed into the next coarser scale of wavelet coefficients. The 64 coefficients of C after two consecutive applications of the forward transform can be organized in a hierarchy, called *decomposition tree*, depicted in Figure 4, which consists of an average c, one set of detail coefficients $\{d_{0j}, j = 1, \cdots, 7\}$ on level 0, and eight additional detail sets $\{d_{ij}, i = 1, \cdots, 8, j = 1, \cdots, 7\}$ on level 1 that are associated with the eight $2 \times 2 \times 2$ regions. The hierarchical structure of a transformed

$$
\begin{aligned}
c_{000} &= c_{lll} + c_{llh} + c_{lhl} + c_{lhh} + c_{hll} + c_{hlh} + c_{hhl} + c_{hhh} \\
c_{001} &= c_{lll} + c_{llh} + c_{lhl} + c_{lhh} - c_{hll} - c_{hlh} - c_{hhl} - c_{hhh} \\
c_{010} &= c_{lll} + c_{llh} - c_{lhl} - c_{lhh} + c_{hll} + c_{hlh} - c_{hhl} - c_{hhh} \\
c_{011} &= c_{lll} + c_{llh} - c_{lhl} - c_{lhh} - c_{hll} - c_{hlh} + c_{hhl} + c_{hhh} \\
c_{100} &= c_{lll} - c_{llh} + c_{lhl} - c_{lhh} + c_{hll} - c_{hlh} + c_{hhl} - c_{hhh} \\
c_{101} &= c_{lll} - c_{llh} + c_{lhl} - c_{lhh} - c_{hll} + c_{hlh} - c_{hhl} + c_{hhh} \\
c_{110} &= c_{lll} - c_{llh} - c_{lhl} + c_{lhh} + c_{hll} - c_{hlh} - c_{hhl} + c_{hhh} \\
c_{111} &= c_{lll} - c_{llh} - c_{lhl} + c_{lhh} - c_{hll} + c_{hlh} + c_{hhl} - c_{hhh}
\end{aligned}
$$

Figure 3: An Inverse 3D Haar Transform (Reconstruction)

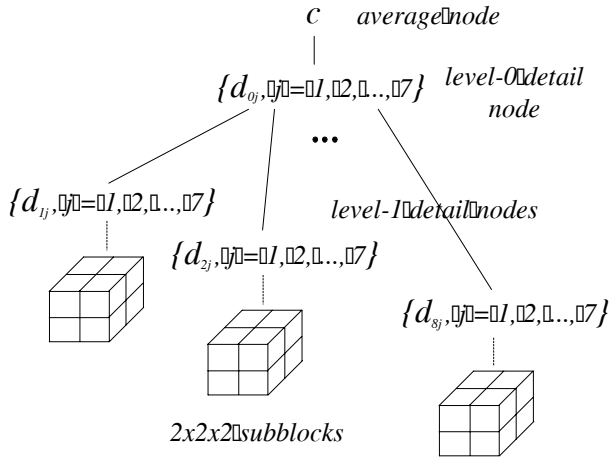


Figure 4: A Two-Level Wavelet Decomposition of a $4 \times 4 \times 4$ Cell C

cell C_{wvlt} can be represented in a set notation as $C_{wvlt} = \{c, \{d_{01}, d_{02}, \cdots, d_{07}\}, \{\{d_{11}, d_{12}, \cdots, d_{17}\}, \{d_{21}, d_{22}, \cdots, d_{27}\}, \cdots, \{d_{81}, d_{82}, \cdots, d_{87}\}\}\}$, in which c is called an *average node*, and each set of seven detail coefficients as a *detail node*.

Note that eight averages of the $2 \times 2 \times 2$ regions are implicitly represented in the decomposition tree, and are reconstructed using the average node and detail node on level 0. The original voxel colors are then reconstructed using the computed averages and eight level-1 detail nodes. In our scheme, two applications of the 3D Haar transform are thought to be enough, considering that a smaller number of applications of inverse transform results in a faster reconstruction, and that most of the data $\frac{63}{64}$ $(= 1 - (1/8^2))$ are already decomposed into detail coefficients.

2.3 Truncation of Insignificant Wavelet Coefficients

After the Haar transform, voxel values of a 3D image are decorrelated, and the energy in the original data is packed into a relatively small number of coefficients. The information in a cell C is expressed as a weighted sum of wavelet basis functions whose weights are stored in its decomposed cell C_{wvlt}. The theory behind wavelet compression tells that the best way to pick a fixed number of wavelet coefficients, making the resulting error in the L^2 norm as small as possible, is simply to select coefficients with the largest norms, and replace the rest by null values [5, 6]. The original information is thus approximated by a smaller number of nonzero wavelet coefficients.

Our compression scheme is designed to use about 1 to 7 per cent of wavelet coefficients, and truncate the remaining. Hence, after the wavelet transform and truncation, 93 to 99 per cent of coefficients become null. The level of wavelet compression is easily controlled by specifying a ratio λ of non-zero coefficients that survive the truncation. Then, a proper threshold value τ needs to be specified to cut off smaller wavelet coefficients. In our framework, we specify a target ratio $\bar{\lambda}$ of nonzero wavelet coefficients to be used, then corresponding threshold values are automatically computed. For a given $\bar{\lambda}$, τ can be computed by selecting the ($\bar{\lambda}$ · the total number of voxels)-th largest coefficient of the entire dataset.

When the 3D image is very large, as in our case, implementing the selection algorithm becomes complicated. In our work, we propose to use an approximate method to compute a threshold value that is easier to implement. Suppose that a 3D image has resolution $n_x \times n_y \times n_z$ (for convenience' sake, assume that n_x, n_y, and n_z are multiples of 16.). The 3D image is partitioned into a collection of unit blocks of size $16 \times 16 \times 16$. We first apply a Haar wavelet transform to each unit block i, and compute the ratio r_i of nonzero wavelet coefficients to the entire number 4096 (= 16^3) of coefficients in unit blocks. This ratio is a good approximate measure that indicates how complicatedly voxel colors change in the unit blocks. The total number of nonzero coefficients to be used for the entire data is thus adaptively distributed to unit blocks according to their complexity. It is reasonable that more nonzero coefficients are assigned to unit blocks with higher ratios.

For an image of size $n_x \times n_y \times n_z$, $n_x \cdot n_y \cdot n_z \cdot \bar{\lambda}$ nonzero coefficients are to be distributed to $\frac{n_x}{16} \cdot \frac{n_y}{16} \cdot \frac{n_z}{16}$ unit blocks. For unit block i, we allocate $n_i = \frac{r_i}{\sum_j r_j} \cdot n_x \cdot n_y \cdot n_z \cdot \bar{\lambda}$ coefficients, where the weight $\frac{r_i}{\sum_j r_j}$ is the relative measure of data complexity. Then the n_ith largest wavelet coefficient becomes the threshold value τ_i of the unit block, and coefficients smaller than τ_i are replaced by zeros. We find that this adaptive decision of thresholds diminishes the "blockiness" effect that often occurs when a single threshold value is applied to the entire wavelet image. Notice that the actual ratio λ is slightly different from the target ratio $\bar{\lambda}$, since unit blocks often contain

more than one wavelet coefficient having the same value as their thresholds.

2.4 Quantization of Wavelet Coefficients

During decomposition, we use floating-point numbers to calculate average and detail coefficients as correctly as possible. To achieve a high compression ratio, non-zero wavelet coefficients, surviving from the truncation, are vector-quantized. In our scheme, 24-bit coefficients are quantized into 8-bit indices with codebooks having 24-bit codewords using the median-cut algorithm [13]. While each component of RGB colors ranges from 0 to 255, its averages fall between 0 and 255, and details between -128 and 127. To take care of two different ranges, the average and detail coefficients are vector-quantized using two codebooks, called *average* and *detail codebooks*, respectively. Furthermore, since it is not space-efficient to have distinct codebooks for each cell, we have a group of cells in a contiguous region share codebooks. The experimental experience tells that a rather large contiguous region can share codebooks with only little degradation of reconstructed image quality.

3 The Zerobit Encoding Scheme

3.1 Spatial Coherence in Wavelet Coefficients

In this section, we describe the final encoding stage of our compression scheme. The encoding stage takes the symbol stream from the quantizer, and attempts to represent the data stream as efficiently as possible without loss. Popular variable length coders, such as Huffman or arithmetic coders, work very well. However, such techniques are not appropriate when individual data items must be quickly decompressed in an arbitrary sequence. An encoding technique, called zerotree encoding [32], and its variations [29, 4] have proven particularly useful in combination with wavelet transform coding, but is too slow for interactive applications. In Ihm et al. [15, 16], an effective encoding technique was proposed that supports fast random access to compressed density data. In this paper, we extend the technique to compress 3D images with RGB colors, and improve its performance, based on the property of wavelet coefficients, to achieve a lot faster random access as well as higher compression ratio without deteriorating image quality.

After the decomposition process, detail coefficients with smaller magnitude are zeroed out, and the non-zero coefficients are quantized. When 93 to 99 per cent of wavelet coefficients are cut off, only 1 to 7 per cent of 64 coefficients of a cell contain nonzero values. As a result of quantization, nonzero coefficients are represented by one-

	$\bar{\lambda}$: Target Ratio of Nonzero Coef's			
	3.0%	5.0%	7.0%	10.0%
γ_0 (Level 0)	0.230	0.111	0.075	0.061
γ_1 (Level 1)	0.949	0.843	0.720	0.546

(a) Visible Man CT Dataset

	$\bar{\lambda}$: Target Ratio of Nonzero Coef's			
	1.0%	2.0%	3.0%	5.0%
γ_0 (Level 0)	0.754	0.314	0.152	0.058
γ_1 (Level 1)	0.993	0.965	0.918	0.795

(b) Visible Man RGB Dataset

Figure 5: Ratios of Null Detail Nodes

byte indices to shared codebooks. A cell is now in the form $\bar{C}_{wvlt} = \{\bar{c}, \{\bar{d}_{01}, \bar{d}_{02}, \cdots, \bar{d}_{07}\}, \{\{\bar{d}_{11}, \bar{d}_{12}, \cdots, \bar{d}_{17}\}, \{\bar{d}_{21}, \bar{d}_{22}, \cdots, \bar{d}_{27}\}, \cdots, \{\bar{d}_{81}, \bar{d}_{82}, \cdots, \bar{d}_{87}\}\}\}$, where each element is either null or an index to codebooks. In the encoding stage, $\bar{C}_{wvlt}$ must be encoded as efficiently as possible with furthermore a guarantee of fast decoding to random access. In particular, the binary information, called the *significance map*, that denotes whether each element of $\bar{C}_{wvlt}$ is null or not, must be efficiently encoded. As shown in [32], the cost attributed to encoding the significance map represents a significant portion of the bit budget at a low bit-rate, and is likely to become an increasing fraction of the total cost as the target rate decreases.

A careful observation on the values of $\bar{C}_{wvlt}$ suggests an efficient encoding scheme that offers much faster reconstruction as well as higher compression ratio than in [15, 16]. A large portion of wavelet coefficients would be replaced by null values during truncation. Considering the usual spatial coherence in 3D images, it is very probable that the null coefficients exist in thick clusters. We observe that the ratio of null detail nodes in the decomposition trees where seven coefficients are all zero, are fairly high. Figure 5 shows sample statistics for two different datasets in which the ratios of null detail nodes are measured for several target rates. This empirical evidence reveals that the ratios of null detail nodes increase as the target ratio decreases, and in particular, that those for level 1 nodes are very high.

In our old encoding scheme [15, 16], we simply used 64 bit-flags, or eight bytes, to store all the significance information of 64 transformed coefficients regardless of their values. In the new encoding scheme, a two-stage significance map system is used: There are 9 detail nodes in a cell $\bar{C}_{wvlt}$, one on level 0 and eight on level 1. The

information whether detail nodes are null or not, is represented in 9 bits (stage 0), called *zerobits*. For each non-null detail node, the significance information of its seven detail coefficients is stored in additional seven bits (stage 1). Using this two-stage system for significance maps improves the encoding technique in two ways. First, the cost for encoding significance maps is reduced. When a detail node is not null, an extra bit for zerobit as well as 7 stage-1 bits for significance information becomes necessary. However, a large portion of detail nodes are null as observed in Figure 5, and they can be represented using only one zerobit per detail node, hence saving the encoding cost.

More significantly, the reconstruction process also becomes much faster. In order to reconstruct a voxel in a cell, the inverse Haar transform must be applied twice. Conceptually, this corresponds to traversing the decomposition tree from the root to a leaf. First, the average c and the seven details $\{d_{0j}, j = 1, 2, \cdots, 7\}$ on level 0 are used to reconstruct the average c_i ($i = 1, 2, \cdots, 8$) of the ith $2 \times 2 \times 2$ region where the voxel belongs. Then, c_i and the details $\{d_{ij}, j = 1, 2, \cdots, 7\}$ on level 1 reconstruct the voxel value. When a zerobit of a detail node indicates that the node is null, whether it is on level 0 or 1, neither decoding of its seven details nor application of the inverse transform is necessary. Since all the detail coefficients are zero, the average value is simply propagated to its child nodes without extra computations. As our experimental results show, this provides large savings in the reconstruction computation.

3.2 Zerobit Encoding

Now we describe our *zerobit encoding* technique in detail (See Figure 6). To compress a 3D image, it is first partitioned into a set of unit blocks that are subblocks of size $16 \times 16 \times 16$. A unit block contains 64 cells of size $4 \times 4 \times 4$, the basic units for our 3D image compression scheme. After going through the first two compression stages, they are encoded as follows. For each unit block we allocate one byte of memory to store the number of non-null cells in the unit block that contain at least one non-zero cell (*Number of Non-Null Cells (NNNC)*). When *NNNC* is zero, it means that all the coefficients in the $16 \times 16 \times 16$ region are zero, and this void region is encoded in one byte.

If the unit block contains at least one non-null cell, its 64 cells are enumerated in left-to-right, front-to-back, and top-to-bottom fashion, identifying non-null cells with nonnegative integers in increasing order. To indicate whether a cell is null or not, we use a *Cell Bit Flag Table (CBFT)*, made of four unsigned short integers (64 bits), in which 64 bit flags are turned off if and only if their corresponding cells are null. When a voxel is reconstructed, the bit flag of a cell that contains it, is checked to see if the cell is empty, in which case its decompressed color is

black. *CBFT* makes it possible to quickly get rid of void $4 \times 4 \times 4$ regions of non-null unit blocks in the encoding stage.

In case a cell is not null, suitable information is kept for reconstruction of voxel colors. Recall that a non-null cell $\bar{C}_{wvlt}$ has elements whose values are either null or indices to codebooks. In order to keep this information, an additional chunk of memory, called *cell information*, is allocated per non-null cell, and is stored in *Cell Information Array (CIA)*.

Our primary goal is to encode a non-null cell $\bar{C}_{wvlt}$ as efficiently as possible so that the encoding technique offers both fast random access and high compression ratio. There are two kinds of indices in $\bar{C}_{wvlt}$, one average index and possibly several non-null detail indices, which point to the *Average* and *Detail* codebooks, respectively, in *Shared Codebooks (SC)*. The average index is stored in the *average index* field (one byte) of cell information, and non-null detail indices are enumerated in a proper order in a byte stream, called *Detail Index Stream (DIS)*. Since *DIS* is shared by several non-null cells, the address of the first detail index of $\bar{C}_{wvlt}$ is remembered in a two-byte variable *detail offset* of cell information.

The positional, or significance, information of 63 detail coefficients in $\bar{C}_{wvlt}$ are encoded through zerobits and a significance map. Nine bits are necessary to store zerobits of nine detail nodes, one for level 0 node, and eight for level 1 nodes. The eight level 1 zerobits are stored in a byte. Each seven-bit significance map of non-null detail nodes is also put in a byte. In our encoding, eight bits are allocated, and the most significant bit simply indicates the level of detail nodes, 1 for level 0, 0 for level 1, though this information is not necessary in decoding. The level 1 zerobits, followed by a significance map of non-null detail nodes of $\bar{C}_{wvlt}$, are stored in another byte stream, called *Zerobit and Significance Map Stream (ZSMS)*. The address of the first byte is then stored in a two-byte variable *zerobit offset* of cell information. The position of zerobit and significance map flag of a coefficient $\bar{d}_{ij}$ can be computed quickly by a few table accesses as explained in the next subsection. Note that at most 640 bytes (64 cells in a unit block and at most 10 bytes per cell) are necessary for *ZSMS*. Since the most significant bit of *zerobit offset* is always free, we put the level 0 zerobit there.

In order to understand the structure of *ZSMS* clearly, consider the eighth non-null cell in Figure 6. In this example, the first bit of *zerobit offset* indicates that the level 0 zerobit is 1. The level 1 zerobits are always stored in the byte, indexed by the remaining bits of *zerobit offset* (the 24th byte in this case). There are three non-null detail nodes, one for level 0 and two for level 1. Their significance maps follow the level 1 zerobits, and are stored in the entries from 25 to 27.

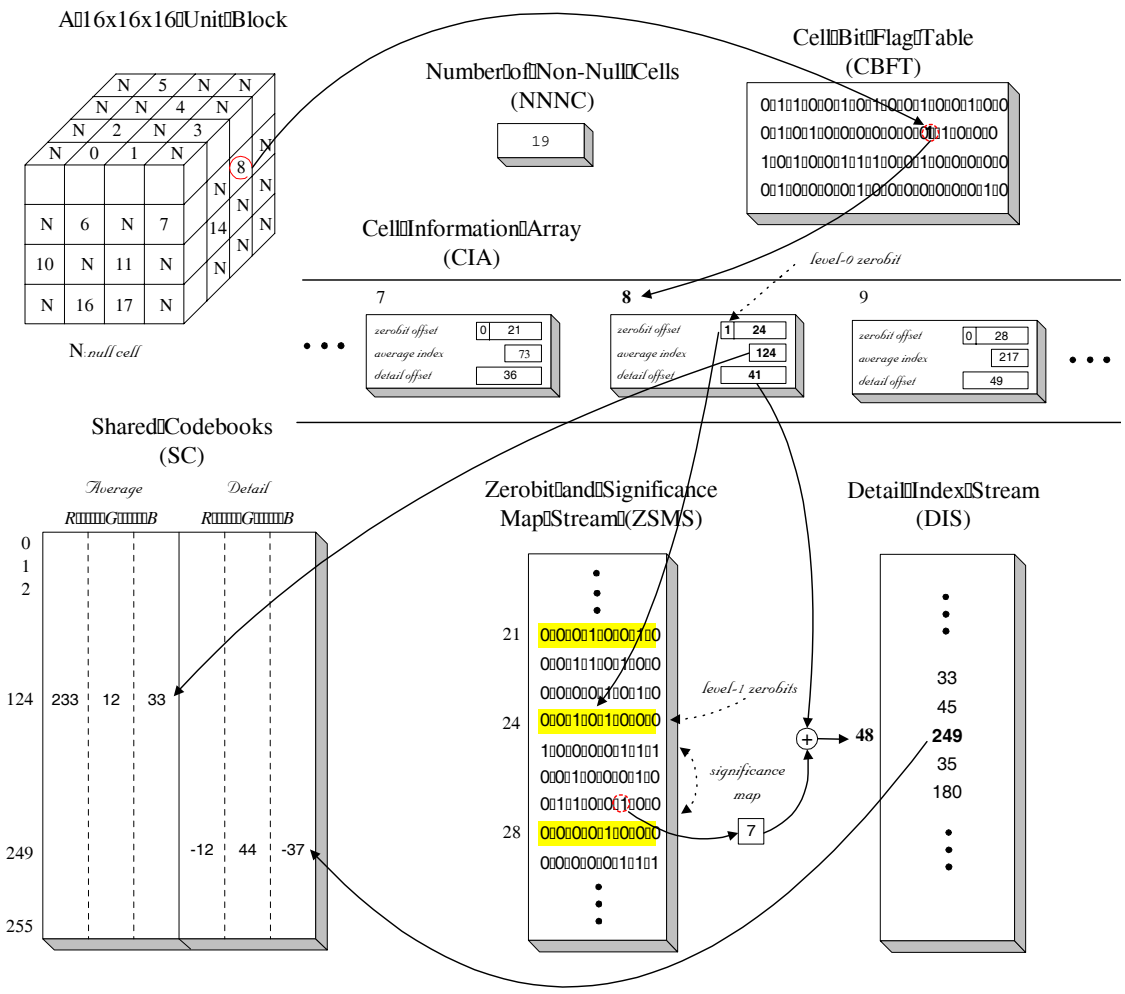


Figure 6: The *Zerobit Encoding* Scheme

3.3 Reconstruction and its Costs

The reconstruction process consists of two steps. All the coefficients necessary for reconstruction are decoded from the zerobit-encoded structure, then the inverse 3D Haar transform is applied to compute voxel colors. When the color of a voxel v is to be reconstructed, we go to the unit block that contains v. If its *NNNC* is zero, the color is simply null, that is, black ([case 1]). Otherwise, the bit flag of a cell C that contains v, is looked up in *CBFT*. In case it is zero, the color is again null ([case 2]). For voxels in a void region, say background, the reconstruction cost is just one or two variable accesses and address computations.

If the bit flag is on, it means that the voxel v belong to a non-null cell, and additional computations are necessary ([case 3]). First, the address or identification number of cell information for C in *CIA* is calculated by counting the number of bit 1 in *CBFT* that precedes it in the cell enumeration. To count the number quickly, we use a pre-computed counting table with $2^{16} = 65,536$ entries. Indexed by a two-byte unsigned short, corresponding to 16 bit flags of *CBFT*, the table returns the number of 1 bit in the index. The address can be obtained efficiently by accessing the table only a few times, 2.5 on average.

3.3.1 Decoding of Coefficients

Once we know the address of the specific cell C, we must decode all the necessary coefficients before the application of the inverse Haar transform. The decoding algorithm is described in Figure 7. Suppose that we are decoding the index of a coefficient x contained in a cell C. Once the index is decoded, its value can be obtained from *SC*. If x is an average coefficient, its index is simply found in *average index* of cell information for C ([case 3a]). If x is a detail coefficient d_{ij}, that is, the jth detail of the ith detail node, we first check if the ith detail node is null, by accessing its zerobit which can be quickly fetched using *zerobit offset*. If the zerobit is off, its index is null ([case 3b]). When d_{ij} belongs in a non-null detail node, we look up its bit flag in the significance map. The position of the significance map of the ith detail node in *ZSMS* can be easily determined using zerobits and *zerobit offset*. If the bit flag for d_{ij} is off, the index is again null ([case 3c]).

In the last case ([case 3d]) where the bit flag is on, the index for d_{ij} points to a significant wavelet coefficient in the detail codebook of *SC*. Its displacement in *DIS* can be obtained by counting the number of 1 bits in the significance map that precedes it in the detail coefficient enumeration. Finally, the sum of *detail offset* and the displacement becomes the address of the index in *DIS*. Note that finding the position of bit flags, and counting non-zero bits can be efficiently implemented in a few bit-wise operations and pre-computed table accesses.

```
/* Input:  a cell C and a coefficient x to be decoded
   Output:  the index of x to SC */
index_to_SC decode_coef (cell C, coef x) {
    if (x is an average c) {
        Fetch the index c̄ for x from average index of C;
        return (c̄); [case 3a]
    }
    else {
        Let x be d_ij;
        /* x is the jth coefficient of the ith detail node of C */
        if (the zerobit of the ith detail node is zero)
            return (NULL); [case 3b]
        else {
            Determine the position of bit flag for d_ij in significance map;
            if (the bit flag is zero)
                Return (NULL); [case 3c]
            else {
                Find the displacement in DIS by counting the number of preceding nonzero coefficients;
                Compute the address of index d̄_ij for d_ij in DIS;
                return (d̄_ij); [case 3d]
            }
        }
    }
}
```

Figure 7: The Coefficient Decoding Algorithm

Figure 6 shows an example in which the index $\bar{d}_{55}$ of detail coefficient d_{55} of a cell C, numbered 8, is being retrieved. The level 1 zerobits of this cell in the 24th byte of *ZSMS* tells that the third and fifth detail nodes are nontrivial. Furthermore, the most significant bit of *zerobit offset* indicates that the significance map for the level 0 detail node needs to be stored in *ZSMS*. Since two non-null detail nodes, one on level 0 and another on level 1, precede it, the significance map for the fifth detail node is found in the 27th (=24+1+2) byte of *ZSMS*. The fifth bit flag in the map (circled one) is on, and the displacement, or the number of 1 bit in the significance map which precedes $\bar{d}_{55}$, is counted by a few simple table accesses. The displacement 7 is then added to *detail offset* 41 to get the address 48 for $\bar{d}_{55}$ in *DIS*. Note that the most significant bits of the significance maps are used to indicate the levels, hence, are not counted.

Now, let's briefly analyze the timing costs for decoding a coefficient x. The costs for the first two cases [case 3a] and [case 3b] are trivial. When the zerobit is on, the bit flag for x in the significance map can be quickly found using a few bit-wise operations and a table access ([case 3c]). Even the most expensive case [case 3d]

requires only a few more bit-wise operations and table accesses. Recall that null detail nodes take a significance portion as empirically shown in Figure 5. This implies that the probability that either of the two cheap cases [case 3a] and [case 3b] occurs, is quite high. Furthermore, considering the fact that we usually use only 1 to 7 per cent of non-zero wavelet coefficients, 93 to 99 per cent of decoding belongs in either [case 1], [case 2], [case 3b] or [case 3c]. From this analysis, we see that decoding a coefficient in an encoded unit block is very efficient.

3.3.2 Applications of Inverse Transform

After retrieving all the necessary coefficients, the inverse 3D transform is applied which requires additional integer arithmetic operations. Our compression method offers three different reconstruction modes: voxel_mode, plane_mode and cell_mode. In voxel_mode, an individual voxel is reconstructed one by one. On the other hand, groups of voxels in a cell are simultaneously reconstructed in plane_mode and cell_mode for efficiency.

When a voxel v of a cell is to be reconstructed in voxel_mode, one average and 7 detail coefficients on level 0 are decoded. Next an appropriate one among the eight reconstruction formulae in Figure 3 is applied to compute the average of a $2 \times 2 \times 2$ subblock that contains v. Then another set of 7 detail coefficients are decoded, and another reconstruction formula is applied to compute the color of v. Note that seven vector addition/subtraction operations need to be carried out per formula, in which one vector operation amounts to 3 integer addition/subtraction operations. In total, it costs 15 decoding operations and 14 vector arithmetic operations per voxel reconstruction in voxel_mode.

Frequently, voxel access patterns exhibit some degree of locality. For example, a contiguous region might have to be decompressed, say, to show axial, coronal, or sagittal slices of the Visible Human RGB data, or retrieve a proper set of colors for image-based rendering of light field data. To enhance efficiency of voxel reconstruction, we also provide two optimized access modes. In cell_mode, the entire region of a cell becomes a reconstruction unit, and all the 64 voxels are reconstructed at the same time. In this case, 64 coefficients in an encoded cell are first decoded. Next the set of 8 reconstruction formulae are evaluated 9 times, one for computing 8 averages on level 1, and eight for computing 64 voxel colors. Although there appears to be 56 ($= 7 \cdot 8$) vector addition/subtraction operations in each application of the 8 formulae, a simple optimization technique from compiler theory that removes redundant arithmetic computations, as briefly explained in Subsection 2.2, shows that 24 operations are optimal [1]. Since the number of total operations is $9 \cdot 24$, it costs 3.375 ($= \frac{9 \cdot 24}{64}$) vector arithmetic operations and one decoding operation per voxel in cell_mode.

		Decoding (Worst Case)	Vector Operations (Worst Case)	Vector Operations (Average Case)
voxel_mode		15	14	$(1-\gamma_0)\cdot 7+(1-\gamma_1)\cdot 7$
plane_mode	x-axis	2.25	6.25	$(1-\gamma_0)\cdot 1.25+(1-\gamma_1)\cdot 5$
	y-axis	2.25	5	$(1-\gamma_0)\cdot 1+(1-\gamma_1)\cdot 4$
	z-axis	2.25	3.75	$(1-\gamma_0)\cdot 0.75+(1-\gamma_1)\cdot 3$
cell_mode		1	3.375	$(1-\gamma_0)\cdot 0.375+(1-\gamma_1)\cdot 3$

Figure 8: Reconstruction Costs per Voxel

The plane_mode provides efficient reconstruction when arbitrary 2D slices, orthogonal to principal axes, need to be decompressed. In this mode, 16 voxels in a 4×4 perpendicular plane of a cell are simultaneously reconstructed. The planes are orthogonal to either x-, y-, or z-axis, and the reconstruction costs are not symmetrical due to the filtering sequence of the 3D Haar transform. A careful analysis reveals that the decoding cost is 2.25 ($=\frac{36}{16}$) per voxel, and the cost for vector arithmetic operations per voxel is 6.25 ($=\frac{100}{16}$) for x-axis, 5 ($=\frac{80}{16}$) for y-axis, 3.75 ($=\frac{60}{16}$) for z-axis.

Figure 8 summarizes the reconstruction costs for the three modes. It should be emphasized that the analyzed costs are only for the worst case. Our zerobit encoding technique allows us to avoid unnecessary decoding and arithmetic operations. Suppose that voxels are reconstructed in, for instance, cell_mode. Recall that γ_0 and γ_1 are the ratios of null detail nodes on level 0 and 1, respectively (Figure 5). The 24 vector arithmetic operations for reconstruction of eight level-1 averages, are carried out only with probability $1-\gamma_0$, since the level-0 average simply propagates to them without arithmetic operations in case that the level-0 detail node is null. The same computation occurs eight times to reconstruct voxel colors with the probability $1-\gamma_1$. In total, the average cost of integer vector operations per voxel is $\frac{(1-\gamma_0)\cdot 24+(1-\gamma_1)\cdot 24\cdot 8}{64}=(1-\gamma_0)\cdot 0.375+(1-\gamma_1)\cdot 3$. The average costs for other modes are also listed in the figure. Due to the high ratio γ_1, the average costs are usually far less than the worst case costs.

4 Experimental Results

We have implemented the compression method described in this paper, and tested it with three different kinds of 3D images on an SGI workstation with a 195MHz MIPS R10000 CPU. The first test dataset was constructed simply by stacking up gradually varying 2D slices from medical imaging. In particular, we used a pre-cropped cryosection color images of the Visible Man data from the National Library of Medicine (NLM) [25]. The second

	Head	Thorax/Abdomen	Pelvis	Legs	Feet
Number of Slices	225	695	299	520	139
Resolutions	576×768	1760×1024	1376×864	1280×640	1344×832
Crop Offsets	(600, 160)	(0, 0)	(160, 0)	(224, 64)	(192, 64)
Sizes (Gbytes)	0.278	3.500	0.993	1.190	0.434

Figure 9: The Dimensions of the Visible Man Cryosection RGB Images

type of 3D images were built from the 4D light field data, which have been used in [19] for image-based rendering. The last type of 3D images were generated for real-time 3D texture mapping by sampling solid textures that are procedurally defined in continuous texture space.

4.1 Compression of Visible Human Cryosection RGB Images

Instead of experimenting with the original cryosection RGB images, disseminated by NLM, that require a great deal of efforts for preprocessing, we used the Visible Man dataset, commercially available from Research Systems, Inc.. The dataset contains 1,878 axial RGB images, stored in JPEG, which are partitioned into five sections of different dimensions (See Figure 9.). The uniform blue background in the original slices were removed, and then they were cropped to represent only regions of interest. This preprocessing yields some file size reduction. From this dataset, we constructed a 3D image whose size is about 6.4 GBytes.

To see the effectiveness of the zerobit encoding technique, we implemented two compression algorithms. First, the new method (NEW), presented in this paper, was implemented. Next the compression algorithm for CT/MRI data [15, 16] was extended for 3D images (OLD). In the following two subsections, we present experimental results for these two implementations. Particularly, it is demonstrated how effectively the zerobit encoding method removes unnecessary computations during reconstruction, and enhances the timing performances over the old method.

4.1.1 Compression Ratio and Visual Fidelity

Statistics for the compression ratio and quality of our compression method are given in Figure 10. We compressed at four different target ratios $\bar{\lambda} = 0.02, 0.03, 0.04$ and 0.05, in which the actual ratios after coefficient truncation are slightly different. The new method (NEW) yielded a compression ratio of 39.72 to 81.07 for the four target ratios. Compared with the method without zerobit encoding (OLD), the compression ratios increase by 10 to 15 $\%$.

		$\bar{\lambda}$: Target Ratio of Nonzero Coef's			
		2.0%	3.0%	4.0%	5.0%
Size (MB)		80.78	107.90	136.25	164.88
Compression Ratio		81.07	60.69	48.07	39.72
PSNR (dB)	total	32.84	34.27	35.58	36.61
	cropped_abdomen	27.60	28.87	29.77	30.67

Figure 10: Experimental Results on Compression Ratio and Visual Fidelity (Visible Man)

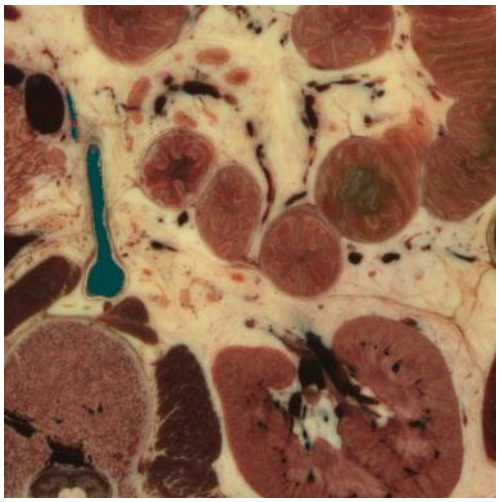
To examine distortion or difference between the original and reconstructed 3D images, we measured the mean-square peak-signal-to-noise ratio PSNR (dB) that indicates the size of the error relative to the peak value of the signal. The numbers in the row total were obtained by selecting every tenth slices from both original and compressed datasets, and computing their differences. It should be mentioned that these values are affected by the proportion of empty background region in data. To find out reconstruction quality in the interior region, we took a cropped region in the Abdomen section, which is one of the most complex parts, and evaluated the same measure (cropped_abdomen). Figure 11 shows a sample slice in the cropped region, and compares between the original and compressed images. When the target ratio is 2%, the blocky artifacts are clearly visible. When the ratio is greater than 5%, our compression technique reconstructs slices quite faithfully.

We could not find other statistics to compare ours with for the Visible Human RGB dataset. Considering that the goal of our compression scheme is to provide fast reconstruction to random access while achieving good compression ratio and reconstruction quality, we believe the zerobit encoding technique produces a favorable compression performance.

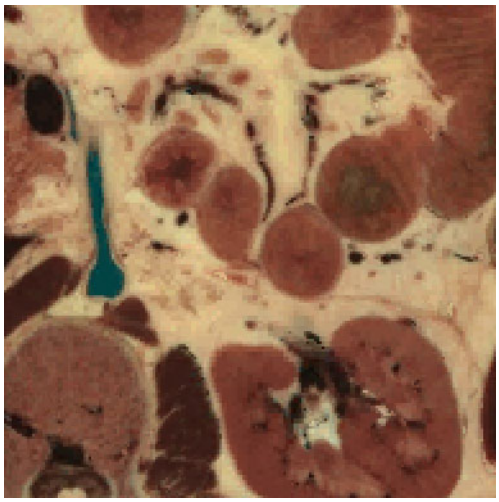
4.1.2 Voxel Reconstruction Time

To measure the reconstruction overheads, we used slices in the Thorax section, which is highly complex, hence, would be rather slow to reconstruct compared to other sections. As the analysis in Figure 8 indicates, the average reconstruction cost decreases as the ratios γ_0 and γ_1 increase. These ratios tend to become greater as the proportion of empty background region in a 3D image gets higher. We cropped the Thorax section further to produce another test dataset with a lower background proportion. Figure 12 illustrates sample slices from the two test datasets.

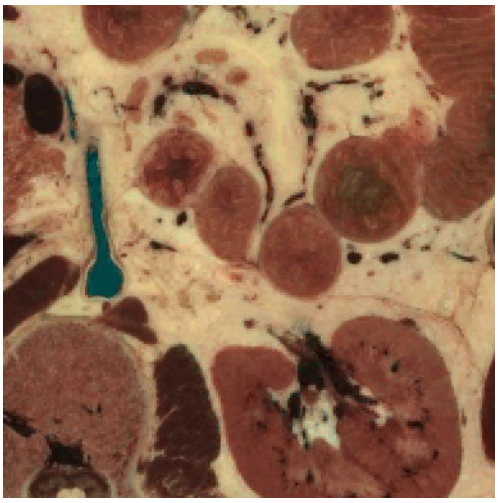
All the three reconstruction modes were tested to evaluate overheads for reconstructing voxel colors from compressed 3D image (See Figure 13.). The timings in voxel_mode 1 were taken by repeatedly fetching voxels with randomly generated indices (i, j, k). We first accessed one million voxels from the uncompressed Thorax data,



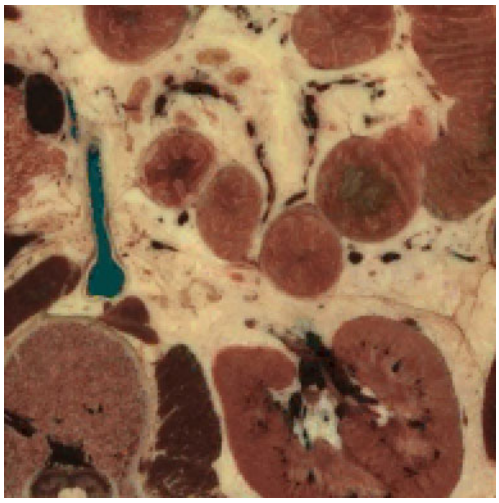
(a) Uncompressed



(b) Compressed ($\bar{\lambda} = 2\%$, Comp. Ratio = 81.1)

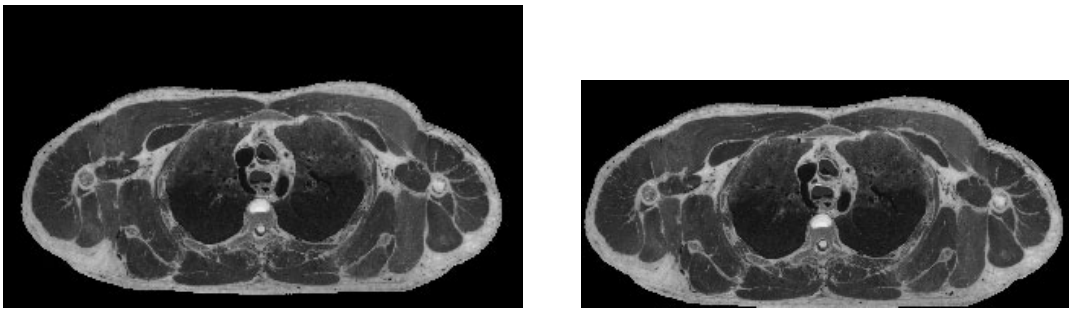


(c) Compressed ($\bar{\lambda} = 5\%$, Comp. Ratio = 39.7)



(d) Compressed ($\bar{\lambda} = 7\%$, Comp. Ratio = 29.4)

Figure 11: A Sample Slice in the Cropped Region (Abdomen)



(a) Data 1 (Background Region: 43.4%) (b) Data 2 (Background Region: 21.1%)

Figure 12: Sample Slices from Two Test Datasets (Thorax)

stored in a simple 3D array. Then the same measurement was taken using the compressed data. As indicated by the results, our zerobit encoding method provides very fast reconstruction speeds to random accesses. Even in the worst case of the experiment (Target ratio = 5.0%, Background Region = 21.1%), voxel reconstruction is only 1.4 times slower than just fetching colors from a 3D array.

In another, more practical, experiment for timing performances, we generated a series of cutting planes with arbitrary positions and orientations repeatedly, and accessed voxels, necessary for displaying the planes, until voxels are decompressed one million times (voxel_mode 2). The results show that reconstruction turns out faster than the "pure" random access (voxel_mode 1). When voxels neighboring cutting planes are reconstructed from compressed data, they are reconstructed with spatial coherence. We conjecture that the locality in memory access achieves higher hit ratios of hardware caches, and produces faster computation.

The timings for two other modes plane_mode and cell_mode were taken by reconstructing one million times randomly selected 4×4 planes and $4 \times 4 \times 4$ cells, respectively. The test results imply that the two reconstruction modes are very competitive. In most cases, a voxel reconstruction is even faster than a simple memory fetch, which requires computation of an address of 3D array. Our compression method allows a group of voxels to be decompressed simultaneously at the minimal expense. Of course, spatial data structures, such as octree, may help speed up voxel fetch operations when a 3D image is uncompressed. In this test, we just took an unstructured 3D array access as a relative criterion for measuring the reconstruction speed. The test results also demonstrate how prominently the new zerobit encoding technique improves the timing performance over the previous encoding method. Compared to the implementation without zerobit encoding (OLD), the new method (NEW) is 2.5 to 3.3 and 3.9 to 5.7 times faster in voxel_mode and cell_mode, respectively. From the timing performance results, we

	Uncompressed	$\bar{\lambda}$: Target Ratio of Nonzero Coef's				
			2.0%	3.0%	4.0%	5.0%
voxel_mode 1 (1M Voxels)	1.87	NEW	1.54	1.73	1.86	1.97
		OLD	3.95	4.39	4.71	4.94
voxel_mode 2 (1M Voxels)	1.54	NEW	0.86	1.00	1.11	1.20
		OLD	2.76	2.98	3.15	3.29
plane_mode (1M Planes)	4.32	NEW	2.35	2.70	2.99	3.28
		OLD	N/A	N/A	N/A	N/A
cell_mode (1M Cells)	11.53	NEW	3.52	4.08	4.65	5.28
		OLD	18.49	19.17	19.79	20.33

(a) Data 1 (Unit: Seconds)

	Uncompressed	$\bar{\lambda}$: Target Ratio of Nonzero Coef's				
			2.0%	3.0%	4.0%	5.0%
voxel_mode 1 (1M Voxels)	1.87	NEW	2.03	2.30	2.47	2.62
		OLD	5.22	5.84	6.31	6.65
voxel_mode 2 (1M Voxels)	1.54	NEW	1.07	1.25	1.39	1.50
		OLD	3.58	3.87	4.08	4.25
plane_mode (1M Planes)	4.32	NEW	3.07	3.55	3.97	4.37
		OLD	N/A	N/A	N/A	N/A
cell_mode (1M Cells)	11.53	NEW	4.34	5.14	5.93	6.80
		OLD	24.72	25.62	26.48	27.14

(b) Data 2 (Unit: Seconds)

Figure 13: Experimental Results on Voxel Reconstruction Time (Visible Man): The three modes voxel_mode, plane_mode, and cell_mode were tested to measure the times taken in reconstructing one million voxels, 4×4 planes, and $4 \times 4 \times 4$ cells, respectively. The timing performances of the new zerobit encoding (NEW) are compared with the old method [16] (OLD) for the various target ratios.

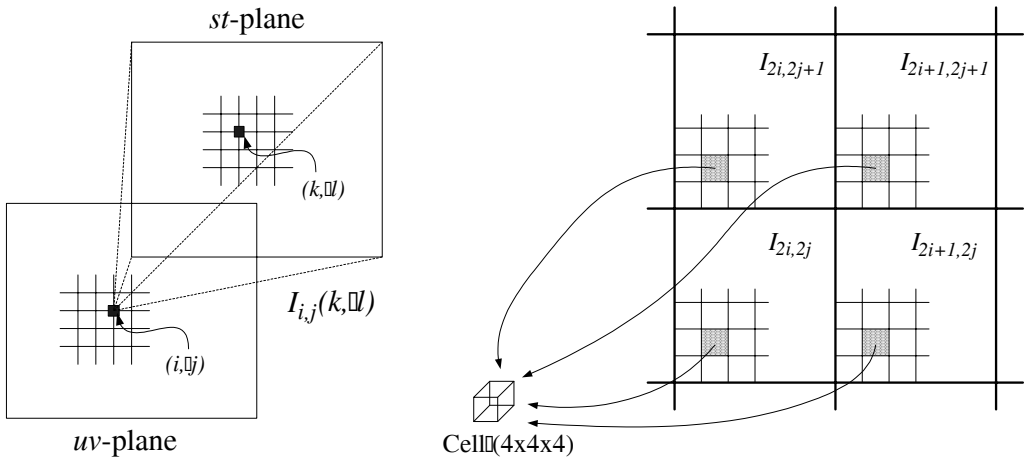
can see that the zerobit encoding method is very effective in accelerating decompression speed.

4.2 Light Field Rendering

In this subsection, we apply zerobit encoding to compress datasets produced for image-based rendering. In [19], the *light field* was defined as a radiance at a point in a given direction, and was sampled by lines determined by their intersection points with two parallel planes. The two points, parameterized by (u, v) and (s, t), respectively, define a point in 4D space, hence the discrete representation of light field can be regarded as a 4D RGB image. The same representation was independently defined as the *lumigraph* in [12]. Light fields are usually very large in size, and must be compressed. They proposed to use vector quantization [19] and JPEG [12] to compress the light field. Recently, different compression schemes were presented to improve compression efficiency [18, 36].

In order to use our 3D compression technique in image-based rendering, 4D sampled light field datasets were reformulated into 3D images. Assume that we have a sampled light field whose resolution is $n_u \times n_v$ and $n_s \times n_t$ in the uv-plane (front) and st-plane (back), respectively. The 4D function $LF(i, j, k, l)$ is usually produced by rendering, or taking a picture of, a set of 2D images $I_{i,j}(k, l)$ with the center of projection of the camera at the sample (i, j) on the uv-plane (Figure 14(a)). The set of 2D images is partitioned into groups of four adjacent images $I_{2i,2j}$, $I_{2i+1,2j}$, $I_{2i,2j+1}$, and $I_{2i+1,2j+1}$, $0 \leq i < \frac{n_u}{2}, 0 \leq j < \frac{n_v}{2}$ (Figure 14(b)). Notice that there exists a high degree of inter-pixel coherence between adjacent images in the same group. The $n_s \times n_t$ images are then subdivided into tiles of size 4×4, and the four corresponding tiles, each from the four adjacent images, form $4 \times 4 \times 4$ cells in a reformulated 3D image. In this way, 4D light fields are converted into 3D images. Although we lose data coherence in one dimension, cells in the 3D image still keep the coherence that exists in the three remaining dimensions, and this makes our 3D image compression technique performs well.

We compressed rearranged light fields with our method, and compared its performance with the vector quantization technique, employed in [19]. In this experimentation, we used the source programs and datasets of the LightPack package, publicly available at [20]. Figure 15 compares the performance of two compression methods on two representative datasets buddha and dragon whose resolutions are $32 \times 32 \times 256 \times 256$ (192MBytes). While the vector quantization method yielded compression rates 21.79 and 20.18 for buddha and dragon, our method produced higher ratios of 44.51 to 91.11 and 38.21 to 83.03, respectively (Figure 15(a)). These rates exclude the gzip compression, that could follow both compression methods for efficient storage and transmission as in [19]. The PSNR data shows that the reconstructed image quality for the light field datasets is almost the same for the two



(a) A Sampled Light Field

(b) Cell Construction

Figure 14: A 3D Formulation of 4D Light Fields

compression methods when about 2% and 5% of coefficients are used in our method for the buddha and dragon datasets, respectively.

In order to examine the timing performance, we measured the image-based rendering time, spent on displaying 76 frames of 382×382 pixels with gradually varying viewing parameters. Considering the way 3D images are rearranged from 4D light fields, it is natural and efficient to reconstruct compressed data in plane_mode. As explained in Subsection 3.3.2, it is the fastest when 4×4 planes perpendicular to the z-axis are decompressed. Hence, we stacked up four 4×4 tiles during rearrangement so that they are orthogonal to the z-axis. Two cases of bilinear interpolation on the st-plane (st-lerp) and quadralinear interpolation on both uv- and st-planes ($uvst$-lerp) were tested (Figure 15(b)). The table shows our method is faster for both datasets in most cases. Note that the reconstruction cost per voxel for vector quantization is very cheap since voxels are decompressed simply by accessing codebooks. It must be cheaper than our compression scheme on average. In our implementation, we maintain a small set of cache blocks that hold 4×4 planes, and all the 16 voxels in 4×4 planes are quickly decompressed into the cache at the same time. There is a lot of inter-frame coherence for light field rendering, and this property, coupled with our cache scheme, results in faster rendering on the whole. When nearest samples without interpolation are taken during image-based rendering, our method yielded frame rates 39 and 52 for the two datasets.

Figure 16 presents sample images obtained by applying the image-based rendering technique to the compressed

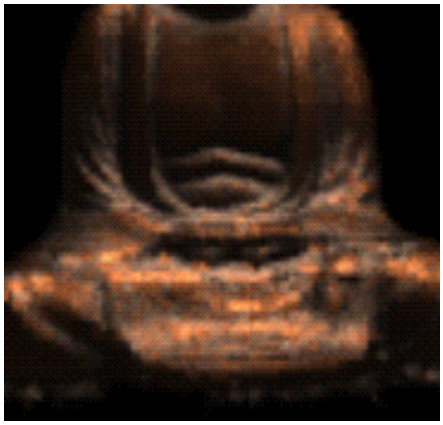
		Vector Quantization	Zerobit Encoding (Target Ratio $\bar{\lambda}$)			
			2.0%	3.0%	4.0%	5.0%
buddha	Size (MB)	8.81	2.11	2.90	3.63	4.31
	Comp. Ratio	21.79	91.11	66.26	52.89	44.51
	PSNR (dB)	38.00	39.26	41.70	43.63	45.18
dragon	Size (MB)	9.52	2.31	3.15	4.09	5.02
	Comp. Ratio	20.18	83.03	60.87	46.99	38.21
	PSNR (dB)	35.58	31.00	32.17	33.37	34.40

(a) Compression Ratio and Visual Fidelity

		Vector Quantization	Zerobit Encoding (Target Ratio $\bar{\lambda}$)			
			2.0%	3.0%	4.0%	5.0%
buddha	st-lerp	9.46	13.60	13.60	13.60	13.60
	$uvst$-lerp	2.68	2.99	2.98	2.98	2.98
dragon	st-lerp	17.55	24.60	24.44	24.20	23.97
	$uvst$-lerp	5.66	5.74	5.71	5.66	5.62

(b) Rendering Time (Frames per Second)

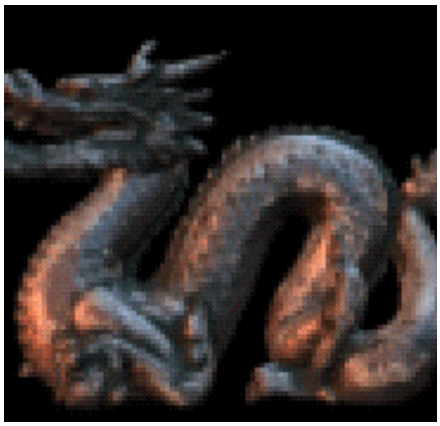
Figure 15: Comparisons with Vector Quantization on Light Field Datasets: The zerobit encoding scheme is compared with the vector quantization method used in [19]. The sizes (Size) and compression ratios (Comp. Ratio) in (a) exclude the gzip compression, that could follow both compression methods for efficient stroage and transmission. The rendering times in (b) were obtained by averaging the image-based rendering times, spent on displaying 76 frames of 382×382 pixels.



(a) buddha: VQ (8.81MB)

(b) buddha: ZE ($\bar{\lambda} = 3\%$, 2.90MB)

(c) buddha:ZE ($\bar{\lambda} = 5\%$, 4.31MB)



(d) dragon: VQ (9.52MB)

(e) dragon: ZE ($\bar{\lambda} = 3\%$, 3.15MB)

(f) dragon: ZE ($\bar{\lambda} = 5\%$, 5.02MB)

Figure 16: Sample Rendered Images (st-lerp)

datasets. It is obvious that zerobit encoding achieves both higher compression rates/image quality and faster rendering even though the 3D versions of light fields, rearranged for zerobit encoding, can not fully exploit the data redundancy that exists in all four dimensions of the original data. We expect that the future extension of the current 3D zerobit encoding technique to 4D space will provide faster frame rates for light field rendering while having much higher image quality.

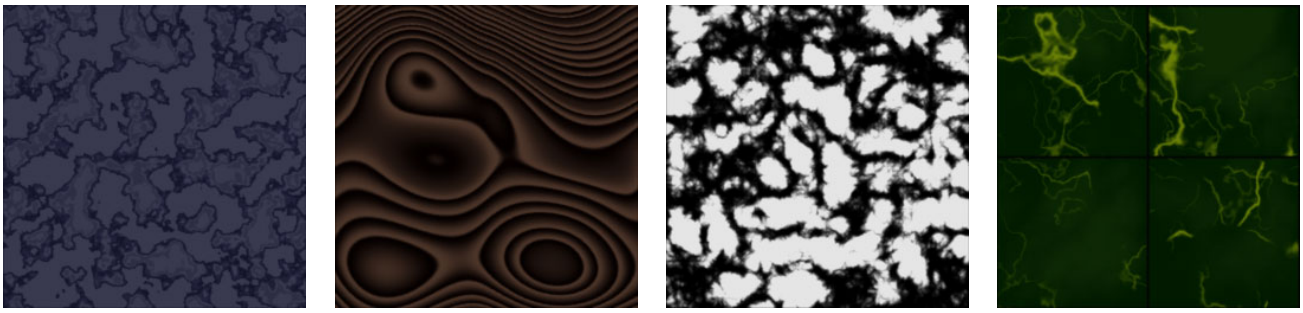
4.3 3D Texture Mapping for Real-Time Rendering

As the last example, we describe briefly how we applied the zerobit-encoding technique to real-time solid texture mapping which often requires a prohibitively large amount of texture memory[1]. Two-dimensional texture mapping has proved very useful in adding realism in rendering, however, it often suffers from the limitation that it is not easy to wrap 2D patterns, without visual artifacts, onto the surface of objects with complicated shapes [14]. As an attempt to alleviate the computational complications of wrapping as well as to resolve the visual artifacts, Peachey [27] and Perlin [28] proposed the use of space filling 3D texture images, called *solid textures*. Many of the textures found in nature such as wood and marble, are easily simulated with solid textures that map three-dimensional object space to color space [7].

Solid textures are usually synthesized procedurally instead of painting or digitizing them. They are often based on mathematical functions or programs that take 3D coordinates of points as input, and compute their corresponding texture colors. The evaluation is generally performed on the fly during the rendering computation. While procedural models provide a compact representation of textures, evaluating procedures as necessary during texture mapping leads to slow rendering. Explicitly storing sampled textures in dedicated memory, and fetching texture colors as necessary, as in the current graphics accelerator supporting real-time texture mapping, can generate images faster, however, they tend to take up a large amount of texture memory. For example, when a 3D RGB texture with resolution $256 \times 256 \times 256$ is represented in one byte per color channel, it requires 48 MBytes of texture memory. Storing more elaborate textures with higher resolution, say, $512 \times 512 \times 512$, which amount to 384 MBytes per RGB texture, would be prohibitive even to the most advanced rendering systems. To make 3D texture mapping practical, efficient solutions for handling potentially huge textures of non-trivial resolutions need to be devised.

As one solution, we propose to compress 3D textures using zerobit encoding. The idea of rendering directly from compressed textures was presented in [3], where they used vector quantization to compress 2D textures in

[1] The details on this compression-based 3D texture mapping for real-time rendering are described in [2].



(a) Bmarble (b) Wood (c) Eroded (d) Gmarbpol

Figure 17: Sample Slices from the four 3D Textures

simple or mip-map form. The key point in our texture mapping scheme is to extract only the necessary portions from the discrete 3D texture map, then selectively compress them in compact form using zerobit encoding. In the implementation, a 3D texture image in texture space is subdivided into subblocks of size $4 \times 4 \times 4$, called *texture cells*, which coincide with *cells*, the basic compression units of zerobit encoding. Then each polygon on the boundary of a polygonal object is 3D-scan-converted in texture space to find all the texture cells that intersect with the surface of the solid object. Notice that texels in the selected texture cells contain all the texture information necessary for rendering. The cells that are not chosen are replaced by null cells, that is, cells with black color. By preserving only nearby texture colors surrounding the surface of an object in this intermediate stage, a large portion of texture data is removed to alleviate the potential prohibitive storage requirement. The selected texture cells usually take only a small percentage of the original texture data. The null cells still exist in the texture map in this stage, and the uncompressed texture size remains the same. However, the spatial coherence additionally created by null cells allows the zerobit encoding scheme to compress the 3D texture very efficiently.

For our experiments, we generated four different 3D textures of resolution $256 \times 256 \times 256$ (48MBytes) and applied them to four polygonal models with various shapes and sizes (Figure 17 and 18). To reduce the texture memory sizes, the scan-converted texture images were compressed using zerobit encoding with various target ratios $\bar{\lambda}$. In Figure 19 (a), we compare sizes and compression rates for various cases. Observe that it took only a small amount of memory, ranging from 188 KBytes to 540 KBytes. Considering that the size of the original textures is 48 MBytes, we see that the proposed texture mapping scheme achieves very high compression rates through texture cell selection and zerobit encoding.

To find out how zerobit encoding affects rendering in the point of computation time and image quality, we

have implemented the compression-based 3D texture mapping scheme by extending the MESA 3D Graphics Library [26]. MESA is a publicly available OpenGL implementation, and its current version 3.0 supports 3D texture mapping with uncompressed texture images only. Figure 18 shows sample images rendered with the linear filter from the textures compressed with a target ratio of 10%. In Figure 20, we cropped and enlarged a portion of the Bunny-with-Eroded images twice to make the compression artifacts more visible. When the target ratio is 3%, the blocky artifacts are clearly visible, but most features are still preserved well enough for many real-time applications such as 3D games and animation. When they are compressed with a ratio higher than 10%, the texture-mapped images are almost free of aliasing artifacts.

We also measured the computation time, spent on rendering 54 frames of 512×512 pixels, without hardware graphics acceleration, with incrementally varying viewing parameters. They include all computations for rendering including 3D texture mapping, view parameter setting, and displaying the final images. Figure 19 (b) reports the average time per frame in seconds for three different rendering modes. In the table, our compression-based texturing scheme was compared with texture mapping without compression to evaluate overheads for fetching texels from zerobit-encoded textures. Both nearest and linear filtering methods were tested whose performances are presented in the "NEAR" and "LINE" fields, respectively. As indicated by the test results, the fast random access ability of our compression method results in a small impact on rendering time. We observe only a 14 percent and a 15 percent increase on rendering time on average for the nearest and the linear filters, respectively. Observe that the linear filtering method takes, for instance, 0.37 second to render a Teapot image from the uncompressed texture of size 48 Mbytes. On the other hand, the same filtering takes 0.44 second to produce the Teapot image with few visual artifacts from the compressed texture of size 268 KBytes ($\bar{\lambda}$ = 10%). The benefit from zerobit encoding is evident, and is critical in particular when texture memory resource is rather limited. From the experiments, we conclude that the zerobit encoding technique is very effective for compressing 3D textures. Notice that the zerobit-encoded 3D textures implicitly represent three levels of details. As well as it compresses textures well, its capability of multi-resolution representation makes it easy to implement 3D mip-maps using only a small amount of texture memory. The reduction images on the next three levels could be stored in another zerobit-encoded structure, or could be just stored compactly in simple 3D arrays (Less than 110 KBytes of texture memory is necessary for storing all the lower resolution images on level $3, 4, \cdots, 8$ of a 256^3 RGB texture.). Refer to [2] for test results on textures with higher resolution $512 \times 512 \times 512$ whose sizes are 384 Mbytes.

5 Concluding Remarks

In this paper, we have presented a new 3D RGB image compression scheme designed for interactive real-time applications. The experimental results on three different 3D images from medical imaging, image-based rendering, and 3D texture mapping show that it provides fast random access to compressed data in addition to achieves fairly high compression ratios. It is easy to implement, and provides a hierarchical representation with three levels of detail. It is suitable for applications wherein data are accessed in somewhat unpredictable fashions, and fast decompression is critical. Our method will be used as another candidate, along with the vector quantization technique, for a compression tool supporting real-time performance.

Our compression method, based on the Haar wavelets, neither yields as good compression rates nor offers as high fidelity in a reconstruction as JPEG or multi-tap Daubechies wavelets do. It has been designed to compromise between compression rates and random decoding speeds, and is geared towards good performance for various 3D RGB images whose voxel colors have some extent of coherence within, at least, each $4 \times 4 \times 4$ grid, as observed in most volume datasets found in computer graphics and visualization. The Haar filters are not powerful enough to handle 3D images with very sophisticated or random variations of voxel colors. For such datasets, better filters, such as Daubechies wavelets, must be adopted, but only at increased costs for random decoding as observed in [35].

A primary motivation for this research was to develop a compression technique that can be employed effectively in real-time applications that must handle large datasets, made of samples taken in three- or higher-dimensional space. We are currently extending the 3D compression technique to four-dimensional volume data. Once effective compression schemes for arbitrary dimensional datasets are developed, the high memory requirement, which often troubles many volume graphics algorithms will be alleviated to a great extent.

Acknowledgements

We would like to thank the Stanford University Computer Graphics Lab. for the source programs and light field datasets in the LightPack package. The MESA 3D Graphics Library is an OpenGL implementation by Brian Paul. We also wish to thank Viewpoint, the Stanford University Computer Graphics Lab., the RenderMan software [34], and the Blue Moon Rendering Tools (BMRT) for their public polygonal models and surface shaders. This work was supported in part by a University Foundation Research Program 2000 grant from the Ministry of Information &

Communication of Korea, an NSF grant (KDI-DMS-9873326), a NASA grant (NCC 2-5276) and a Sandia/LLNL grant (BD-4485).

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] C. Bajaj, I. Ihm, and S. Park. Compression-based 3D texture mapping for real-time rendering. *Graphical Models*, 62(6):391–410, November 2000.

[3] A. Beers, M. Agrawala, and N. Chaddha. Rendering from compressed texture. *Computer Graphics (Proc. SIGGRAPH '96)*, pages 373–378, 1996.

[4] Y. Chen and W. Pearlman. Three-dimensional subband coding of video using the zero-tree method. In *Proceedings of SPIE - Visual Communications and Image Processing '96*, pages 1302–1312, Orlando, March 1996.

[5] C. K. Chui. *An Introduction to Wavelets*. Academic Press Inc., 1992.

[6] I. Daubechies. *Ten Lectures on Wavelets*. SIAM, 1992.

[7] D.S. Ebert (Editor), F.K. Musgrave, D. Peachey, K. Perlin, and S. Worley. *Texturing and Modeling: A Procedural Approach*. AP Professional, 1994.

[8] A. Fournier, editor. *Wavelets and Their Applications in Computer Graphics*. ACM SIGGRAPH, 1995. ACM SIGGRAPH '95 Course Notes.

[9] J.E. Fowler and R. Yagel. Lossless compression of volume data. In *1994 Symposium on Volume Visualization*, pages 43–50, October 1994.

[10] A. Gersho and R.M. Gray. *Vector Quantization and Signal Compression*. Kluwer Academic Publishers, 1992.

[11] R. Gonzalez and R. Woods. *Digital Image Processing*. Addison-Wesley, 1993.

[12] S. Gortler, R. Grzeszczuk, R. Szeliski, and M. Cohen. The Lumigraph. *Computer Graphics (Proc. SIGGRAPH '96)*, pages 43–54, 1996.

[13] P. Heckbert. Color image quantization for frame buffer display. *Computer Graphics (Proc. SIGGRAPH '82)*, pages 297–307, 1982.

[14] P. Heckbert. Survey of texture mapping. *IEEE Computer Graphics and Applications*, 6(11):56–67, 1986.

[15] I. Ihm and S. Park. Wavelet-based 3D compression scheme for very large volume data. In *Proceedings of Graphics Interface '98*, pages 107–116, Vancouver, Canada, June 1998.

[16] I. Ihm and S. Park. Wavelet-based 3D compression scheme for interactive visualization of very large volume data. *Computer Graphics Forum*, 18(1):3–15, 1999.

[17] A. Kaufman, editor. *Volume Visualization*. IEEE Computer Society Press, 1991.

[18] M. Kiu, X. Du, R. Moorhead, D. Banks, and R. Machiraju. Two-dimensional sequence compression using MPEG. In *Visual Communication and Image Processing '98*, pages 914–921, January 1998.

[19] M. Levoy and P. Hanrahan. Light field rendering. *Computer Graphics (Proc. SIGGRAPH '96)*, pages 31–42, 1996.

[20] LightPack: Light Field Authoring and Rendering Package. $http://graphics.stanford.edu/software$, 1996.

[21] S. Muraki. Approximation and rendering of volume data using wavelet transforms. In *Proceedings of Visualization '92*, pages 21–28, Boston, October 1992.

[22] S. Muraki. Volume data and wavelet transforms. *IEEE Computer Graphics and Applications*, 13(4):50–56, 1993.

[23] G. M. Nielson, H. Hagen, and H. Müller. *Scientific Visualization: Overviews, Methodologies, and Techniques*. IEEE Computer Society Press, 1997.

[24] P. Ning and L. Hesselink. Fast volume rendering of compressed data. In *Proceedings of Visualization '93*, pages 11–18, San Jose, October 1993.

[25] NLM. $http://www.nlm.nih.gov/research/visible/visible\_human.html$, 1997.

[26] B. Paul. *The Mesa 3D Graphics Library*. $http://www.mesa3d.org$, 1999.

[27] D.R. Peachey. Solid texturing of complex surfaces. *Computer Graphics (Proc. SIGGRAPH '85)*, 19(3):279–286, 1985.

[28] K. Perlin. An image synthesizer. *Computer Graphics (Proc. SIGGRAPH '85)*, 19(3):287–296, 1985.

[29] A. Said and W. Pearlman. Image compression using the spatial-orientation tree. In *Proceedings of IEEE Intl. Symp. on Circuits and Systems*, pages 279–282, Chicago, May 1993.

[30] K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann Publishers, Inc., 1996.

[31] P. Schröder and W. Sweldens, editors. *Wavelets in Computer Graphics*. ACM SIGGRAPH, 1996. ACM SIGGRAPH '96 Course Notes.

[32] J. M. Shapiro. Embedded image coding using zerotrees of wavelet coefficients. *IEEE Transactions on Signal Processing*, 41(12):3445–3462, December 1993.

[33] E. Stollnitz, T. DeRose, and D. Salesin. *Wavelets for Computer Graphics: Theory and Applications*. Morgan Kaufmann Publishers, Inc., 1996.

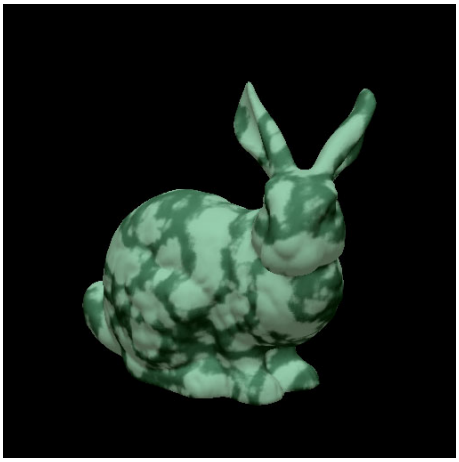
[34] S. Upstill. *The RenderMan*$^{\mathrm{TM}}$ *Companion*. Addison-Wesley, 1990.

[35] R. Westermann. A multiresolution framework for volume rendering. In *1994 Symposium on Volume Visualization*, pages 51–58, October 1994.

[36] C. Zhang and J. Li. Compression of lumigraph with multiple reference frame (MRF) prediction and just-in-time rendering. In *IEEE Data Compression Conference*, pages 253–262, March 2000.

(a) Teapot with Bmarble (1,152 faces)

(b) Dragon with Wood (12,078 faces)



(c) Bunny with Eroded (69,451 faces)

(d) Head with Gmarbpol (203,544 faces)

Figure 18: Four Renderings of Polygonal Models with 3D Textures

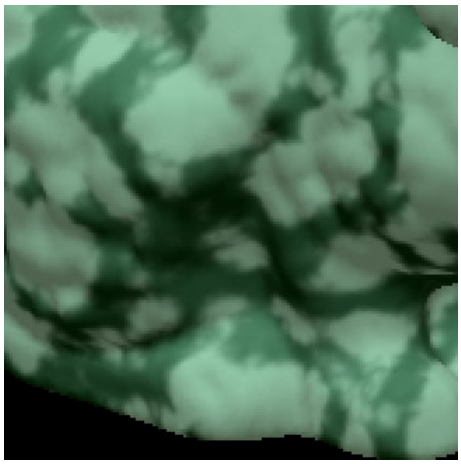
Object & Texture	Target Ratio $\bar{\lambda}$	Size (KB)	Comp. Ratio
Teapot with Bmarble	3%	188	261.5
	5%	224	219.4
	10%	268	183.4
Dragon with Wood	3%	192	256.0
	5%	232	211.9
	10%	308	159.6
Bunny with Eroded	3%	280	175.5
	5%	356	138.1
	10%	492	99.9
Head with Gmarbpol	3%	332	148.1
	5%	420	117.0
	10%	540	91.0

(a) Sizes of Compressed Textures

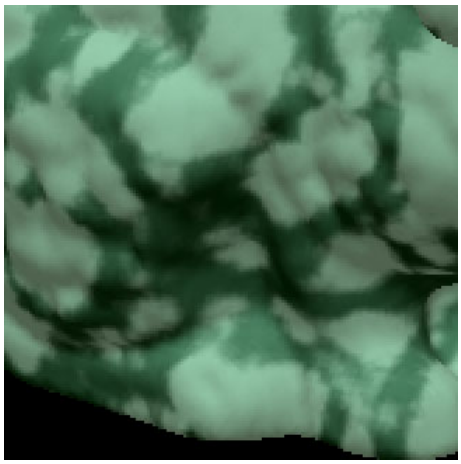
Object & Texture	Target Ratio $\bar{\lambda}$	NEAR	LINE
Teapot with Bmarble (1,152 faces)	uncomp.	0.13	0.37
	3%	0.14	0.42
	5%	0.15	0.43
	10%	0.16	0.44
Dragon with Wood (12,078 faces)	uncomp.	0.45	0.89
	3%	0.50	0.98
	5%	0.52	1.00
	10%	0.55	1.04
Bunny with Eroded (69,451 faces)	uncomp.	1.39	1.77
	3%	1.56	2.04
	5%	1.60	2.13
	10%	1.66	2.21
Head with Gmarbpol (203,544 faces)	uncomp.	3.68	4.51
	3%	3.89	4.85
	5%	3.94	4.90
	10%	4.00	5.03

(b) Rendering Time per Frame (Seconds): NEAR - 3D Texture Mapping with the Nearest Filter, LINE - 3D Texture Mapping with the Linear Filter

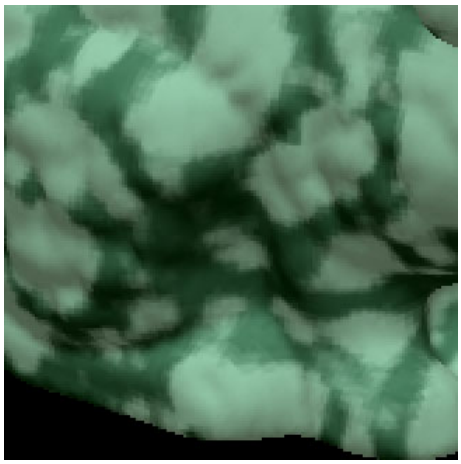
Figure 19: Experimental Results on Four 3D Textures and Objects



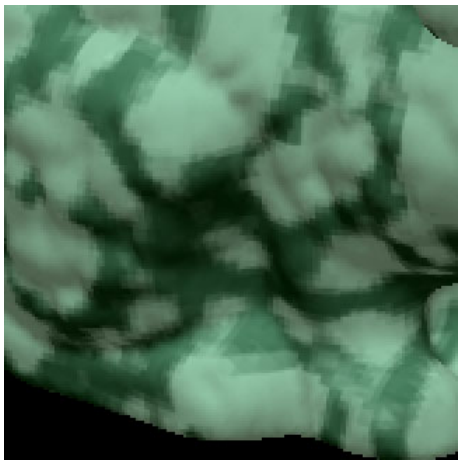
(a) Uncompressed (48MB)



(b) Compressed ($\bar{\lambda} = 10\%$, 492KB)



(c) Compressed ($\bar{\lambda} = 5\%$, 356KB)



(d) Compressed ($\bar{\lambda} = 3\%$, 280KB)

Figure 20: Aliasing Artifacts of Compression-Based 3D Texture Mapping (2X)