Eurographics Conference on Visualization (EuroVis) 2012
S. Bruckner, S. Miksch, and H. Pfister
(Guest Editors)

*Volume 31 (2012), Number 3*

# COVRA
# A compression-domain output-sensitive volume rendering architecture based on a sparse representation of voxel blocks

Enrico Gobbetti, José Antonio Iglesias Guitián, and Fabio Marton

Visual Computing Group, CRS4, Pula, Italy – http://www.crs4.it/vic/

**Abstract**

*We present a novel multiresolution compression-domain GPU volume rendering architecture designed for interactive local and networked exploration of rectilinear scalar volumes on commodity platforms. In our approach, the volume is decomposed into a multiresolution hierarchy of bricks. Each brick is further subdivided into smaller blocks, which are compactly described by sparse linear combinations of prototype blocks stored in an overcomplete dictionary. The dictionary is learned, using limited computational and memory resources, by applying the K-SVD algorithm to a re-weighted non-uniformly sampled subset of the input volume, harnessing the recently introduced method of coresets. The result is a scalable high quality coding scheme, which allows very large volumes to be compressed off-line and then decompressed on-demand during real-time GPU-accelerated rendering. Volumetric information can be maintained in compressed format through all the rendering pipeline. In order to efficiently support high quality filtering and shading, a specialized real-time renderer closely coordinates decompression with rendering, combining at each frame images produced by raycasting selectively decompressed portions of the current view- and transfer-function-dependent working set. The quality and performance of our approach is demonstrated on massive static and time-varying datasets.*

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.3]: Picture/Image Generation—Computer Graphics [I.3.7]: Three-dimensional graphics and realism—Coding and Information Theory [E.4]: Data compaction and compression—Compression (Coding) [I.4.2]: Approximate methods—

## 1. Introduction

GPU accelerated direct volume rendering on consumer platforms is nowadays the standard approach for interactively exploring rectilinear scalar volumes. Even though the past several years witnessed great advancements in commodity graphics hardware, long data transfer times and GPU memory size limitations are often the main limiting factor, especially for massive, time-varying, or multi-volume visualization in both local and networked settings. To address this issue, a variety of level-of-detail data representations and compression techniques have been introduced. In order to improve capabilities and performance over the entire storage, distribution, and rendering pipeline, the encoding/decoding process must be highly asymmetric [FM07]. Compression and level-of-detail precomputation does not have real-time constraints and can be performed off-line for high quality results. In contrast, adaptive real-time rendering from compressed representations requires the incorporation of low-delay and spatially independent decompression within a multiresolution out-of-core renderer. Such a compression-domain adaptive rendering solution, however, imposes severe constraints on the compression method, as well as on the adaptive rendering architecture. Current solutions, often combining data transformations with fixed-rate block coding or vector quantization, exhibit a number of limitations in terms of achievable compression rate, quality, or capability to support interpolation and shading (see Sec. 2).

In this work, we present a novel multiresolution compression-domain GPU volume rendering architecture, which improves the state-of-the-art in terms of scalability and flexibility. In our approach – dubbed *Compression-domain Output-sensitive Volume Rendering Architecture (COVRA)* – the volume is decomposed into a multiresolution hierarchy of bricks. Each brick is further subdivided into

smaller blocks, which are compactly described by sparse linear combinations of prototype blocks stored in an overcomplete dictionary. The dictionary is learned, using limited computational and memory resources, by applying the K-SVD algorithm [AEB06] to a smartly non-uniformly sampled and re-weighted subset of the input volume, using the recently introduced method of *coresets* [AHPV05, CS07]. The result is a scalable high-quality coding scheme, which allows huge volumes to be compressed off-line and then adaptively streamed and decompressed on-demand by a real-time GPU-accelerated renderer. Our contributions are manifold:

- we introduce a flexible compression-domain rendering architecture supporting high-quality multi-sample rendering from general block-compressed data formats using a specialized decompress-and-render approach;
- we introduce sparse representations to the GPU volume rendering field as an effective asymmetric compression/decompression framework with fast GPU decoding, increasing quality and scalability of current vector quantization solutions;
- we introduce a coreset technique based on importance sampling for effectively learning a good quality sparse representation of a massive input volume using the K-SVD algorithm;
- we describe and evaluate an optimized CUDA implementation of our architecture, capable of preprocessing, streaming, and real-time rendering multi-gigabyte static and dynamic datasets using a limited memory footprint.

## 2. Related Work

Our system extends and combines state-of-the-art results in a number of technological areas. In the following, we only discuss the approaches most closely related to ours.

Filtering out as efficiently as possible the data that is not contributing to a particular image is paramount for massive volume rendering applications. This can be achieved by combining visibility and level-of-detail culling with out-of-core data management techniques. In this context, the out-of-core organization of massive volumetric data into a volume octree is a classic one. Early systems used the CPU for view- and transfer-function-based selection of blocks, and generated images by frame-buffer compositing of individual blocks rendered through slicing [LHJ99, BNS01, GS04] or raycasting [HQK05, KWAH06]. Small blocks, required for adaptivity, lead, however, to high communication overhead and pressure on compositing hardware. For this reason, researchers have recently introduced out-of-core GPU methods, which traverse adaptively maintained space-partitioning GPU structures covering the full working set [Lju06, GMI08, CNLE09]. In this paper, we introduce a hybrid approach, in which each frame is subdivided in a small number of octrees of compressed bricks. These octrees are decompressed and composited front-to-back. This allows us to co-ordinate the decompression and rendering processes, extending to adaptive multiresolution methods the capabilities of current deferred filtering solutions [FAM*05]. We thus fully harness the power of native texture filtering without requiring the storage of the full working set, nor forcing single resolution slice-based approaches.

In this context, data compression associated to GPU decompression is of great importance to save storage space and bandwidth at all stages of the processing and rendering pipelines. Few methods, however, support on-demand, fast and spatially independent decompression on the GPU, which is required for maximum benefits [FM07]. The simplest hardware-supported fixed-rate block-coding methods (e.g., OpenGL VTC [Cra, NIH08] or per-block scalar quantization [YNV08, IGM10]) have limited flexibility in terms of supported data formats and achievable compression. Vector quantization of volume blocks has often been used for realizing fast GPU decoders, in conjunction with adaptive texture maps [KE02] Laplacian pyramid compression schemes [SW03], or wavelet-based transform coders [FM07, PK09]. However, dictionary size imposes a hard limit on achievable quality and compression rate of vector quantization solutions [Ela08]. Recently, tensor approximation has been presented as an alternative in which compression is achieved through rank reduction of a preferential basis learned from data [SIM*11]. However, GPU tensor reconstruction costs are high, and real-time reconstruction is feasible only for small blocks, limiting achievable compression.

We improve on current methods by employing a representation in which each block is represented as a sparse linear combination of few dictionary elements. Recent years have witnessed a growing interest in such sparse representations (see the recent survey of Rubinstein et al. [RBE10] for an overview of the state-of-the-art). Data-specific dictionaries learned from each training set tend to perform better than dictionaries based on a mathematical model of the data (e.g., wavelets) [Ela08]. In this work, we employ the K-SVD algorithm [AEB06] for dictionary training, a state-of-the-art method in the domain [Ela08]. Performing K-SVD calculations directly on massive input volumes would, however, be prohibitively expensive. Even though memory problems could be circumvented with emerging online training techniques [MBPS10, SE10], massive datasets still lead to large computational time and possible numerical instabilities. For Bidirectional Texture Functions (BTF) compression, Ruiters and Klein [RK09] attacked this problem by reducing the dataset prior to K-SVD training through a truncated SVD. Instead, we perform data reduction by smartly subsampling and re-weighting the original training set, applying the concept of *coreset* [AHPV05, CS07]. A unified theoretical framework for constructing coresets for data approximation and clustering applications has been recently presented [FL11] and applied to image processing via K-SVD [FFS11]. We use here a simpler approach based on importance sampling.

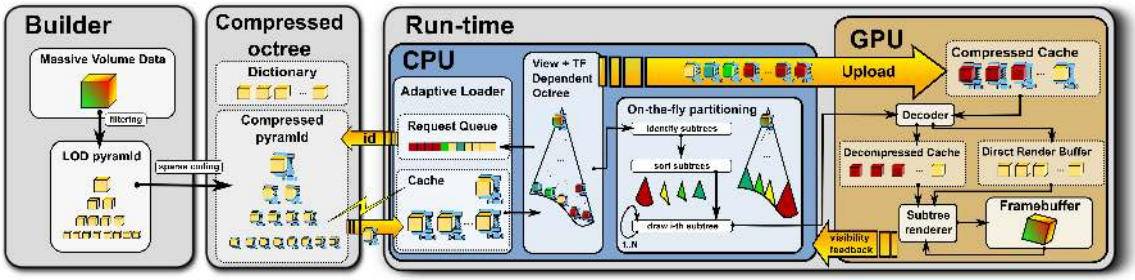


**Figure 1:** ***Architecture overview.*** *The volume is decomposed off-line into an octree of compressed bricks. At run-time, an adaptive loader moves data to GPU based on visibility feedback and transfer function criteria. A specialized renderer, then, generates and combines at each frame images produced by raycasting selectively decompressed portions of the current working set, exploiting a small decompressed brick cache.*

## 3. Architecture overview

Our method, see Fig. 1, is based on the offline decomposition of the original volumetric dataset into small cubical bricks, which are compressed and organized into an octree structure maintained out-of-core. The octree contains data bricks at different resolutions, where each resolution of the volume is represented as a collection of bricks in the subsequent octree hierarchy level. Each brick has a fixed width $B$ with an overlap of two voxels at each brick boundary for efficiently supporting runtime operations requiring access to neighboring voxels (trilinear interpolation and gradient computation). Each brick is in turn decomposed into smaller non-overlapping blocks of fixed width $M$, which are compressed with an efficient sparse coding technique (see Sec. 4). These blocks are compactly described by sparse linear combinations of prototype blocks stored in an overcomplete dictionary. In this work, time-varying datasets are handled by creating a separate octree per time-step, and temporal coherence is exploited by sharing the same dictionary among time-steps. In order to assist our run-time transfer-function aware brick-culling strategy (see Sec. 5), we also store a 64 bins binary histogram of the original volume brick together with the compressed brick representation.

At run-time, the dictionary is first uploaded to texture memory. Then, at each frame, an adaptive loader updates a view- and transfer function-dependent working set of bricks. Data is moved from the local or remote external database to GPU memory always in compressed format. Following earlier approaches [GMI08, CNLE09, IGM10], the working set is maintained by an adaptive refinement method guided by the visibility information fed back from the renderer. Since reconstruction from the dictionary is linear, direct rendering can be easily supported without block decompression by parallel random access to individual voxels. Most advanced volume visualization techniques require, however, a voxel's neighborhood for calculating its visual attributes, e.g., linear interpolation, gradient calculations, or ambient occlusion computation. In order to minimize reconstruction overhead and to fully harness texture filtering hardware, we perform rendering using a multi-pass approach which exploits a small texture cache of decompressed bricks (see Sec. 5). At each frame, the octree covering the current working set is partitioned into a number of subtrees small enough to be decompressed within prescribed cache limits. These subtrees are then rendered in front-to-back order and composited to produce the final frame buffer image. Each individual subtree is rendered using a raycasting approach [GMI08, CNLE09]. Since the bricks accessed by the raycaster are in a decompressed 3D texture cache, native trilinear filtering can be exploited, and shading methods requiring multiple samples per raycasting step can be implemented without additional compression overhead. This method, in contrast to deferred filtering [FAM*05], does not impose a slice-by-slice decompress-and-render approach, and therefore better supports perspective rendering and adaptive multi-resolution volume rendering with empty-space leaping and early-ray termination.

## 4. Building the compressed octree

The volume encoding process transforms a rectilinear volume into a compact multiresolution representation consisting of an octree of compressed bricks of size $B$ and a dictionary $\mathbf{D}$ of prototype blocks. The parameters guiding the process are the brick size $B$, which determines the octree granularity, the compressed block size $M \leq B$, the dictionary size $K \geq M^3$, the sparsity level $S \leq K$, and a threshold $\varepsilon \geq 0$ used for identification of constant and empty bricks.

Processing begins by computing the number of levels $L$ required to cover the entire input volume starting from a single root brick of size $B^3$. The $L$ levels of the LOD pyramid are then computed bottom-up and stored on disk. The compression process, then, first learns a good sparsifying dictionary $\mathbf{D}$ from the data contained in the pyramid (see Sec.4.2), and finally iterates over all octree bricks for final encoding.

Brick encoding starts by computing a 64 bins binary histogram and the range of values $v_{min}..v_{max}$ contained in the brick. Empty bricks, i.e., those for which $v_{max} \leq \varepsilon$, are skipped and will be considered full of zeros at rendering time. Constant bricks, i.e., those for which $v_{max} - v_{min} \leq \varepsilon$, simply store the average brick value. All others are, instead, approximated by the $S$-sparse representation of their blocks using dictionary $\mathbf{D}$.

### 4.1. Sparse-coding of blocks

For sparse coding, we map each block of size $m = M^3$ to a column vector $\mathbf{y} \in \mathbb{R}^m$. Given the dictionary $\mathbf{D} \in \mathbb{R}^{m \times K}$ computed in the learning phase, a sparse representation of $\mathbf{y}$ of at most $S$ entries in each column can be found by solving the following problem:

$$\min_{\lambda_i} \left\{ \|\mathbf{y}_i - \mathbf{D}\lambda_i\|^2 \right\} \text{ subject to } \|\lambda_i\|_0 \leq S \qquad (1)$$

where $\|\lambda_i\|_0$ is the number of non-zero entries in $\lambda_i$. The exact solution to this problem is NP-hard, but several efficient greedy pursuit approximation algorithms exist [Ela08]. In this work, we employ ORMP via Choleski Decomposition [CARKD99] because of its simplicity and efficiency.

Compression of $\mathbf{y}_i$ is thus achieved by storing the sparse representation of the vector $\lambda_i$, specifying the indices of its nonzero elements and their magnitudes. Furthermore, we assume that the learning phase produces a dictionary with normalized columns. Further compression with small increase in error is thus easily achieved by quantization of the nonzero entries in $\lambda_i$. As we will show in Sec. 5, the resulting packed representation leads to an efficient GPU implementation of the decoding process.

### 4.2. Dictionary learning

Searching for the best dictionary can be viewed as a generalization of vector quantization, in which we allow each block $y_i$ to be represented by a linear combination of dictionary entries rather than by a single representative. This corresponds to solving the following optimization problem:

$$\min_{\mathbf{D},\lambda_i} \left\{ \sum_i \|\mathbf{y}_i - \mathbf{D}\lambda_i\|^2 \right\} \text{ subject to} \forall i, \|\lambda_i\|_0 \leq S \qquad (2)$$

The K-SVD algorithm [AEB06] employed in this paper has emerged as a highly effective method for finding approximate solutions to this problem [Ela08]. It is a generalization of K-Means clustering which iterates between a sparse coding step, in which all $\lambda_i$ are optimized using the pursuit algorithms of Sec. 4.1, and a dictionary update step, in which $\mathbf{D}$ is optimized. Update is done independently for each dictionary entry $\mathbf{d}_k$ by performing a SVD on the residual matrix computed without $\mathbf{d}_k$ itself and for only those samples which are represented by $\mathbf{d}_k$. The first dictionary element, denoted as DC, is kept fixed to $\mathbf{d}_1 = \frac{\mathbf{1}}{\sqrt{m}}$. The DC takes part in all representations, and as a result, all other dictionary elements remain with zero mean during all iterations. As in Rubinstein et al. [RZE08], we replace the explicit SVD computation with a numeric power approximation, which eliminates the need to fully compute and store the residual matrix.

Even with this modification, performing K-SVD calculations directly on massive input volumes would, however, be prohibitively expensive, as at the minimum, full sweeping of all input samples is required at each coding and update steps.

In order to tackle the problem, we reduce the amount of data used for training. Just uniformly sub-sampling the original volume is not an option, since, in typical input volumes, a small subset of the blocks, e.g., on material boundaries, carries more information than much of the rest, e.g., empty space or other uniform/slowly varying areas. Therefore, a dictionary for the sub-sampled set would not be a good solution for the entire data set. Instead, we use an importance sampling approach, motivated by the notion of *coreset*, which, informally, is a small weighted subset that approximates the original data in a problem-dependent sense.

First, we associate an importance $\iota_i$ to each of the original volume blocks. Since the blocks represent small volume patches, we choose to set $\iota_i$ to the standard deviation of the entries in $\mathbf{y}_i$. Then, we take a non-uniform random sample of the set of blocks, picking $C$ elements with probability proportional to $\iota_i$. In this way, there will be with a reasonable probability enough samples from the more important blocks inside the input volume. Applying the K-SVD algorithm to the selected subset $\tilde{\mathbf{Y}}$ would, however, not solve the original problem in Eq. 2, since non-uniform sampling introduces a severe bias. We thus scale each selected block $\mathbf{y}_j$ by a weight $w_j = \frac{1}{\sqrt{p_j}}$, where $p_j$ is the associated picking probability. Sparse coding a scaled vector $\tilde{\mathbf{y}}_j = w_j \mathbf{y}_j$ leads to scaled coefficients $\tilde{\lambda}_j = w_j \lambda_j$. It is easy to prove using Eq. 2, that, since

$$\sum_j \left\| \tilde{\mathbf{y}}_j - \mathbf{D}\tilde{\lambda}_j \right\|^2 = \sum_j \frac{1}{p_j} \left\| \mathbf{y}_j - \mathbf{D}\lambda_j \right\|^2 \approx \sum_i \|\mathbf{y}_i - \mathbf{D}\lambda_i\|^2$$

applying K-SVD to the resampled and re-weighted set of blocks $\tilde{\mathbf{y}}_j$ will converge to a dictionary approximating the one associated to the original problem. As illustrated in Sec. 6, using this approach, extremely good approximations can be obtained by using a tiny fraction of original data.

### 4.3. Coreset construction

We compute the coreset $\tilde{\mathbf{Y}}$ using a multipass streaming algorithm over all blocks of the multiresolution hierarchy. In a first streaming pass over the input blocks, we compute lower and upper bounds of the importance function. We then perform a second streaming pass over the blocks to compute an histogram of size $H$ (512 in our tests) of the importance function, counting the blocks that fall into each importance bin $Q_q$. The histogram is used to determine the number of blocks $B_q$ that will be sampled in each importance interval, and the associated weight $w_q = \sqrt{Q_q / B_q}$.

In order to compute $B_q$, we start by assigning an initial budget $B_q^{(0)} = Q_q \cdot \frac{q}{H}$ compatible with the desired picking probability, which should be proportional to the importance. We then refine this initial estimate to achieve the desired coreset size $C$, i.e., to obtain $\sum B_q = C$. To achieve this goal, we first handle undersampling (i.e., $\sum B_q < C$) by setting the minimum number $U$ of high importance budget to full

sampling ($B_h = Q_h \forall h \in H, H-1, ..., H-U$). We then manage undersampling by linearly adjusting the sampling rate of the remaining slots to achieve the desired total number of samples $C$. Given the budget $B_q$ and the input block count $Q_q$, coreset construction is then performed in a final streaming pass, using Vitter's reservoir sampling approach [Vit85], with a separate reservoir for each of the $Q$ importance bins and a weighting by $w_q$ of each of the picked elements associated to reservoir $q$.

## 5. Compression-domain adaptive rendering

The run-time stage exploits an adaptive loader to access local and remote compressed volume databases. Data is moved from the external database to GPU memory always in compressed format, and rendering is performed using a multi-pass approach, implemented in CUDA, using three levels of cache. A large RAM cache is used to reduce pressure to the network/disk, while the two GPU caches store, respectively, recently visited compressed bricks and decompressed bricks required for rendering portions of the current frame. The decompressed brick cache can be maintained at a small size. While direct rendering from compressed bricks is definitely possible, this approach is more efficient for non-trivial rendering methods, which require repeated access to multiple samples at each ray-tracing step.

### 5.1. Multi-pass adaptive rendering strategy

The rendering process is subdivided in three main phases: (1) adaptive refinement of the multiresolution octree, (2) partitioning of the octree into a set of subtrees, and (3) subtree decompression, raycasting and compositing.

In the first phase, an adaptive loader updates a view- and transfer function-dependent working set of bricks, maintaining on the GPU a cache of recently used compressed volume bricks stored in a 3D texture by asynchronously loading it from a local or remote external database. Following earlier approaches [GMI08, CNLE09], the working set is maintained by an adaptive refinement method guided by the visibility information fed back from the renderer.

We then split the working set defined in the first phase into separate subtrees, using the size of the decompressed brick cache as a constraint. In a post-order visit of the tree, we construct the subtrees bottom-up, recursively merging subtrees if their union fits into the cache. At the end of this second phase, each subtree thus contains only up to the number of leafs that can fit in decompressed form into the decompressed brick cache.

In the last rendering phase, we produce the final image, decompressing, raycasting, and compositing subtrees in front-to-back order. Rendering starts by clearing the frame-buffer to transparent black. Subtrees are then rendered and composited during a front-to-back hierarchical traversal of the octree. At each visited octree brick, we test whether it was identified as an octree root in the previous phase. If not, we recursively visit the brick's children in front-to-back order, using the octree split planes for fast visibility ordering.

Subtree rendering starts by decompressing all volume bricks covered by the subtree into the decompressed brick cache, using an efficient CUDA decompressor (see Sec. 5.2), before using a raycaster to produce the image. Volume bricks are decompressed in GPU and directly written to GPU memory. To exploit temporal coherence we also implemented in GPU a LRU cache of decompressed brick data, which records recently used decompressed bricks. If the amount bricks required for a single frame exceeds the cache size, only the first accessed bricks are kept inside the cache, while the remaining bricks are decompressed to a temporary area of the cache which is rewritten for successive subtrees.

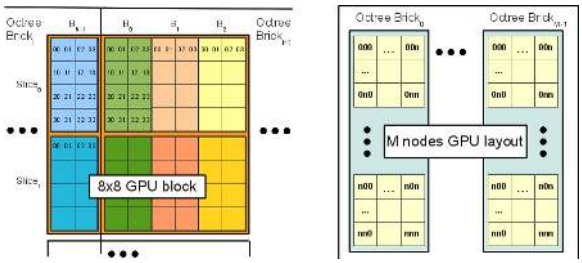


**Figure 2:** ***Decompression layout.*** *Left: voxel distribution inside an* $8 \times 8$ *CUDA thread block for decompression of* $4^3$ *K-SVD blocks; the CUDA block decompresses two slices of four adjacent K-SVD blocks, and each thread computes two output voxels; a single CUDA block can decompress blocks belonging to two bricks. Right: GPU grid layout for M octree bricks; each brick is made by* $N^3$ *compressed blocks which are mapped to a 2D grid of* $N \times N^2$ *blocks; small quads represent single blocks, which in turn are unrolled as shown on the left.*

### 5.2. CUDA decompression

In our approach, the GPU decompresses in parallel all the bricks referenced by a given subtree to a 16bit texture before rendering them.

Each brick is made of $N^3$ compressed blocks, each of them of linear block size $M$. Thus a decompressed brick is made of $(N \times M)^3$ voxels. All the elements of a single K-SVD block with sparsity $S$ share the same representation, encoded as $S$ pairs of coefficients $c_i$ and indices $a_i$. The decoded block is given by $\sum_{i=1}^{S} c_i * \mathbf{d}[a_i]$, where $\mathbf{d}[i]$ is the i-th dictionary element, made of $M^3$ values.

The dictionary $\mathbf{D}$ is stored in a 2D texture of unsigned shorts, where each row of the texture represents a dictionary element $\mathbf{d}$. Each $(index, coefficient)$ pair is encoded in 24 bits, using from 8 to 16 bits for the index (depending on the dictionary size), and the remaining bits for the coefficient.

For example a dictionary of 1024 elements uses 10 bits for the index and 14 for the coefficient. For each brick, we also store the coefficient range of all the compressed blocks of the brick, which permits to dequantize the coefficient values.

Decompression is implemented by a two-phase CUDA kernel: in the first phase the first $T$ threads of the CUDA block cooperatively move to shared memory the $T$ $(c_t, a_t)$ pairs needed to decode all the voxels belonging to the K-SVD blocks associated to the current CUDA block. Threads are then synchronized, and in the second phase, voxels are reconstructed by linear combination of dictionary elements. In order to optimize memory access, each CUDA thread reconstructs two voxels. The idea behind this coupled voxel decompression is that we can fetch two adjacent unsigned short dictionary values with a single 32 bit fetch operation, and write the decompressed result as a single 32 bit write operation, significantly reducing memory access overheads.

Fig. 2 illustrates the memory layout of the data referenced by the CUDA kernel, which decompresses multiple bricks. The layout is exploited in the first phase of the decoding method, where we need to identify which are the (maximum) two bricks partially covered by the current GPU block. In this layout, 3D bricks are mapped to a 2D grid, trying to minimize the number of K-SVD blocks which are decoded by a single GPU block, and thus the number of fetches performed in the first part of the algorithm. In this layout, 2D brick slices are appended one after the other in the Y direction of the CUDA grid, and bricks are appended one after the other in the X direction. In the first phase, we can thus fetch $3 \times K$ bytes for each decoded K-SVD block. In order to optimize memory transfer, the first $\left\lceil \frac{3\times K}{4} \right\rceil \times$ blockcount threads of the CUDA block moves 4 bytes to shared memory. In order to support this approach, brick rows in the database are aligned to 4-byte boundaries, while bricks on shared memory are aligned to 12-byte boundaries, ensuring proper alignment for write (4 bytes) and read (3 bytes) operations.

### 5.3. Subtree rendering and compositing

After all bricks are decompressed, the renderer builds an octree spatial index, which is uploaded in GPU and is used to traverse the subtree using a stackless raycaster, similarly to Crassin et al. [CNLE09]. The spatial index is linearized in an array of 32-bit index nodes. The first byte encodes the node type (split, data, or constant). The next bytes encode the index of the first child in the spatial index for split nodes, the average of brick values for constant nodes, or the brick location inside the texture storing decompressed bricks for data nodes. Similarly to [IGM10] a parallel write-only Boolean array, initialized at false, is used for visibility feedback.

The raycaster is implemented in a single CUDA kernel, which renders the subtree to a viewport of the frame buffer that strictly encloses the projection of the subtree's bounding box. Each thread produces the color and opacity of a single pixel, and updates the visibility status of traversed nodes. The subtree raycasting procedure starts from the color and opacity fetched from the frame buffer, and follows the ray accumulating colors and opacity until maximum opacity is achieved or the ray leaves the subtree. During traversal, the visibility feedback array of each traversed leaf is set to true. At the end, the accumulated color is written to the frame buffer. The CPU then reads back the visibility array and updates the visibility status of rendered octree bricks. After rendering all subtrees, the frame-buffer contains the final composited image for the volume, and the visibility status of all octree bricks is up-to-date and can be used to guide the next frame's refinement step.

| | Chameleon | | | Visible Human | | | Supernova | | |
|---|---|---|---|---|---|---|---|---|---|
| Coreset | Size | Time | PSNR | Size | Time | PSNR | Size | Time | PSNR |
| % | Mvox | h | dB | Mvox | h | dB | Mvox | h | dB |
| 1.40% | 18 | 0.16 | 52.18 | 8 | 0.06 | 38.19 | 82 | 0.76 | 49.23 |
| 3.13% | 38 | 0.29 | 52.39 | 16 | 0.15 | 38.32 | 164 | 1.49 | 49.30 |
| 6.25% | 77 | 0.56 | 52.57 | 33 | 0.28 | 38.43 | 329 | 2.94 | 49.32 |
| 12.50% | 154 | 1.12 | 52.69 | 67 | 0.56 | 38.54 | 659 | 5.79 | 49.35 |
| 25.00% | 308 | 2.21 | 52.76 | 134 | 0.97 | 38.59 | 1318 | 11.73 | 49.46 |
| 50.00% | 617 | 4.37 | 52.81 | 268 | 2.09 | 38.59 | 2636 | N/A | N/A |
| 100.00% | 1234 | 8.66 | 52.85 | 536 | 4.16 | 38.60 | 5273 | N/A | N/A |

**Table 1:** ***Coreset tests.*** *Training time and reconstruction quality as a function of coreset size. Tests marked N/A were not completed due to memory overflow on a 8GB PC. All tests were performed with* 25 *iterations, fixed dictionary size* $K = 1024$*, block size* $M = 6$*, sparsity* $S = 8$ *and the tolerance* $\varepsilon = 0.015$.

## 6. Implementation and results

An experimental software library has been implemented on Linux using C++, OpenGL and NVIDIA CUDA 4.0. The out-of-core octree structure is implemented on top of Berkeley DB. Network access to compressed datasets is implemented through a HTTP protocol, using Apache 2.2 on the server side. The preprocessor is structured to exploit OpenMP for parallel encoding.

We have tested our system with a variety of high resolution models and settings. In this paper, we discuss the results obtained with the inspection of three datasets: a micro-CT scan of a Veiled Chameleon specimen ($1024 \times 1024 \times 1080$, 16bit/sample: 2.1GB), the Visible Human Male Frozen CT ($512 \times 512 \times 1877$, 16bit/sample: 0.91GB), and a 60 time steps time-varying Supernova simulation ($432^3 \times 60$, float – 18GB). All the tests have been performed on a Linux Intel 3.2 GHz Core I7 PC with a NVIDIA GTX 560 with 1GB of video memory.

### 6.1. Encoding performance

In order to evaluate our coreset-based training strategy, we ran a battery of tests, changing the fraction of the input dataset retained in the coreset for dictionary learning. In all these tests, we used a K-SVD block size $M = 6$, a target

| | **Chameleon** (2.1 GB) | | | | **Visible Human** (0.91 GB) | | | | **Supernova** (18 GB) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **K-SVD,** $C = 64$**Mvox,** $K = 1024$ | **M4 S16** | **M6 S8** | **M6 S4** | **M8 S4** | **M4 S16** | **M6 S8** | **M6 S4** | **M8 S4** | **M4 S16** | **M6 S8** | **M6 S4** | **M8 S4** |
| Size(MB) | 1112.1 | 170.3 | 85.4 | 36.4 | 487.4 | 72.6 | 36.5 | 16.7 | 1741.3 | 258.4 | 129.4 | 56.6 |
| PSNR(dB) | 65.98 | 52.44 | 49.25 | 46.77 | 49.74 | 38.55 | 36.34 | 35.02 | 59.49 | 49.11 | 46.57 | 43.88 |
| Bps | 4.9 | 0.75 | 0.38 | 0.16 | 4.95 | 0.74 | 0.37 | 0.15 | 1.39 | 0.21 | 0.10 | 0.05 |
| Training time | 2h39m | 33m | 15m | 14m | 2h40m | 32m | 14m | 12m | 2h9m | 41m | 26m | 24m |
| Encoding time | 2h21m | 28m | 12m | 12m | 58m | 12m | 5m | 4m | 3h13m | 45m | 23m | 21m |
| Decoding (MVox/sec) | 1283 | 1701 | 1942 | 1838 | 1283 | 1701 | 1942 | 1838 | 1283 | 1701 | 1942 | 1838 |
| **HVQ,** $C = 64$**Mvox** | **D8192** | **D4096** | **D1024** | **D256** | **D8192** | **D4096** | **D1024** | **D256** | **D8192** | **D4096** | **D1024** | **D256** |
| Size(MB) | 144.5 | 143.9 | 143.4 | 143.3 | 62.1 | 61.5 | 61.1 | 60.9 | 218.9 | 218.3 | 217.8 | 217.7 |
| PSNR(dB) | 48.98 | 48.51 | 47.53 | 46.46 | 37.29 | 36.84 | 35.85 | 34.66 | 46.93 | 46.43 | 45.29 | 43.75 |
| Bps | 0.64 | 0.63 | 0.63 | 0.63 | 0.63 | 0.62 | 0.62 | 0.62 | 0.18 | 0.17 | 0.17 | 0.17 |
| Training time | 1h8m | 35m | 10m | 4m | 1h5m | 33m | 9m | 3m | 1h19m | 46m | 23m | 17m |
| Encoding time | 32m | 18m | 6m | 3m | 13m | 7m | 2m | 1m | 54m | 32m | 15m | 11m |
| Decoding (MVox/sec) | 2012 | 2011 | 2010 | 2011 | 2012 | 2011 | 2010 | 2011 | 2012 | 2011 | 2010 | 2011 |
| **TD** | **R16** | **R12** | **R8** | **R4** | **R16** | **R12** | **R8** | **R4** | **R16** | **R12** | **R8** | **R4** |
| Size(MB) | 357.2 | 201.2 | 102.6 | 42.2 | 152.8 | 86.1 | 43.9 | 18.0 | 502.2 | 283.0 | 144.3 | 59.3 |
| PSNR(dB) | 55.72 | 46.63 | 44.56 | 42.70 | 39.63 | 37.68 | 35.30 | 31.74 | 50.60 | 47.61 | 43.52 | 39.42 |
| Bps | 1.48 | 0.83 | 0.43 | 0.17 | 1.38 | 0.78 | 0.40 | 0.16 | 0.42 | 0.24 | 0.12 | 0.05 |
| Encoding time | 1h45m | 1h22m | 1h2m | 31m41s | 51m | 30m | 29m | 21m | 2h22m | 1h54m | 1h27m | 59m |
| Decoding (MVox/sec) | 415 | 498 | 625 | 781 | 415 | 498 | 625 | 781 | 415 | 498 | 625 | 781 |

**Table 2:** ***Compression/decompression performance tests.*** *We fixed the tolerance* $\varepsilon = 0.015$ *for all algorithms. K-SVD and HVQ were run for* 25 *training iterations.*

sparsity $S = 8$ and a dictionary size $K = 1024$. The K-SVD algorithm was run for 25 iterations. Consistent behaviors are obtained with other parameter settings.

As highlighted in Table 1, dictionary learning time is linear in coreset size, and very good dictionaries are obtained with extremely small coresets, since PSNR (Peak Signal-to-Noise-Ratio) reduction saturates very quickly. Using only about 3% of the input data leads to a decrease in PSNR of approx $0.2 - 0.6$ dB with respect to using the full dataset, while decreasing by over $30\times$ the required time and memory resources. This fact allows us to effectively process datasets which would be otherwise infeasible due to memory (or time) constraints. For instance, we were unable, due to memory overflow on our 8GB PC, to complete training for the time-varying Supernova datasets with coresets exceeding 25% of the dataset size. It is also interesting to note that, given the quick convergence of the method, we can expect to obtain good quality compression with bounded coreset sizes, leading to a practically constant-time/memory learning method.

### 6.2. Compression rate and distortion

We evaluated compression performance on both static and dynamic datasets by analyzing the results of our method with different parameter settings. We maintained fixed the dictionary size $K = 1024$, the coreset size $C = 64$Mvoxels and the tolerance $\varepsilon = 0.015$, while varying block size $M = 4, 6, 8$ and target sparsity $S = 4, 8, 12, 16$. The small size of the dictionary improves cache coherence and allows us to allocate 10 bits for the index and 14 bits for the coefficient in our block encoding scheme. Fig. 3 illustrates the compression performance of our method in term of rate-distortion curves. As is common in image compression we use bits per sample (i.e., bits/output voxel) to measure compression rate and PSNR to measure error. The scalability of our method is demonstrated

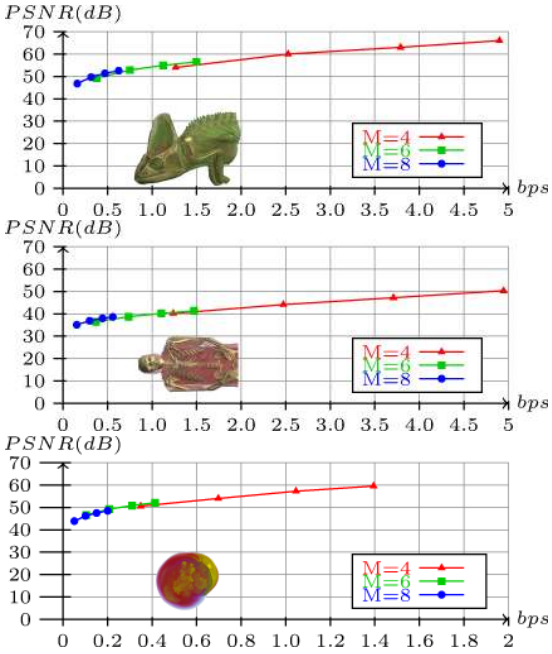


**Figure 3:** ***Compression performance.*** *Rate-distortion curves for large static and dynamic datasets.*

by the fact that, by suitably tuning block sizes and sparsity, our method can span wide ranges of both compression rates and quality. The much higher compression rate of the Supernova dataset is due to the higher input bit count and the larger fraction of empty/constant voxels for this dataset.

Fig. 4 illustrates the quality obtained at various bitrates. It can be seen that despite the high compression rates, the essential parts as well as details of a certain feature size can be visualized in all datasets. As in all current block-based lossy compression methods, our method may, however, introduce

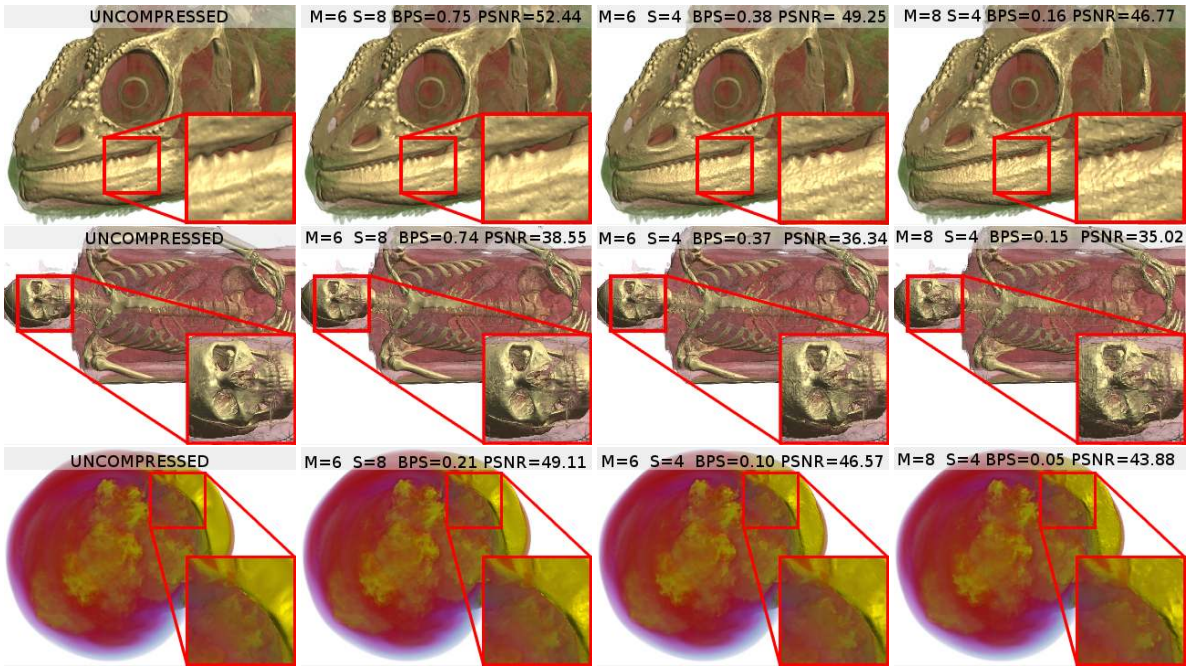


**Figure 4:** ***Compression quality.*** *Quality and bitrate of the three compressed datasets with different compression parameters.*

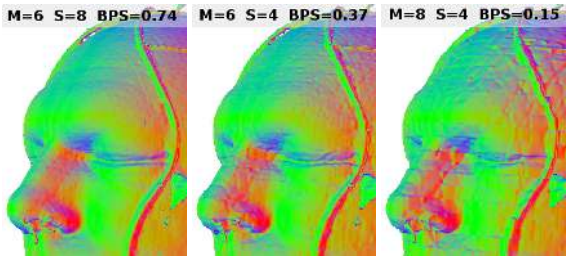


**Figure 5:** ***Visualization of gradients.*** *Compression artifacts are emphasized by using the gradient vector of the skin isosurface of the Visible Human mapped to RGB color. Block boundaries become apparent at low bit rates.*

block artifacts at low bit rates that manifest themselves as an annoying discontinuity between adjacent blocks. Fig. 5 shows the effects of compression on gradient quality. We plan to investigate how to visually improve the results, incorporating in our framework solutions inspired by the post-processing block-artifact removal approach of current image compression methods.

In order to provide a context for the evaluation of our work, we have implemented Hierarchical Vector Quantization (HVQ) [SW03] and Tensor Decomposition (TD) [SIM*11] within our framework for the encoding of octree bricks. The TD codec is unmodified with respect to the original implementation, while our HVQ implementation is optimized using a coreset technique similar to the one presented in this paper. We verified that our implementations were able to replicate (or improve) the results presented in the original papers for all the available datasets, before running compression/decompression tests with a variety of settings. Numerical results for both compression and decompression are presented in Table 2.

In terms of encoding time, while HVQ and TD appear to be generally faster, K-SVD performance remains tractable, thanks to the coreset approach, especially at low bitrates, when the sparsity level is reasonably low. On the other hand, our method sensibly achieves, on average, better rate-distortion performance with respect to HVQ and TD. The PSNR for HVQ on the Visible Human is very similar to the one reported by Fout et al. [FM07] for HVQ at the same bitrates. This allows us to also compare our results with their Transform Coding Vector Quantization approach (TCVQ) [FM07], which reports a PSNR of about 33.8dB at 0.37bps and 37.8dB at 0.74bps. At the same bitrates, our compressor achieves, respectively, 36.3dB and 38.6dB.

### 6.3. Interactive streaming and rendering

We evaluated the rendering performance of our framework on a number of interactive inspection sequences. The qualitative performance of our adaptive GPU ray-caster is illustrated in an accompanying video, using the dataset configurations selected in Fig. 4. Because of video frame capture constraints, the sequence is recorded using a window size of $1024 \times 576$ pixels. In all recorded sequences, we used a 1 voxel/pixel accuracy and a decompression budget of 64 bricks per subtree to drive the adaptive renderer.

As shown in the video, the system is fully interactive. It is

possible to translate, rotate, and scale the model as well as to change the transfer function, also for the time-varying models, without affecting rendering performances. Frame rates are generally above 10Hz, varying from 5Hz for closeups of semitransparent animated data to above 60Hz for overall views of moderately opaque objects, for which we can better exploit early ray termination and visibility culling. A few popping artifacts, caused by discrete resolution changes during adaptive loading and refinement, are visible in the video. As in other other adaptive techniques, they could be removed by blending between level of details using quadrilinear interpolation to continuously blend between resolution levels [CNLE09, IGM10].

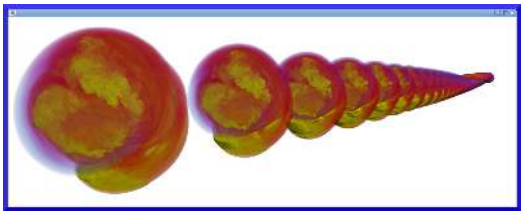

**Figure 6:** ***Multi-volume visualization.*** 60 *time stamps of the supernova. Working set composed of* 2051 *octree bricks, distributed among* 13 *subtrees rendered at* 10 *fps in a window of* $1280 \times 512$ *pixels.*

When the decompressed brick cache size is large enough to cache the entire working set, only few bricks/frame are decompressed at cache misses, and performance is similar to previous single-pass GPU raycasters working on uncompressed data [CNLE09, IGM10]. Such a configuration may only typically occur for static datasets and small rendering windows. Table 2 reports the decoding performance for the various benchmarked methods. The GPU reconstruction process of TD is the more costly, which reduces its applicability in compression-domain real-time renderers. TD can thus be used only in situations where few bricks/frame can be decoded. On the other hand, both HVQ and our method achieve reconstruction speeds compatible with the full decoding of volume working sets during rendering. The fastest decoder is HVQ, which sustains 2Gvox/s for all the settings. The decoding time of our method, which depends on sparsity, ranges from about 1.2Gvox/s for $(M = 4, S = 16)$ to 1.9Gvox/s for $(M = 6, S = 4)$. We can thus afford to render working sets of thousands of bricks at fully interactive frame rates even without any caching. In addition to improving responsiveness during refinement, fast decompression is particularly useful for time-varying datasets, where the decompressed cache is not big enough to keep all the decompressed animation, and thus data is completely overwritten from one frame to the other. Moreover, the high compression rates of our method permit to store in the GPU compressed cache entire datasets at high quality. For instance, the compressed Supernova datasets and $M = 6$ and $S = 8$ of Fig. 4 with PSNR $= 46.8$dB fits within less than 200MB of texture memory. For time-varying datasets, this allows the playback of full animations without unpredictable delays due to external data loading. These low GPU memory requirements can also be exploited for rendering multiple volumes within the same environment. Fig. 6 illustrates this concept with a frame captured during an interactive simultaneous inspection of all the 60 time-steps of the Supernova dataset. As for the animation case, it is possible to pre-load the entire dataset in texture-memory.

Networking scenarios are at the opposite side of the spectrum. In this case, compression is used to reduce bandwidth requirements throughout the entire pipe-line. The accompanying video illustrates this concept with the exploration of the Chameleon dataset over a 8Mbps ADSL connection. Compression settings were $M = 6$ and $S = 4$, leading to PSNR $= 49.26$dB for a bitrate of 0.36bps. As shown in the video, data quickly arrives to the client in an incremental way, and a full semi-transparent view of the Chameleon, requiring about 4MB of data, is displayed in about 5 seconds. By contrast, over 3.5 minutes would be needed to achieve the same result using uncompressed brick data.

Please refer to the video for additional results.

## 7. Conclusions and Future Work

We have presented a multi-resolution compression-domain GPU volume rendering architecture designed for interactive local and networked applications on commodity platforms. Compressed models are adaptively streamed and loaded on demand. The method supports high quality multi-sample rendering from general block-compressed data formats, extending to adaptive multi-resolution methods the capabilities of current deferred filtering solutions. The compression method introduced in this paper supports quick GPU-accelerated on-the-fly reconstruction through sparse linear combination of prototype blocks stored in an overcomplete dictionary. We have shown how extremely massive volumes can be processed off-line thanks to the application of the K-SVD method to a coreset representing the full block-based hierarchical representation of the input volume. Results in terms of rate-distortion are on par or significantly better than previous GPU accelerated methods. The principal limitation of the method, common to all contemporary block-based compression techniques, is that reconstruction artifacts at low bitrates manifest themselves as visible discontinuities between adjacent blocks. Our future work will concentrate on alleviating them by incorporating compression-aware post-process filters in the decompression and rendering process.

**Acknowledgments.** The authors would like to thank Alex Bronstein for helpful discussions. Datasets are courtesy of: Digital Morphology Project, the CTLab and the University of Texas, Austin; Visible Human Project; Dr. John Blondin at North Carolina State University through SciDAC Institute for Ultrascale Visualization. This work is partially supported by the EU FP7 Program under the DIVA project (290277). We also acknowledge the contribution of Sardinian Regional Authorities.

## References


[AEB06] AHARON M., ELAD M., BRUCKSTEIN A.: *rmk*-svd: An algorithm for designing overcomplete dictionaries for sparse representation. *IEEE Transactions on Signal Processing 54*, 11 (2006), 4311–4322. 2, 4

[AHPV05] AGARWAL P., HAR-PELED S., VARADARAJAN K.: Geometric approximation via coresets. *Combinatorial and Computational Geometry 52* (2005), 1–30. 2

[BNS01] BOADA I., NAVAZO I., SCOPIGNO R.: Multiresolution volume visualization with a texture-based octree. *Visual Computer 17*, 3 (2001), 185–197. 2

[CARKD99] COTTER S., ADLER R., RAO R., KREUTZ-DELGADO K.: Forward sequential algorithms for best basis selection. In *Proc. Vision, Image and Signal Processing* (1999), vol. 146, IET, pp. 235–244. 4

[CNLE09] CRASSIN C., NEYRET F., LEFEBVRE S., EISEMANN E.: Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proc. I3D* (2009), pp. 15–22. 2, 3, 5, 6, 9

[Cra] CRAIGHEAD M.: Gl_nv_texture_compression_vtc. OpenGL Extension Registry. 2

[CS07] CZUMAJ A., SOHLER C.: Sublinear-time approximation algorithms for clustering via random sampling. *Random Structures & Algorithms 30*, 1-2 (2007), 226–256. 2

[Ela08] ELAD M.: *Sparse and Redundant Representations*. Springer, 2008. 2, 4

[FAM*05] FOUT N., AKIBA H., MA K., LEFOHN A., KNISS J.: High-quality rendering of compressed volume data formats. In *Proc. EG/IEEE Symposium on Visualization* (2005). 2, 3

[FFS11] FEIGIN M., FELDMAN D., SOCHEN N.: From high definition image to low space optimization. In *Proc. SSVM* (2011). 2

[FL11] FELDMAN D., LANGBERG M.: A unified framework for approximating and clustering data. In *Proc. 43rd annual ACM symposium on Theory of computing* (2011), ACM, pp. 569–578. 2

[FM07] FOUT N., MA K.-L.: Transform coding for hardware-accelerated volume rendering. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (2007), 1600 – 1607. 1, 2, 8

[GMI08] GOBBETTI E., MARTON F., IGLESIAS GUITIÁN J. A.: A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *Visual Computer 24*, 7–9 (2008), 797–806. 2, 3, 5

[GS04] GUTHE S., STRASSER W.: Advanced techniques for high quality multiresolution volume rendering. *Computers & Graphics 28* (2004), 51–58. 2

[HQK05] HONG W., QIU F., KAUFMAN A.: GPU-based object-order ray-casting for large datasets. In *Proc. Volume Graphics* (2005), pp. 177–186. 2

[IGM10] IGLESIAS GUITIÁN J., GOBBETTI E., MARTON F.: View-dependent exploration of massive volumetric models on large scale light field displays. *The Visual Computer 26*, 6–8 (2010), 1037–1047. 2, 3, 6, 9

[KE02] KRAUS M., ERTL T.: Adaptive texture maps. In *Proc. Graphics Hardware* (2002), pp. 7–15. 2

[KWAH06] KAEHLER R., WISE J., ABEL T., HEGE H.-C.: GPU-assisted raycasting for cosmological adaptive mesh refinement simulations. In *Proc. Volume Graphics* (2006), pp. 103–110. 2

[LHJ99] LAMAR E. C., HAMANN B., JOY K. I.: Multiresolution techniques for interactive texture-based volume visualization. In *IEEE Visualization* (1999), pp. 355–362. 2

[Lju06] LJUNG P.: Adaptive sampling in single pass, GPU-based raycasting of multiresolution volumes. In *Proc. Volume Graphics* (2006), pp. 39–46. 2

[MBPS10] MAIRAL J., BACH F., PONCE J., SAPIRO G.: Online learning for matrix factorization and sparse coding. *The Journal of Machine Learning Research 11* (2010), 19–60. 2

[NIH08] NAGAYASU D., INO F., HAGIHARA K.: A decompression pipeline for accelerating out-of-core volume rendering of time-varying data. *Computers & Graphics 32*, 3 (2008), 350–362. 2

[PK09] PARYS R., KNITTEL G.: Giga-voxel rendering from compressed data on a display wall. In *Proc. WSCG* (2009), pp. 73–80. 2

[RBE10] RUBINSTEIN R., BRUCKSTEIN A., ELAD M.: Dictionaries for sparse representation modeling. *Proceedings of the IEEE 98*, 6 (2010), 1045–1057. 2

[RK09] RUITERS R., KLEIN R.: Btf compression via sparse tensor decomposition. In *Computer Graphics Forum* (2009), vol. 28, Wiley Online Library, pp. 1181–1188. 2

[RZE08] RUBINSTEIN R., ZIBULEVSKY M., ELAD M.: *Efficient implementation of the K-SVD algorithm using batch orthogonal matching pursuit*. Tech. rep., CS Technion, 2008. 4

[SE10] SKRETTING K., ENGAN K.: Recursive least squares dictionary learning algorithm. *IEEE Transactions on Signal Processing 58*, 4 (2010), 2121–2130. 2

[SIM*11] SUTER S., IGLESIAS GUITIÁN J., MARTON F., AGUS M., ELSENER A., ZOLLIKOFER C., GOPI M., GOBBETTI E., PAJAROLA R.: Interactive multiscale tensor reconstruction for multiresolution volume visualization. *IEEE Transactions on Visualization and Computer Graphics* (2011). 2, 8

[SW03] SCHNEIDER J., WESTERMANN R.: Compression domain volume rendering. In *Proc. IEEE Visualization* (2003), pp. 293–300. 2, 8

[Vit85] VITTER J.: Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS) 11*, 1 (1985), 37–57. 5

[YNV08] YELA H., NAVAZO I., VAZQUEZ P.: S3dc: A 3dc-based volume compression algorithm. In *Proc. CEIG* (2008). 2