EUROGRAPHICS-IEEE VGTC Symposium on Visualization (2005)
K. W. Brodlie, D. J. Duke, K. I. Joy (Editors)

High Quality Rendering of Compressed Volume Data Formats

Nathaniel Fout Hiroshi Akiba Kwan-Liu Ma Aaron E. Lefohn Joe Kniss
Department of Computer Science
The University of California, Davis
Department of Computer Science
The University of Utah

Abstract

Rendering directly from packed or compressed volume data formats using graphics hardware has advantages in terms of memory consumption and bandwidth, but results in lower-quality images due to the prohibitive cost of performing interpolation and gradient-based shading on the reconstructed data. The problem with the existing method lies in its close coupling of decompression and interpolation. We demonstrate that deferred filtering overcomes this problem by using a two-pass decompression and rendering strategy. With this method interpolation and gradient calculations are very efficient, allowing high quality rendering directly from packed or compressed volume data. We evaluate the cost of creating interpolated, gradient-shaded renderings using traditional on-the-fly decompression and deferred filtering, and show that deferred filtering can provide up to twenty times speed-up for high quality rendering.

Catagories and Subject Descriptors: I.3.1: [Computer Graphics]: Picture and Image Generation – *Graphics processors*; I.3.3 [Computer Graphics]: Picture and Image Generation – *Viewing algorithms*; E.4 [Coding and Information Theory]: Data Compaction and Compression

1. Introduction

Volume rendering of large data sets is the subject of ongoing research in the field of visualization, and with larger and larger volume data being generated the problem will not disappear soon. On one hand we have time-varying volumes, wherein measurements or computer simulations of dynamic phenomena may provide hundreds of time steps. On the other hand we have large static volumes resulting from high resolution acquisition devices and/or the need to visualize fine features in the spatial domain. To compound the problem many data sets store multiple properties or variables for each voxel. In fact the presence of any or all of these properties presents challenges in volume rendering due to the increase in data available for visualization.

Research into large data visualization has offered various potential solutions to handling large data in the context of volume rendering, and one of these is data reduction via compression. Data compression is a well-established field offering a plethora of techniques to reduce the size of data, either losslessly or lossily, and though a large part of this work targets image compression most of these techniques can be extended in a straightforward manner to volume compression as well. Several options exist when combining compression and volume rendering, as described by Yang [Yan00]. These include decompression prior to rendering, rendering on-the-fly during decompression, decompressing on-the-fly during rendering, and rendering directly from the compressed volume.

In this work we show how a recently proposed algorithm, deferred filtering [LKHW03, KLF05], dramatically improves the performance for decompressing on-the-fly during rendering. In the recent past several compression options have been presented which enable decompression on-the-fly [KE02, LK02, BCF03, SW03], but all of these suffer from one key problem: in order to support continuous reconstruction from the compressed representation either data replication or costly manual interpolation is necessary (in some cases, for instance whenever vector quantization is used, only the latter option is available). We demonstrate that the reason for this is the close coupling of decompression and interpolation implicit in previous methods. Furthermore, we demonstrate that deferred filtering overcomes this problem, thereby enabling efficient rendering of compressed volume data while taking advantage of native hardware interpolation. The improved efficiency of decompression afforded by deferred filtering allows for a) continuous reconstruction from the compressed volume, b) calculation of gradients on-the-fly for shading, and c) more flexibility in the compression algorithm.

2. Related Work

There are numerous applications where compression is useful, and while many compression methods are general in their application, tailoring the compression method to the application is a standard technique to achieve improved performance. For instance, MPEG compression of video relies not just on compression of separate frames, but also on inter-frame correlation with motion compensation. Similarly, compression of volume data can be tailored to the application of volume rendering. The essential requirements of compression, if it is to be closely coupled with volume rendering, are described by Ning and Hesselink [NH92] as fast, direct, random access to voxels.

As can be observed in many of the existing methods, the degree to which the former principles are followed largely determines the amount of interactivity possible when rendering from compressed volume data.

Rendering on-the-fly from compressed volume data is efficient in that from the renderer's perspective the memory consumption and bandwidth are reduced. Ning and Hesselink [NH93] compressed volumes with vector quantization and achieved very efficient rendering by ray casting the relatively small codebook once and reusing the results. In this way vector quantization not only compresses the data but also increases the efficiency of rendering in some cases. Yeo and Liu [YL95] compressed volumes with a JPEG-like method and rendered by decompressing blocks as needed. Rendering was accelerated by extracting only the DC coefficient for nearly homogeneous blocks. A more recent technique by Guthe et al. [GWGS02] creates a block-wise hierarchical decomposition or octree of volume data with entropy encoding of wavelet coefficients. During rendering projective classification, priority-based decompression, and block caching are used to efficiently render large data sets interactively.

With the recent introduction of programmability in the graphics pipeline, several on-the-fly decompression methods have been developed which allow rendering directly from compressed data stored in graphics memory. In texture packing non-empty regions of the volume are packed into smaller textures. This method achieves a kind of lossless compression and allows access to packed data, both static [KE02, LK02] and dynamic [LKHW03, LKHW04], through an index texture. A similar technique is described by Binotto et al. [BCF03] for time-varying data. Coherence between time steps allows reuse of packed blocks, and a refinement structure is used to take advantage of homogeneous regions and to provide efficient rendering. Schneider and Westermann [SW03] use a hierarchical vector quantization scheme based on a Laplacian decomposition to compress volumes, with decompression on-the-fly in graphics hardware. This method achieves better performance than using simple vector quantization but uses a slightly more complex decompression scheme.

Various other compression methods have been applied to volume data. Lossless compression [FY94] as well as lossy methods based on fractal encoding [CHF96], Laplacian pyramid encoding [GY95], wavelet encoding [Mur93, GS01, NS01, LHJ99, Wes95], and vector quantization [NH92] have also been proposed, with varying amounts of integration with rendering. Several 4D techniques have been specifically designed with time-varying data in mind, including tree-based [WvG94, SCM99] and subdivision [LPD*02] methods. Lum et al. [LMC01] use the Discrete Cosine Transform (DCT) of time series for compression with decompression on-the-fly in graphics hardware.

3. Deferred Filtering

In this section we describe deferred filtering, an efficient algorithm for rendering from packed or compressed volume data. The algorithm is a two-pass technique that decompresses into slices in the first pass and then renders filtered slices in the second pass. This approach was introduced by Lefohn et al. [LKHW03] in the context of rendering level-set surfaces from a sparse dynamic (i.e. GPGPU) texture format and later generalized [KLF05].

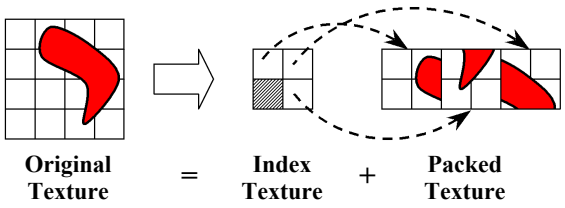


Figure 1: *Overview of block-based volumetric compression method for graphics hardware. Lossless compression (texture packing) stores only non-empty blocks and uses an index texture for mapping from the original volume domain to the packed volume. Lossy compression is similar to packing but uses a small representative set of blocks called the codebook instead of the packed texture.*

Here we explore the application of this idea in the context of rendering from compressed volume formats.

We first describe the conventional way to perform on-the-fly decompression using graphics hardware, followed by a discussion of the problems with this method. We then describe the basic deferred filtering technique and offer an analysis in terms of decompression costs. Our focus then shifts to implementation of deferred filtering in graphics hardware, including the calculation of gradients for lighting.

3.1. The Algorithm

In the context of decompression in graphics hardware there are essentially two options available. The first option is natively supported hardware decompression, which includes methods such as the S3 texture compression standard provided by S3 Inc. These types of methods are generally fixed-rate and lossy, designed primarily for color textures used in gaming. The second option is custom hardware decompression, which comes in two forms: lossless compression, also called texture packing, and lossy compression, which is based on vector quantization. Implementations of these decompression methods rely on the programmability of pixel shaders in modern graphics cards.

Texture packing partitions the volume into blocks and stores only the non-empty blocks. As shown in Figure 1, an additional texture often called the index texture is used to define a mapping from the original data domain to blocks in the packed texture. In order to support continuous reconstruction and shading, a space-filling arrangement as described by Ning and Hesselink [NH93] and Yeo and Liu [YL95], respectively, is applied to each block (Figure 2). As proposed by Kraus and Ertl [KE02], a remapping of the

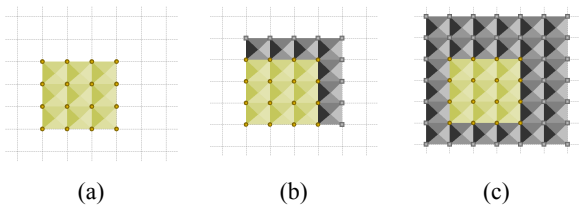


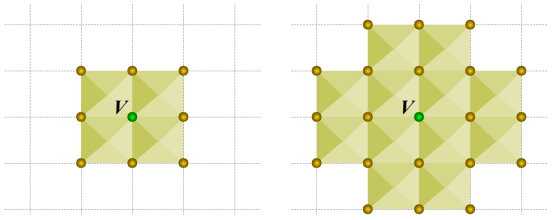
Figure 2: *Space-filling arrangements shown in 2D for a 4x4 block (a). Linear (in this case bilinear) interpolation requires 1-space-filling (b), whereas shading based on gradients requires 3-space-filling (c).*

Padding Overhead	Block Size			
Padding Type	4^3	8^3	16^3	32^3
Trilinear Interpolation	95%	42%	20%	10%
Gradient Calculation	436%	160%	67%	31%

Table 1: *Percent increase in block size when padding volume blocks. For most data sets the optimum block size is 8^3 or 16^3, in which case to support shading the packed blocks will need to be 160% or 67% larger, respectively.*

texture coordinate domain from cell-based texels (used by OpenGL) to vertex-based texels allows space-filling blocks in hardware, such that a block of size b^3 is increased in size by a factor of $(b+1)^3/b^3$ for trilinear interpolation and $(b+3)^3/b^3$ for gradient calculation. Although space-filling (or padding) allows the use of native hardware interpolation in the packed texture, the replication of data at block boundaries serves to undermine the compression rate by inflating the packed texture size. Furthermore, because each block is padded at the boundaries it is difficult to reuse blocks, since not only the block itself but also the boundary values would need to be identical [KE02]. Table 1 gives the overhead of texture packing for various volumetric block sizes. Note that in order to support gradient calculation a substantial increase in the block size is required, and in some cases packing may actually increase the size of the data.

Lossy methods rely on vector quantization, which is essentially a texture packing where blocks are reused. Again we have a mapping from the original data domain to the packed texture, which in this case is referred to as the codebook. Because a given block in the codebook texture may be used to represent many blocks in the original volume, it is usually not possible to pad the blocks. Although an "average" boundary could be constructed by looking at the blocks which reference the code block, this would result in increased blocking artifacts, which are already a problem for block-based compression methods. Consequently, each voxel needed in the interpolation filter must be individually decompressed in the fragment program prior to interpolation. Furthermore, the interpolation itself must then be carried out in the fragment program in order to obtain the scalar value needed for classification. A consequence of this is that since the interpolation is performed on a per-fragment basis, the complexity of the decompression is a direct function of the sampling rate. The problem with this is that although a single voxel may be needed for several fragments, rather



(a) (b)

Figure 3: *The filter support for linear interpolation (a) and gradient calculation (b) shown in 2D. Any sample within this support will require decompression of **V**. If **V** must be decompressed separately for each sample then we will perform 4 decompressions of **V** in (a) and 12 decompressions of **V** in (b). Analogously, in 3D we will perform 8 decompressions for linear interpolation and 32 for gradient calculation.*

than decompressing that voxel once and caching the result we must recalculate the decompression of that voxel for each use. For instance, consider the 2D cases depicted in Figure 3. Any sample point within the highlighted region will require voxel ***V*** to be decompressed. If we consider sampling once per voxel then for interpolation (depicted in Figure 3a) we will perform 4 decompressions (3 redundant decompressions) of each voxel; likewise, in 3D we will perform 8 decompressions (7 redundant decompressions) of each voxel. Now consider gradient calculations as well (Figure 3b). In order to compute the gradient for shading using central differences we need six additional interpolated values. This would mean 11 redundant decompressions in 2D and 31 in 3D. Thus we see that even when sampling just once per voxel the massive amount of work required to calculate an interpolated and shaded sample prohibits real-time calculation for reasonable sized data sets. This is why implementations of on-the-fly decompression typically forego shading and even continuous reconstruction, opting instead to use a nearest-neighbor reconstruction kernel requiring only one decompressed sample. However, even this approach will perform redundant decompressions when sampling more than once per voxel and/or performing shading.

We should say that texture packing can also use this approach of "hand-coded" interpolation, thereby alleviating the need for padding; however, texture packing would then suffer from the same problems associated with this method in terms of redundant decompressions.

It is clear that both packing and compression suffer from some form of inefficiency; in particular, packing is memory-inefficient, whereas compression is computation-inefficient. The problem is that with the inability to cache decompressed voxels among fragments the close coupling of decompression and interpolation results in redundant decompressions. The solution, therefore, is to separate these two steps in such a way that intermediate decompression results can be cached. We do this by using deferred filtering, a two-pass approach in which a small subset of the volume is decompressed in the first pass and in the second pass this subset is used for conventional rendering. The basic idea is to render slab-by-slab using axis-aligned slabs as shown in Figure 4. To render a single slab we use a first pass which decompresses two consecutive slices of the original volume. Then in the second pass sampling slices are rendered using native filtering to compute trilinearly interpolated samples. In short, deferred filtering proceeds as follows for a volume consisting of *k* slices:

1. Decompress volume slice 0.
2. For volume slice n=0 to k-1:
a. Decompress volume slice n+1
b. Render sampling slices between n and n+1

Whereas for lossy compression the first pass is used for performing the decompression computation, for lossless packing the first pass simply retrieves the voxel from the packed texture using the address from the index texture.

In this approach the slabs will be axis-aligned according to the current view, as in 2D texture-based volume rendering [REB*00]. However, unlike 2D texture-based volume rendering where three sets of slices are needed for the three axes, we dynamically reconstruct slices to be aligned with whichever axis is needed. It is important to realize that in the decompression pass slices are being

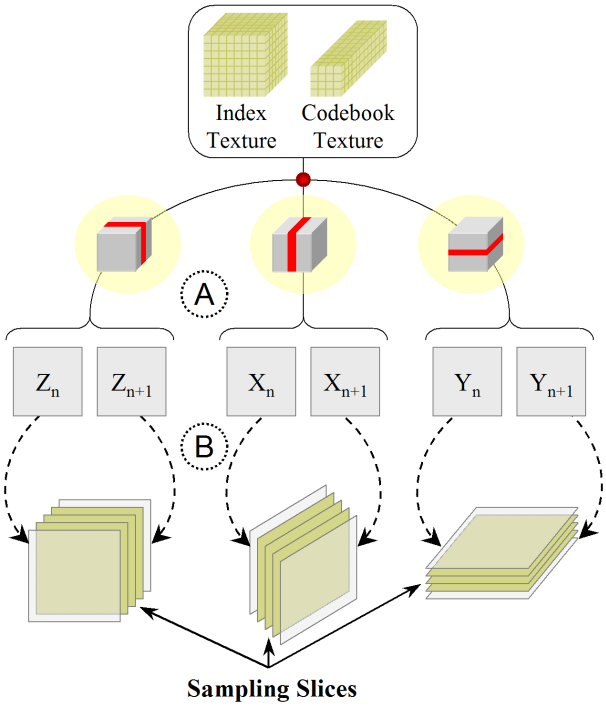


Figure 4: *The basic algorithm for deferred filtering. In the first pass (Step A) two consecutive slices are decompressed according to the axis with which the view is most closely aligned. In the second pass (Step B) the axis-aligned slab is rendered using sampling slices which lie in between the two decompressed volume slices.*

reconstructed voxel by voxel (i.e. nearest-neighbor interpolation) at exactly the resolution of the original volume. This guarantees that no matter how many times a voxel is needed it will be decompressed only once. Furthermore, in order to obtain an interpolated sample for each fragment of the sampling slices we read from the two decompressed volume slices defining the slab using native bilinear interpolation, and then finally weight each value by the slice's position within the slab to obtain the final value. On current hardware this is faster than performing the entire interpolation "by hand" in the fragment program, and this will probably be the case for future architectures as well.

The advantage of using deferred filtering for packing is that we no longer have to pad the blocks, thereby increasing the compression rate. Another advantage is that it is easier to reuse blocks as in vector quantization, as long as the

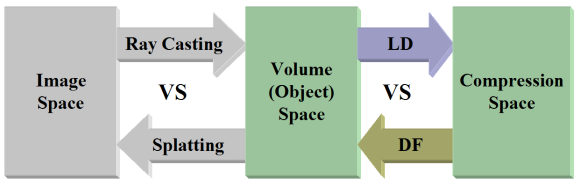


Figure 5: *Overview of integrated volume rendering and decompression. Whereas Lazy Decompression (LD) maps backward from volume space to compression space, Deferred Filtering (DF) maps forward from compression space to volume space. In mapping forward DF achieves greater efficiency by decompressing each voxel only once, regardless of how many times it is needed in filtering.*

values within the block are identical. This is important because many volumes contain significant redundancy, not just in the empty space but also in homogeneous regions and in natural patterns that repeat. These two factors should increase the utility of packing considerably; whereas previously a volume would need to contain significant amounts of empty space to justify packing, with deferred filtering we need only a small amount of empty space or some amount of redundancy in order to justify the added complexity of packing.

As we discuss at greater length in the next section, one nice feature of deferred filtering from an implementation perspective is the modular approach; all compression-specific information is used in the first pass only, and the second pass is canonical volume rendering: interpolation, classification, shading, and compositing (with a slightly modified interpolation for samples). Therefore the same implementation for the second pass of deferred filtering can be used regardless of whether we are rendering from packed or compressed volume data, or even custom GPGPU formats.

The disadvantages of deferred filtering are the restriction to axis-aligned slicing and the overhead of using two passes for each slab. However, adjusting the number and spacing of sampling slices per slab can almost entirely eliminate artifacts from renderings [REB*00]. The overhead of two-pass rendering comes both from additional rasterization and context/state changes. In practice we can minimize the costs associated with both of these items, as we describe in the next section.

One way to understand deferred filtering is to view decompression as a mapping, similar to the projection of volume data to form images in volume rendering (see Figure 5). In volume rendering there are two ways to map: forward and backward. In forward mapping (e.g. splatting), for each voxel an image contribution is computed, projected and finally accumulated. In contrast, backward mapping (e.g. ray casting) computes for each image element the accumulation of the contributions of voxels along the image ray.

If we cast decompression in terms of a mapping, then the conventional method (which henceforth will be called lazy decompression) would be backward mapping, whereas deferred filtering would be forward mapping, as shown in Figure 5. Difficulties sometimes associated with forward mapping, for instance computing the mapping function, unmapped fragments (holes), and interpolation, are not a problem because of the simple configuration of deferred filtering, in which the mapping is a simple one-to-one function which leaves no holes and requires no interpolation (interpolation is deferred to the second pass). In this work we show that in the context of decompression forward mapping is more appropriate for rendering from packed or compressed volumes.

To see why this is true let us consider the complexity of both lazy decompression (LD) and deferred filtering (DF). We can estimate the cost of decompression for lazy decompression, C_{LD}, to be:

$$C_{LD} = k \cdot d \cdot s \cdot n^3 \qquad (1)$$

Where k is the size of the filter kernel, d is the cost of decompressing a single voxel, s is the sample rate (samples per voxel), and n is the dimension of the data. Likewise we

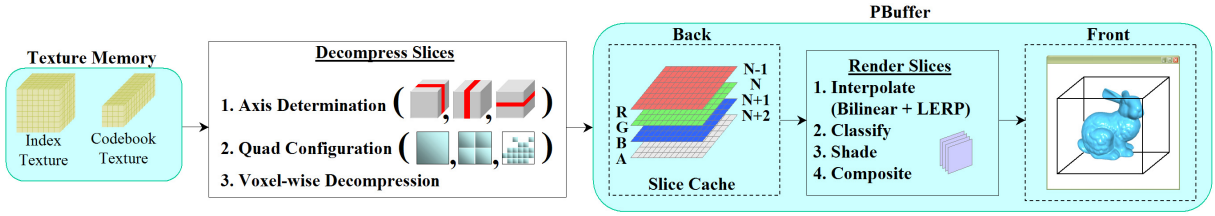


Figure 6: *Implementation of deferred filtering. In the first pass we use proxy geometry to generate a fragment per voxel; possible configurations are simple quad (left), multiple slice-filling quads for static resolution of volume partitions (center), and multiple quads for region-selective decompression (right). Four consecutive slices stored in the four back pbuffer channels allow efficient linear interpolation and gradient calculation for a single slab. Finally, in the second pass sampling slices within the slab are composited in the front pbuffer surface.*

can estimate the cost of decompression for deferred filtering, C_{DF}, as:

$$C_{DF} = d \cdot n^3 \qquad (2)$$

Keep in mind that these costs are for decompression only; filtering calculations are not included.

Let us first consider magnification, which we will define as a sample rate equal to or greater than one ($s \geq 1$). Since our aim is quality we desire at least first-order interpolation and shading based on gradients using central differences. In this configuration $k=32$, which means that decompression using deferred filtering will be $32s$ times more efficient than decompression using lazy decompression.

Now consider minification, in which the sample rate is less than one ($s<1$). At first glance it may seem that for some values of $s \ll 1$ lazy decompression will be more efficient than deferred filtering; however this is not in fact true. In order to prevent aliasing artifacts in minification some type of filtering is necessary; in fact, what is actually required is to integrate over the projected area of the sample in texture (or volume) space. As the sample rate decreases the projected area increases and thus the size of the minification filter kernel increases. The net result is that although we may sample less than once per voxel the product ks will remain constant; in fact, if we consider only the effects of minification on k then $ks \cong 1$ always.

If the compression method is inherently multi-resolution then deferred filtering can take advantage of this by decompressing the appropriate resolution directly. This offers pre-computed minification filtering similar to mip-mapping, and is potentially faster than decompressing the highest resolution.

3.2 Implementation in Graphics Hardware

While deferred filtering could also be used for software volume rendering of volumes too large to fit in core memory, we focus only on the implementation in graphics hardware. In software deferred filtering would not gain the efficiency afforded by hardware interpolation, but it would still provide efficient decompression. Deferred filtering would work well in conjunction with shear warp [LL94], for instance. It would be interesting to compare deferred filtering with an implementation of lazy decompression that cached decompressed voxels.

There are several ways to implement deferred filtering using current graphics hardware. We describe one way using OpenGL which is particularly efficient and requires only the ability to use fragment programs and to write to and read from off-screen buffers. Figure 6 shows the important components of this method. In order to minimize context changes we use a single pixel buffer (pbuffer) for both passes of the algorithm, although in other APIs and in future OpenGL extensions the ability to use a single context for all buffers will offer an alternative but equivalent implementation which uses two off-screen buffers. The back surface of the pbuffer is used to store the decompressed slices from the first pass, and the front surface is used for volume rendering in the second pass. As mentioned before, this method is modular in that the second rendering pass is independent of the first pass and therefore of the compression-specific code as well.

In the first pass voxels are individually decompressed into slices by rendering a quad the size of the slice into the back buffer. This back buffer is the slab or *slice cache* that will contain two (for shading we will need four) consecutive slices of the volume, with the slices aligned according to the axis closest to the view direction. Since our compressed format is inherently 3D (that is, our index texture is a volume since we compress volumetric blocks) we can reconstruct with quads for any of the three possible orientations. This is in contrast to the approach described by Lefohn et al. [LKHW03] for rendering from a single stack of 2D slices, where lines were used for the two orientations not matching the stored orientation. In order to efficiently use the back buffer we use two channels for the two slices instead of tiling them in one channel.

The purpose of the quad is to generate a fragment for each voxel, so in general we can use any configuration of geometry to achieve the same result. In fact, regions of empty space may not need to be decompressed at all. In this case we just render geometry into the non-empty regions of the slice, and in the empty regions we either initialize the buffer with a background color before each slab is rendered, or else render background-colored geometry. Although this requires calculation and storage of the additional geometry, in sparse volumes the acceleration can be significant.

In certain situations we may wish to partition the volume into regions to be processed separately. One example of this is rate inflation to increase the quality of compression; current methods use one fixed-size codebook for the entire volume, regardless of the volume size. For larger volumes or volumes with little correlation this may result in low quality encodings. Although we can increase the size of the codebook, an alternative solution is to partition the volume into macroblocks and use a separate codebook for each macroblock. The advantages of doing this over increasing codebook size are: a) better adaptation to locally varying

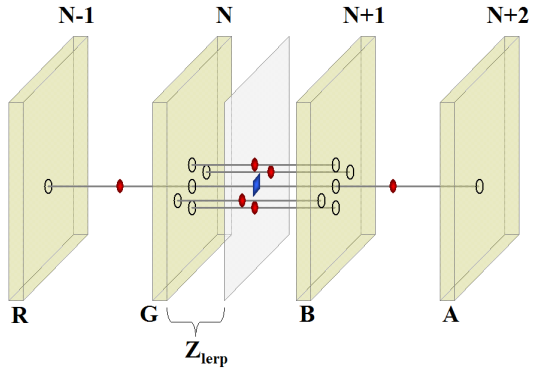


Figure 7: *Calculation of the gradient using central differences for a given sample (■) in a sampling slice requires six gradient samples (●). In order to obtain the gradient samples we need to access the decompressed slices at the corresponding locations (○). As the slices are stacked in the RGBA texture channels we need only five texture reads (horizontal lines). Calculation of the gradient samples then requires six lerps, using Z_{lerp} as the interpolation weight.*

statistics, b) better texture cache coherency, c) faster codebook searches, and d) scalability (increasing codebook size means having larger indices, which scales linearly in memory as opposed to the constant memory of the codebook). However, one negative aspect of using multiple codebooks for the same volume is that there will probably be redundancy among the codebooks.

In the first pass of deferred filtering we simply render quads that cover each partition in the slice, thereby achieving static resolution of the codebook. For instance, if we partition the volume equally into eight macroblocks then in the decompression pass we render four quarter-size quads for any given slice, as shown in Figure 6. In fact, we are free to use different parameters for each partition, including the compression method itself; we could compress one partition and pack another, for instance. Since volume rendering is generally fill-rate limited, the small amount of additional geometry has almost no effect on frame rates. Of course for many small partitions there will be a point where the geometry load might become a bottleneck; on current hardware when using uniform slice-covering partitions this begins to occur with quad sizes 16^2 and smaller.

In the second pass we render some number of sampling planes within the current slab. In order to obtain an interpolated sample we read from the corresponding location in both the front and back decompressed slices. We then perform the final interpolation in the fragment program using the z-value of the current slice. The z-value ranges from 0 to 1 and gives the offset of the slice within the slab; that is, for a given slice all samples will have the same z-value. Since the slices are stored in two channels, we need only one texture read from the back buffer to obtain both slice values. The sampling slices are composited into the front buffer in front-to-back or back-to-front order, depending on whether the slabs are advancing from front-to-back or back-to-front.

After an interpolated sample is obtained we perform classification and shading. Shading requires gradients, which can be pre-calculated or computed on-the-fly. The massive increase in data size required to store pre-computed gradients makes this method inappropriate for packing or compression applications, and so gradients must be computed on-the-fly. In order to do this we need the neighborhood of the sample position, which translates to a slightly larger slice cache; instead of keeping two consecutive slices we keep four, as seen in Figure 6. This works out well since we have exactly four texture channels to use. The slice cache then acts as a circular buffer, so that after decompressing three initial slices we then only need to decompress one slice per pass while using three slices from the previous slab. This is achieved by cycling the color channel, using buffer color masks to select the target channel. In Figure 7 we show the gradient computation using central differences. By stacking the slices in the texture channels we can obtain both the sample and gradient in only 5 bilinearly-filtered texture reads and 7 lerps.

An alternative implementation of deferred filtering is possible given the capability to render slices into a volume texture; instead of using the four channels of a pbuffer we use a four-slice volume texture. By using a volume texture to cache slices we are able to take advantage of native trilinear interpolation while providing a potentially more elegant implementation. In this case it would be necessary to use texture border mirroring for the *r* texture coordinate, along with a careful specification of the *r* texture coordinate domain.

Other options also exist for computing gradients, such as voxel-wise reconstruction of gradients during the decompression pass. We have experimented with two variants of this approach and found them to be less efficient than the approach described above. Note that in rendering adaptive level-set surfaces, Lefohn et al. [LKHW03] were able to reuse gradients computed in the level-set updates.

4. Results

In this section we show the results of applying deferred filtering to existing packing and compression methods. We chose one packing method and one compression method which we believe are representative and compared the original implementations using lazy decompression and their corresponding implementations using deferred filtering. All results were obtained using an NVIDIA GeForce 6800 Ultra and an image resolution of 512^2 pixels.

The texture packing method we use (shown in Figure 1) is a simplified version of previous methods. Specifically, we partition the volume into equal-sized blocks and store only non-empty blocks in a packed texture. We use an index texture to map from an original volume block to a volume block in the packed texture. The blocks themselves are padded using a space-filling arrangement in order to support linear interpolation and gradient calculation.

Figure 8 shows the results of packing applied to two data sets. The first is the $512^2 \times 360$ CT scan of the Stanford bunny volume. This data set can be compressed very effectively using only packing due to the sparseness of the volume. However, due to the overhead incurred by padding this sparseness is more effectively exploited by deferred filtering. The second data set is an atmospheric simulation generated by NCAR. The data set is time-varying (400 time steps), multivariate (5 variables), with a moderately high resolution ($256^2 \times 1024$ voxels). Texture packing without

padding reduces the total size of the data set from 125 GB to 3.2 GB, and with deferred filtering we can interactively render directly from the compressed data with high quality. Although the packed size is still too large to fit in graphics memory it does fit in core memory, and the compressed size of individual volumes (averaging 1.5 MB) allows more efficient storage and transfer to graphics memory. Note that the extra padding necessary with the previous approach results in a compressed size of 27.1 GB (averaging 13.8 MB per volume). Figure 8 shows rendering of one time step and one variable along with the packed texture.

Because of its performance and its more complex decompression scheme, we chose the method developed by Schneider and Westermann [SW03] as our compression subject. This method partitions the volume into blocks of size 4^3 which are then transformed using a Laplacian decomposition to a more energy compact representation consisting of the mean, a block of first-order differences (of size 2^3) and a block of second-order differences (of size 4^3). The first- and second-order difference blocks are separately quantized using vector quantization. In the index volume the mean and codebook indices for the difference blocks are stored. To decompress a voxel we take the mean and add the appropriate differences from the quantized difference blocks. While the original implementation used only nearest filtering, for purposes of comparison we also implemented a version which supports linear interpolation "by hand" in the fragment program. This involves separately decompressing 8 voxels and performing the interpolation itself (14 instructions) in order to obtain a single sample. We further implemented shading by decompressing 24 more voxels and performing six more interpolations (14*6=84 more instructions) for the gradient.

In Figure 9 we show the results of using this compression method on two different data sets. The first one is a 500×470×136 CT scan of a frog. We use a high-frequency transfer function (an implicit isosurface) and shading to test the compression. In order to get acceptable quality the volume was partitioned into 8 equal regions and a separate codebook was used for each partition. The timing results demonstrate the efficacy of deferred filtering. The second data set is a supernova simulation generated by TSI consisting of 200 time steps, 5 variables per time step, and volumes of 480^3 voxels. For compression we found that partitioning each volume into 8^3=512 macroblocks with associated codebooks was necessary for acceptable quality. The compression rate for this scheme is 87%, reducing the data set from 103 GB to 13 GB. Although the entire data set will not fit in memory, the reduced memory consumption and bandwidth greatly facilitate exploration. Rendering of a single time step and variable is shown in Figure 9.

5. Conclusions and Future Work

In this work we addressed the problem inherent in previous on-the-fly decompression methods, namely the inability to provide efficient decompression when coupled with volume rendering. The proposed solution, deferred filtering, enables interactive high quality volume rendering directly from compressed data. Deferred filtering is up to 20 times faster than the existing approach. Furthermore, this method is general enough to be used with all the existing texture packing and custom compression techniques, which we demonstrate with prototype implementations.

In the future we would like to further explore sparse volume optimizations for deferred filtering (both in the decompression pass and the rendering pass) with the goal of rendering large static volumes at higher frame rates.

Finally, in terms of available compression methods it is clear that when using transfer functions with high frequencies and/or shading even small errors in reconstruction result in objectionable artifacts. While using a larger codebook or multiple codebooks can help, a better alternative would be to come up with more performant techniques. The increased efficiency afforded by deferred filtering indirectly supports this by allowing more complex decompression.

6. Acknowledgements

We would like to thank Jens Schneider for his helpful comments and for contributing his compression code. The *nrrd* toolkit by Gordon Kindlmann was used for data set preparation (http://teem.sourceforge.net). The GLEW library by Milan Ikits was used for OpenGL extension management (http://glew.sourceforge.net). Thanks to TSI and NCAR for the simulation data sets. This work has been sponsored in part by the U.S. National Science Foundation under contracts ACI 9983641 (PECASE award), ACI 0325934 (ITR), ACI 0222991, and CMS-9980063; and Department of Energy under Memorandum Agreements No. DE-FC02-01ER41202 (SciDAC) and No. B523578 (ASCI VIEWS). Finally, we are indebted to the reviewers for their insightful comments and suggestions.

References

[BCF03] BINOTTO A., COMBA J., FREITAS C.: Real-time rendering of time-varying data using a fragment-shader compression approach. In *IEEE Parallel and Large Data Visualization and Graphics* (2003), pp. 69-75.

[CHF96] COCHRAN W. O., HART J. C., FLYNN P. J.: Fractal volume compression. In *IEEE Transactions on Visualization and Computer Graphics* (1996), 2(4), pp. 313-322.

[FY94] FOWLER J., YAGEL R.: Lossless compression of volume data. In *Proc. of the 1994 Symposium on Volume Visualization* (1994), pp. 43-50.

[GY95] GHAVAMNIA M., YANG X.: Direct rendering of laplacian pryamid compressed volume data. In *Proc. of IEEE Visualization Conference* (1995), pp. 192-199.

[GS01] GUTHE S., STRASSER W.: Real-time decompression and visualization of animated volume data. In *Proc. of IEEE Visualization Conference* (2001), pp. 349-356.

[GWGS02] GUTHE S., WAND M., GONSER J., STRASSER W.: Interactive rendering of large volume data sets. In *Proc. of IEEE Visualization Conference* (2002), pp. 53-60.

[KLF05] KNISS J., LEFOHN A., FOUT N.: Deferred filtering: rendering from difficult data formats. Pharr M. (editor), *GPU Gems II* (2005), ch. 41, pp. 669-677.

[KE02] KRAUS M., ERTL T.: Adaptive texture maps. In *Proc. SIGGRAPH/EG Graphics Hardware Workshop* (2002), pp. 7-15.

[LL94] LACROUTE P., LEVOY M.: Fast volume rendering using a shear-warp factorization of the viewing transform. In *Computer Graphics, Proc. of SIGGRAPH* (1994), pp. 451-458.

[LKHW03] LEFOHN A., KNISS J., HANSEN C., WHITAKER R.: Interactive deformation and visualization of level set surfaces using graphics hardware. In *Proc. of IEEE Visualization Conference* (2003), pp. 75-82.

[LKHW04] LEFOHN A., KNISS J., HANSEN C., WHITAKER R.: A streaming narrow-band algorithm: interactive computation and visualization of level sets. In *IEEE Transactions on Visualization and Computer Graphics*, 10(4), July-Aug (2004), pp. 422-433.

[LK02] LI W., KAUFMAN A.: Accelerating volume rendering with texture hulls. In *IEEE/SIGGRAPH Symposium on Volume Visualization and Graphics* (2002), pp. 115-122.

[LPD*02] LINSEN L., PASCUCCI V., DUCHAINEAU M., HAMANN B., JOY K.: Hierarchical representation of time-varying volume data with '4th-root-of-2' subdivision and quadrilinear B-spline wavelets. In *Proc. Pacific Graphics* (2002), pp. 346-355.

[LMC01] LUM E., MA K.-L., CLYNE J.: Texture hardware assisted rendering of time-varying volume data. In *Proc. of IEEE Visualization Conference* (2001), pp. 263-270.

[Mur93] MURAKI S.: Volume data and wavelet transforms. In *IEEE Computer Graphics and Applications* (1993), 13(4), pp. 50-56.

[NS01] NGUYEN K., SAUPE D.: Rapid high quality compression of volume data for visualization. In *Computer Graphics Forum* (2001), 20(3), pp. 49-57.

[NH92] NING P., HESSELINK L.: Vector quantization for volume rendering. In *Symposium on Volume Visualization* (1992), pp. 69-74.

[NH93] NING P., HESSELINK H.: Fast volume rendering of compressed data. In *Proc. of IEEE Visualization Conference* (1993), pp. 11-18.

[REB*00] REZK-SALAMA C., ENGEL K., BAUER M., GREINER G., ERTL T.: Interactive volume rendering on standard PC graphics hardware using multi-textures and multi-stage rasterization. In *Proc. SIGGRAPH/EG Graphics Hardware Workshop* (2000), pp. 109-118.

[SW03] SCHNEIDER J., WESTERMANN R.: Compression domain volume rendering. In *Proc. of IEEE Visualization Conference* (2003), pp. 293-300.

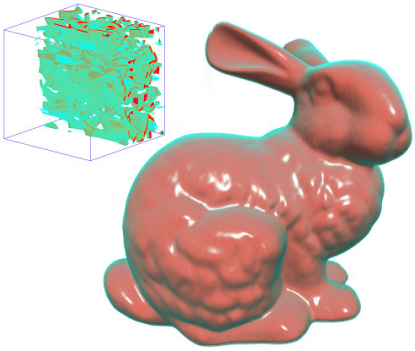
[SCM99] SHEN H.-W., CHIANG L., MA K.-L.: A fast volume rendering algorithm for time-varying fields using a time-space partitioning (TSP) tree. In *Proc. of IEEE Visualization Conference* (1999), pp. 371-377.

[Wes95] WESTERMANN R.: Compression domain rendering of time-resolved volume data. In *Proc. of IEEE Visualization Conference* (1995), pp. 168-175.

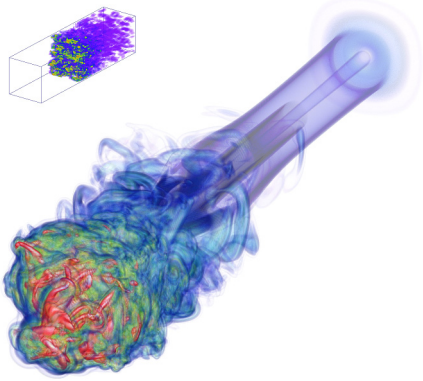
[WvG94] WILHELMS J., VAN GELDER A.: Multidimensional trees for controlled volume rendering and compression. In *Proc. of the 1994 Symposium on Volume Visualization* (1994), pp. 27-34.

[Yan00] YANG C.-K.: Integration of volume visualization and compression: a survey. Research Proficiency Exam Report, Sept. 2000.

[YL95] YEO B., LIU B.: Volume rendering of DCT-based compressed 3D scalar data. In *IEEE Transactions on Visualization and Computer Graphics* (1995), 1(1), pp. 29-43.



(a) Rendering of packed Stanford bunny data set:
LD: 11 fps, 76.1% compression
DF: 8 fps, 86.7% compression

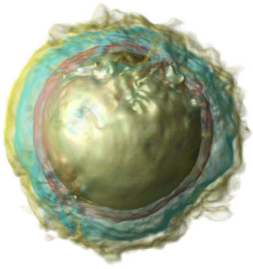


(b) Rendering of a packed NCAR simulation time step:
LD: 2.6 fps, 78.3% compression
DF: 2.2 fps, 97.4% compression

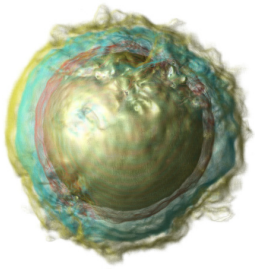
Figure 8: *Comparison of high quality volume rendering from packed textures using lazy decompression (LD) vs. deferred filtering (DF). Although DF is about 15% slower it allows significantly better compression.*

(a) Conventional volume rendering of the original frog data set (22 fps).

(b) Volume rendering of the compressed frog data set:
LD: 0.6 fps, DF: 14 fps



(c) Conventional volume rendering of an original supernova time step (1.8 fps).



(d) Volume rendering of the compressed supernova time step:
LD: 0.06 fps, DF: 1.1 fps

Figure 9: *Comparison of high quality volume rendering from compressed textures using lazy decompression (LD) vs. deferred filtering (DF). Images generated are almost identical, but DF is about 20 times faster than LD.*