This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2014.2346324, IEEE Transactions on Visualization and Computer Graphics

Sparse PDF Volumes for Consistent Multi-Resolution Volume Rendering

Ronell Sicat, Jens Krüger, Torsten Möller, and Markus Hadwiger

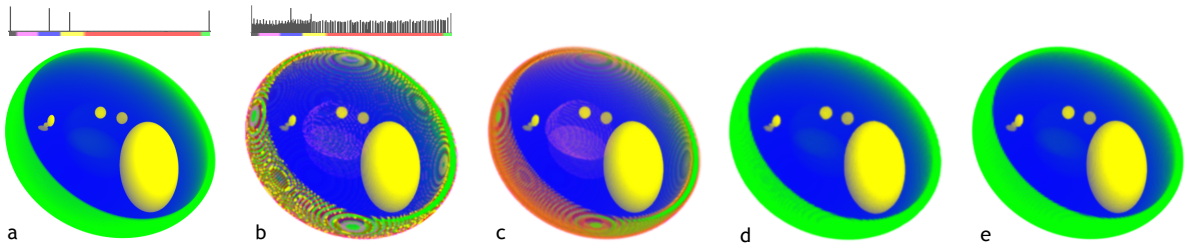

Fig. 1: **Sparse pdf volumes** store probability density functions (pdfs) of voxel neighborhoods in multi-resolution volumes. This enables consistent multi-resolution volume rendering, i.e., equivalent visualizations independent of resolution level. We compare (a) the original 512^3 Shepp-Logan volume against (b,c,d,e) a down-sampled 128^3 volume using (b) standard low-pass filtering and down-sampling (1 byte/voxel), (c) one Gaussian per voxel (3 bytes/voxel), (d) our sparse pdf volume (4 bytes/voxel), and (e) a full histogram (256 bins) per voxel as "ground truth" (512 bytes/voxel). The PSNR between the pdfs encoded in (d) vs. (e) is 44 dB.

Abstract—This paper presents a new multi-resolution volume representation called *sparse pdf volumes*, which enables consistent multi-resolution volume rendering based on probability density functions (pdfs) of voxel neighborhoods. These pdfs are defined in the 4D domain jointly comprising the 3D volume and its 1D intensity range. Crucially, the computation of sparse pdf volumes exploits data coherence in 4D, resulting in a sparse representation with surprisingly low storage requirements. At run time, we dynamically apply transfer functions to the pdfs using simple and fast convolutions. Whereas standard low-pass filtering and down-sampling incur visible differences between resolution levels, the use of pdfs facilitates consistent results independent of the resolution level used. We describe the efficient out-of-core computation of large-scale sparse pdf volumes, using a novel iterative simplification procedure of a mixture of 4D Gaussians. Finally, our data structure is optimized to facilitate interactive multi-resolution volume rendering on GPUs.

Index Terms—Multi-resolution representations, sparse approximation, pursuit algorithms, large-scale volume rendering

1 Introduction

The resolution of volume data has increased significantly over the last decade [1, 26], due to high-resolution data acquisition modalities such as modern CT scanners [25] or electron microscopes [16], and large-scale simulations [5]. However, although the resolution of display hardware has also increased considerably, the gap between the resolution of large-scale volume data and practically feasible output resolutions for visualization is often significant. The most common approach to alleviating this problem is the use of multi-resolution techniques, i.e., the representation of volume data with multiple low-pass filtered and successively down-sampled resolution levels. Multi-resolution volume rendering using data structures such as octrees or 3D mipmaps considerably helps with (1) avoiding aliasing artifacts due to under-sampling, and (2) speeding up the visualization by decreasing the amount of data that needs to be accessed for rendering.

However, applying a low-pass filter and down-sampling the original volume changes it by replacing the original data points by fewer data points that represent weighted averages of the original data. Figs. 1 (a) and (b) illustrate this problem by showing the visual differences and the different histograms of data values between (a) the original volume, and (b) the volume down-sampled by a factor of four in each dimension. These differences can be explained by understanding that, in (b), the transfer function is applied to data with a different distribution, i.e., the histogram shown on top. This problem has been observed before [36, 40], but it is difficult to solve without either a significant increase in memory footprint or only moderate gains in quality.

One often overlooked but important consequence of working with low-pass filtered data is that the application of non-linear operators, such as a transfer function, is incompatible and yields visualization results that are *inconsistent* between resolution levels. From a user perspective, this has the highly undesirable consequence that the visual result of an interactive transfer function specification depends on which resolution level the user was looking at when designing the transfer function. Together with the fact that a transfer function for high-resolution volume data can often either only be designed for a zoomed-out, significantly down-sampled view of the whole volume, or a zoomed-in, but partial view, this issue is becoming more and more problematic when working with large-scale volume data. In this paper, we refer to this phenomenon as *inconsistency artifacts* in multi-resolution volume rendering. In order to avoid erroneous data analysis based on transfer functions that were specified for down-sampled data, it is important to avoid or at least reduce these inconsistency artifacts.

To tackle these challenges, we present the following contributions for consistent multi-resolution volume rendering based on probability density functions (pdfs) of voxel neighborhoods (footprints):

(1) A compact, sparse representation of pdfs. The crucial insight is that instead of storing individual 1D pdfs, we use 4D pdfs in the joint space $\times$ range domain of the volume. This enables a quality similar to storing full histograms, but requires considerably less storage.

(2) The *sparse pdf volume* data structure that is optimized for efficient parallel volume classification and rendering on GPUs. Our approach uses only simple and fast convolutions at run time.

(3) A novel approach for computing a sparse 4D function (pdf) approximation in a multi-resolution hierarchy via a greedy pursuit algorithm for the iterative simplification of 4D Gaussian mixtures.

(4) An out-of-core framework for efficient parallel computation of sparse pdf volumes for large-scale volume data.

- *Ronell Sicat and Markus Hadwiger are with King Abdullah University of Science and Technology (KAUST). E-mail: {ronell.sicat, markus.hadwiger}@kaust.edu.sa.*
- *Jens Krüger is with the University of Duisburg-Essen. E-mail: jens.krueger@uni-due.de.*
- *Torsten Möller is with the University of Vienna. E-mail: torsten.moeller@univie.ac.at.*

Manuscript received 31 Mar. 2014; accepted 1 Aug. 2014; date of publication xx xxx 2014; date of current version xx xxx 2014. For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2014.2346324, IEEE Transactions on Visualization and Computer Graphics

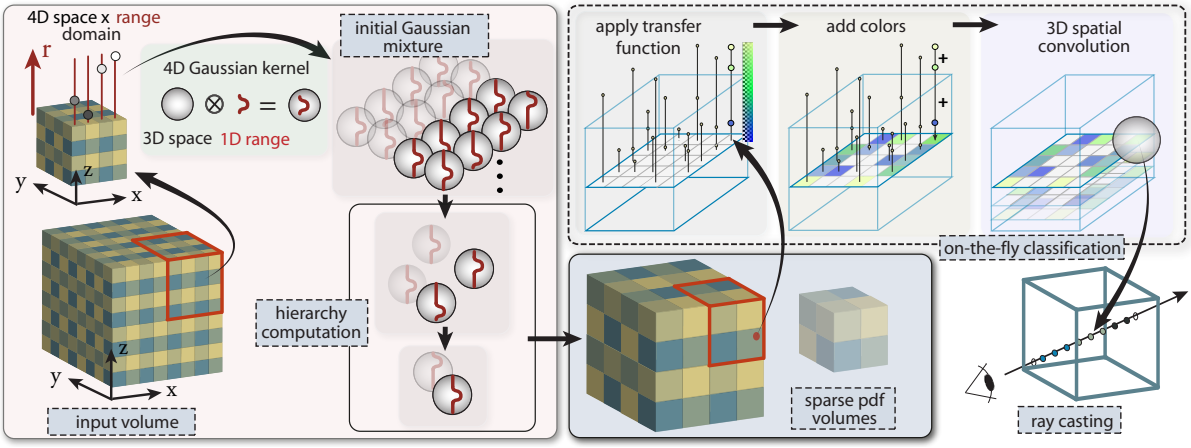

Fig. 2: **Method overview.** *Pre-computation* (left side): We map the 3D input volume (brick by brick) to a pdf in the 4D joint space × range domain (Sec. 4.1). This pdf is represented as a mixture of 4D Gaussians, which is then iteratively simplified to compute a multi-resolution hierarchy of sparse 4D pdfs (Sec. 4.2). *Run time* (right side): The sparse pdf volume data structure storing the sparse 4D pdfs allows dynamically applying transfer functions through simple look-ups in pre-computed 1D tables, followed by 3D convolution (Sec. 4.3), for volume rendering.

2 Related Work

Multi-resolution volume rendering. The most common approach for handling large data in volume visualization is to use multi-resolution techniques [38], usually utilizing hierarchical data structures such as octrees [7, 14, 20, 31, 37] or 3D mipmaps [11, 16, 21]. These representations store iteratively pre-filtered and down-sampled versions of the original volume at discrete resolution levels. Furthermore, several methods use multiple levels-of-detail for a single volume rendering, i.e., they allow for mixed-resolution rendering [2, 20, 22, 37].

However, all of these approaches can suffer from inconsistency artifacts as described above. In mixed-resolution volume rendering, these problems additionally manifest as artifacts in the transition regions between adjacent regions that are rendered with different resolutions. In this paper, our main focus is achieving consistent results when switching between resolution levels. However, our approach would also further reduce transition artifacts in mixed-resolution rendering.

Volume compression. Our goal of using a sparse representation is not reducing the size of the volume itself as in compression techniques, but to represent more information, i.e., probability density functions in a 4D space, albeit in a compact way. Hence, we are not competing with 3D volume compression techniques, e.g., based on sparse coding [13, 39], tensor approximations [34], or compressing floating point values [23]. While compression has been applied to pdfs before [12], we view our approach more as a sparse representation that is optimized for rendering rather than as a general compression technique.

Distribution-based multi-resolution volume rendering. Younesy et al. [40] were the first to show that inconsistency artifacts in multi-resolution volume rendering can be reduced by applying the transfer function to histograms of voxel neighborhoods. However, storing full histograms, e.g., with 256 bins per voxel, incurs an impractical storage overhead. Younesy et al. then only stored the mean and standard deviation per voxel, yielding only moderate quality improvements. Histograms can also be approximated via a Gaussian Mixture Model with N components, storing N means, standard deviations, and weights per voxel [24]. However, the associated storage overhead can be impractical, and reconstruction during rendering is computationally expensive.

The hixel representation [36] represents the uncertainty in large-scale data sets via quantized histograms of voxel neighborhoods.

Hadwiger et al. [17] sparsely encoded pixel neighborhood distributions in their sparse pdf maps data structure to accurately apply non-linear image operations to multi-resolution gigapixel images. Instead of approximating distributions individually, they fitted 3D Gaussians in the combined space × range domain of the image in order to exploit coherence in this 3D domain. However, their approach required long pre-computation times of around two minutes per megapixel, using iterative Matching Pursuit [28] to fit Gaussians to sampled 3D distributions. In the present paper, we completely avoid a sampled intermediate representation and work completely in continuous 4D space.

Our approach builds on the work by Younesy et al. [40] and Hadwiger et al. [17]. We use voxel neighborhood distributions for consistent multi-resolution volume rendering and work in a higher-dimensional domain. Our sparse pdf volume representation is able to capture even multi-modal distributions without incurring impractical memory requirements, and enables fast reconstruction and classification via simple and fast convolutions at run time. Pre-processing is made scalable via iterative simplification of 4D Gaussian mixtures.

3 Basics

We first describe the challenge of consistent multi-resolution volume rendering in more detail, and define the basic model that we are using.

3.1 Consistent multi-resolution volume rendering

Multi-resolution volumes comprise a hierarchy of successively coarser resolutions computed from the original volume. This results in a discrete set of resolution levels, where we will denote the original volume as level ℓ_0. Resolution reduction is performed by first applying a low-pass filter (also called pre-filter [19]) to level ℓ_m, and then down-sampling to obtain level ℓ_{m+1}. It is common to reduce the resolution by a factor of two in each dimension from ℓ_m to ℓ_{m+1}, but other factors [30] or anisotropic reductions are also possible [16]. In this work, we will use a down-sampling factor of two. However, this is not an inherent restriction of our method, and other factors could also be used.

Applying a low-pass filter before down-sampling is crucial for avoiding aliasing artifacts. However, this filtering process substitutes each voxel by a weighted average of its neighborhood, where the weights depend on the filter kernel used. When one computes the histogram of a given resolution level ℓ_m (e.g., Fig. 1 (b)), it is easy to see that this process changes the *distribution* of values in the volume. More specifically, each voxel in level ℓ_m corresponds to the *footprint* [15] of this voxel in level ℓ_0, with the size of these footprints growing from level to level. An accurate representation of a given voxel would be the distribution of values in the voxel's footprint in the original volume. Nevertheless, standard multi-resolution approaches substitute this distribution by a single value, for example the mean of the distribution when a box filter is used for low-pass filtering.

An accurate representation of voxel footprint distributions would therefore provide the conceptual basis for *consistent* multi-resolution volume rendering. However, the first major obstacle to using this idea in practice is the storage overhead associated with each distribution. For quantized 8-bit volume data, each histogram can be stored as a 256-bin histogram, which is already impractical. Naturally, this problem becomes more acute with 12- or 16-bit data. A second major hurdle in practice is applying a transfer function to the distribution, which for an n-bin histogram also requires $\mathcal{O}(n)$ operations.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2014.2346324, IEEE Transactions on Visualization and Computer Graphics

3.2 Basic model

We formalize the above ideas by treating each voxel at position $\mathbf{p}$ in a multi-resolution volume as the realization of a random variable $X_\mathbf{p}$, with an associated probability density function $f_\mathbf{p}(r)$. The argument r corresponds to the *range* of the volume, and for intensity (single-channel) volumes, $f_\mathbf{p}(r)$ is a 1D function of r. Likewise, a 1D transfer function $t(r)$ is defined over the same r, i.e., the domain of the transfer function is the range of the volume. The function $t(r)$ is then applied to the voxel $X_\mathbf{p}$ by computing the *expectation* of t applied to $X_\mathbf{p}$:

$$E\left[t(X_\mathbf{p})\right] = \int t(r) f_\mathbf{p}(r) \mathrm{d}r. \tag{1}$$

We observe that from this viewpoint, standard multi-resolution volume rendering could be "emulated" by simply first computing the expectation (mean) of $X_\mathbf{p}$ as $E\left[X_\mathbf{p}\right]$, and then applying $t(r)$ to the obtained mean. That is, instead of computing $E\left[t(X_\mathbf{p})\right]$, we would first compute $E\left[X_\mathbf{p}\right]$, and then compute $t(E\left[X_\mathbf{p}\right])$. This is in essence what is done in standard multi-resolution volume rendering, where the $E\left[X_\mathbf{p}\right]$ are pre-computed and stored. In contrast, however, our goal is to store the $f_\mathbf{p}(r)$ instead. Then Eq. 1 allows applying the transfer function $t(r)$ directly to the complete distribution $f_\mathbf{p}(r)$ instead of to just its mean.

The big practical challenges now are (1) how all $f_\mathbf{p}(r)$ can be stored compactly, and (2) how Eq. 1 can be evaluated efficiently.

4 Method Overview

Fig. 2 depicts an overview of our method. The most crucial property of our approach is that it works in the 4D joint space $\times$ range domain of the 3D spatial volume domain together with its 1D intensity range. We define this 4D domain in Sec. 4.1. We then exploit the fact that even though volume data are usually not sparse in 3D, they are sparse with respect to a certain basis in 4D, even in a multi-resolution hierarchy.

A *sparse pdf volume* then comprises a hierarchy of resolution levels ℓ_m of progressively smoother functions in 4D, where each ℓ_m is represented by a mixture of 4D Gaussians. Crucially, the number of Gaussians in each level ℓ_m is proportional to its spatial resolution. We give an overview of this hierarchy in Sec. 4.2, and the details in Sec. 5. Sec. 6 describes the corresponding data structure for efficient storage, while at the same time facilitating fast parallel access on GPUs. Sec. 7 describes classification (via transfer functions) and volume rendering.

4.1 Joint 4D space $\times$ range domain

Usually, volume data comprise a three-dimensional scalar function $\mathrm{V}(x,y,z)$, whose domain is a subset of 3D space, $U \subset \mathbb{R}^3$, and whose range (or co-domain) is the 1D intensity axis $\mathbb{R}$:

$$\mathrm{V}(x,y,z) : U \subset \mathbb{R}^3 \mapsto \mathbb{R}. \tag{2}$$

We now join the domain and co-domain of V via the Cartesian product to obtain the 4D *joint* space $\times$ range domain of a new function $\hat{\mathrm{V}}$:

$$\hat{\mathrm{V}}(x,y,z,r) : \hat{U} \subset \mathbb{R}^3 \times \mathbb{R} \mapsto \mathbb{R}. \tag{3}$$

We define $\hat{\mathrm{V}}(x,y,z,r)$ as the "joint probability" of spatial locations (x,y,z) occurring together with a specific intensity value r, via:

$$\hat{\mathrm{V}}(x,y,z,r) := \begin{cases} \delta & \text{if } \exists\ \mathrm{V}(x,y,z) \text{ with } \mathrm{V}(x,y,z) = r, \\ 0 & \text{otherwise,} \end{cases} \tag{4}$$

where δ is the Dirac delta[1]. Instead of assuming exactly occurring values, we now take the viewpoint of *kernel density estimation* [33] and substitute each delta peak by a 4D Gaussian kernel $\mathrm{G}_{(\boldsymbol{\sigma}_s,\sigma_r)}$:

$$\hat{\mathrm{V}}(x,y,z,r) := \sum_{i=0}^{k-1} \mathrm{G}_{(\boldsymbol{\sigma}_s,\sigma_r)}\left((x,y,z,r) - \boldsymbol{\mu}_i\right), \tag{5}$$

[1]The Dirac delta integrates to 1. Note that in order to obtain a properly normalized 4D pdf, $\hat{\mathrm{V}}$ in Eq. 4 would have to be multiplied by $1/k$, where k is the number of voxels. However, to avoid computing with unnecessarily small numbers, we neglect normalization for now and normalize later where needed.

where each voxel i from the original volume with k voxels is mapped to a Gaussian $\mathrm{G}_{(\boldsymbol{\sigma}_s,\sigma_r)}(\cdot)$ in 4D, centered at $\boldsymbol{\mu}_i := (\mathbf{p}_i, r_i)$, corresponding to the voxel's original 3D position $\mathbf{p}_i := (x_i, y_i, z_i)$, and its 1D intensity value r_i. $\mathrm{G}_{(\boldsymbol{\sigma}_s,\sigma_r)}(\cdot)$ is a separable 4D kernel in space $\times$ range:

$$\mathrm{G}_{(\boldsymbol{\sigma}_s,\sigma_r)}(\mathbf{x}) := (G_{\sigma_s} \otimes G_{\sigma_s} \otimes G_{\sigma_s} \otimes G_{\sigma_r})(\mathbf{x}), \tag{6}$$

via the tensor product $\otimes$ of 1D Gaussians with spatial standard deviation σ_s (assuming isotropic voxels), and range standard deviation σ_r.

4.2 A hierarchy of 4D Gaussian mixtures

Our goal is now to compute an entire multi-resolution hierarchy, with each level ℓ_m defined similarly to Eq. 5, but as a more general mixture of k_m individually weighted 4D Gaussians, with weights c_i:

$$\hat{\mathrm{V}}_m(x,y,z,r) := \sum_{i=0}^{k_m-1} c_i \mathrm{G}_{(\boldsymbol{\sigma}_s,\sigma_r)}\left((x,y,z,r) - \boldsymbol{\mu}_i\right). \tag{7}$$

We want to emphasize that a major difference of our approach to a general Gaussian Mixture Model is that *all* of the Gaussians comprising a given $\hat{\mathrm{V}}_m$ are constrained to have the *same* standard deviation $(\boldsymbol{\sigma}_s, \sigma_r)$. This property is crucial to enabling the use of convolutions for efficiency, and not having to store the σ's of all Gaussians.

Our hierarchy will therefore be a set of n resolution levels $\{\hat{\mathrm{V}}_m\}$ with $m \in \{0,..,n-1\}$. Each $\hat{\mathrm{V}}_m$ is the 4D pdf of level ℓ_m in the form of Eq. 7, with k_m Gaussians $c_i \mathrm{G}_{(\boldsymbol{\sigma}_s,\sigma_r)}$ centered at $\boldsymbol{\mu}_i$. With $(\boldsymbol{\sigma}_s, \sigma_r)$ known, each $\hat{\mathrm{V}}_m$ can be stored solely as a set of k_m tuples $\{(\boldsymbol{\mu}_i, c_i)\}$.

4.2.1 Initial Gaussian mixture

We start the hierarchy with $\hat{\mathrm{V}}_0$ for resolution level ℓ_0. In $\hat{\mathrm{V}}_0$, we set the spatial standard deviation σ_s such that it corresponds to one voxel (we use $\sigma_s = 0.3$), and set the range standard deviation σ_r to a fixed value (we use $\sigma_r = 4/256$). We set all $c_i = \frac{1}{(\sqrt{2\pi})^4 \sigma_s^3 \sigma_r}$, so each Gaussian integrates to 1 (cf. Sec. 5.3). Note, however, that in all $\hat{\mathrm{V}}_m$ with $m > 0$, the coefficients c_i will be determined by minimizing an error function.

4.2.2 Hierarchy computation

We then compute the Gaussian mixture of each level $\hat{\mathrm{V}}_m$ with $m > 0$ from the preceding level $\hat{\mathrm{V}}_{m-1}$. In order to avoid incurring aliasing artifacts later on, we first have to low-pass filter $\hat{\mathrm{V}}_{m-1}$. This can be done very efficiently by simply updating the spatial standard deviation σ_s and the coefficients c_i accordingly. See Sec. 5.3.2 and Appendix A.2.

Our goal is now to represent $\hat{\mathrm{V}}_m$ with fewer Gaussians than $\hat{\mathrm{V}}_{m-1}$. If $\hat{\mathrm{V}}_{m-1}$ is represented by k_{m-1} Gaussians, and we would like to keep the typical down-sampling rate of 2^3 for 3D Cartesian volumes, we target $k_m = k_{m-1}/2^3$ Gaussians for $\hat{\mathrm{V}}_m$. We do this by computing a *sparse* approximation to $\hat{\mathrm{V}}_m$ via a greedy pursuit algorithm built on convolutions (Secs. 5.2 and 5.3) and mode finding (Sec. 5.4).

4.3 Volume classification from 4D representation

Given the representation of $\hat{\mathrm{V}}_m$ as in Eq. 7 (via the set $\{(\boldsymbol{\mu}_i, c_i)\}$), our goal for volume rendering is to be able to evaluate Eq. 1 for any voxel with a given position $\mathbf{p}$ via the corresponding voxel footprint pdf $f_\mathbf{p}(r)$.

In principle, each $f_\mathbf{p}(r)$ can be obtained from the 4D pdf given by Eq. 7 by extracting the "conditional probability" of r, given voxel position $\mathbf{p}$. This can be done by simply obtaining a function of r at $\mathbf{p}$ and re-normalizing. However, in order to facilitate fast parallel classification of entire voxel bricks, we employ a method that avoids the reconstruction of individual $f_\mathbf{p}(r)$ entirely, and apply the transfer function in parallel to all voxels in a brick via 1D look-ups and 3D convolution. This greatly facilitates efficient GPU implementation. See Sec. 7.

5 Sparse pdf Volume Computation

Our goal is the computation of each $\hat{\mathrm{V}}_m$ for resolution level ℓ_m with $m > 0$, such that it approximates the low-pass filtered mixture $\hat{\mathrm{V}}_{m-1}$ well. Both functions are given in the form of Eq. 7. The crucial goal is now to represent $\hat{\mathrm{V}}_m$ with fewer Gaussians than $\hat{\mathrm{V}}_{m-1}$, i.e., $k_m < k_{m-1}$.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2014.2346324, IEEE Transactions on Visualization and Computer Graphics

This is a reasonable assumption, because low-pass filtering makes these functions successively smoother. Our basic idea for this approximation is that each $\hat{\mathrm{V}}_m$ should be represented as a *sparse* signal, with respect to a chosen *dictionary* of *atoms* ("basis functions") [9]:

$$\min_{\mathbf{c}} \|\mathbf{c}\|_0 \quad \text{subject to} \quad \mathbf{H}\,\mathbf{c} = \mathbf{v}, \tag{8}$$

where $\mathbf{H}$ denotes the dictionary with the atoms viewed as column vectors, and $\mathbf{c}$ is the coefficient vector that determines the linear combination that should best approximate a signal $\mathbf{v}$, given $\mathbf{H}$. Our dictionary $\mathbf{H}$ consists of translates of Gaussians (see Sec. 5.3), and the target signal $\mathbf{v}$ to approximate is a chosen $\hat{\mathrm{V}}_m$ after low-pass filtering. In order to obtain a sparse representation, $\mathbf{c}$ should have as few non-zero elements as possible, which is indicated by the L_0 pseudo-norm $\|\cdot\|_0$.

5.1 Pursuit algorithms

In principle, finding the solution to Eq. 8 is an NP-hard problem [9]. However, *pursuit algorithms* compute good approximations using greedy iterative strategies to obtain a sparse $\mathbf{c}$. In each iteration, the atom from the dictionary $\mathbf{H}$ that approximates the current *residual* best is picked by *projecting* the residual into the dictionary [9]. Our method is based on the Matching Pursuit algorithm [28]. However, we employ a novel variant that is specialized for a dictionary that consists of translates of a Gaussian kernel, which facilitates the use of fast convolutions (Sec. 5.2). In contrast to earlier work for images [17], our strategy for volumes does not make use of a sampled pdf (i.e., histogram) representation, which is crucial to scaling from 2D image data to 3D volume data. Instead, our approach for volumes works almost completely in continuous 4D space, using continuous 4D Gaussian atoms.

5.2 Dictionary projection as convolution

For brevity, in the following explanations we describe the 1D case of approximating a general 1D function $g(x)$. However, one can always think of $g(x)$ below as corresponding to an actual 4D function $\hat{\mathrm{V}}_m$.

We want to *project* the function $g(x)$, which we want to approximate, onto dictionary atoms $h_\mu(x)$, where the parameter μ selects the atom. This projection is obtained via the inner product of the two functions. The inner product of two (real) functions g and h_μ is defined as:

$$\big(g, h_\mu\big) := \int_{-\infty}^{\infty} g(x)\, h_\mu(x)\, \mathrm{d}x. \tag{9}$$

The (scalar) coefficient of g with respect to h_μ is obtained from projection using Eq. 9 with a dictionary atom that has unit norm:

$$\|h_\mu\|_2 = 1, \tag{10}$$

where $\|\cdot\|_2$ denotes the L_2 norm induced by the inner product, i.e., $\|h\|_2 := \sqrt{(h,h)}$ (see [4, 9, 27]).

All of our dictionary atoms are translates of the same kernel $h(x)$, where h is symmetric around zero, i.e., $h(x) = h(-x)$. h_μ then denotes the dictionary atom centered at $x = \mu$, defined as:

$$h_\mu(x) := h(x - \mu). \tag{11}$$

This will allow us to carry out most operations as simple convolutions. The convolution (denoted by $\star$) of two functions g and h is defined as:

$$(g \star h)(x) := \int_{-\infty}^{\infty} g(t)\, h(x - t)\, \mathrm{d}t. \tag{12}$$

In order to determine the dictionary atom that approximates the function $g(x)$ best, we have to determine which atom results in the largest inner product [9], i.e., we must determine the maximizer of Eq. 9:

$$\max_{h_\mu} \big(g, h_\mu\big). \tag{13}$$

Because we define all dictionary atoms as translates of the same kernel $h(x)$, the maximum inner product of all h_μ can be computed from the convolution of g and h (compare Eqs. 9 and 12, using Eq. 11):

$$\max_{h_\mu} \big(g, h_\mu\big) = \max_x (g \star h)(x). \tag{14}$$

The value of x that maximizes the right-hand side of Eq. 14 is the maximizer of Eq. 13 with the h_μ where $\mu = x$. Thus, the crucial observation is that in order to find the dictionary element that approximates $g(x)$ best, we simply have to find the maximum of the function $(g \star h)(x)$.

5.3 Gaussian dictionaries and Gaussian mixtures

We define a Gaussian centered at $x = \mu$ with standard deviation σ as:

$$G_\sigma(x;\mu) := e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \quad \text{or abbreviated} \tag{15}$$

$$G_\sigma(x) := G_\sigma(x;0) \quad \text{if } \mu = 0. \tag{16}$$

Note that our definition of G_σ is not normalized. We do this because we will need different normalization weights for different purposes. A *weighted* Gaussian is then $c\,G_\sigma(x;\mu)$, with a scalar coefficient c.[2]

5.3.1 Gaussian dictionaries

We use a dictionary $h_\mu(x)$, with different μ, defined as Gaussians:

$$h_\mu(x) := c_h\, G_\sigma(x;\mu), \tag{17}$$

with $c_h = \pi^{-\frac{1}{4}}/\sqrt{\sigma}$, so that $\|h_\mu(x)\|_2 = 1$. We intentionally use dictionaries with a fixed standard deviation σ for all $h_\mu(x)$. This is crucial in order to enable simple convolutions to be used both for dictionary projection (Sec. 5.2), and for classification at run time (Sec. 7).

5.3.2 $g(x)$ as Gaussian mixture

The function $g(x)$ that we want to approximate is given as a mixture of k Gaussians with identical σ, but different weights c_i and positions μ_i:

$$g(x) := \sum_{i=0}^{k-1} c_i\, G_\sigma(x;\mu_i). \tag{18}$$

Our approach needs to perform two main operations on $g(x)$:

1. Low-pass filter $g(x)$ before spatial down-sampling.
2. Compute the maximum of the convolution from Eq. 14 for dictionary projection (Sec. 5.2), with $h_\mu(x)$ as defined in Eq. 17.

Both of these operations can be computed directly on $g(x)$ given as a Gaussian mixture (Eq. 18). We operate directly on the mixture represented by the set $\{(\boldsymbol{\mu}_i, c_i)\}$. We simply modify the c_i and the (global) standard deviation σ that is associated with the mixture. All positions $\boldsymbol{\mu}_i$ stay the same. For details see Appendix A.1 (convolution), Appendix A.2 (low-pass filtering), and Appendix A.3 (projection).

5.4 Pursuit via mode finding

We employ a greedy pursuit algorithm for the computation of the sparse approximation of each $\hat{\mathrm{V}}_m$ given in the form of Eq. 7. The Matching Pursuit algorithm [28] computes a greedy signal approximation in an iterative way, where in each iteration a *residual* function is reduced until a pre-defined L_2 error threshold is reached. In the beginning, the residual is initialized to the original signal. Then, in every iteration the dictionary atom that has the largest *inner product* with the current residual is picked, because it approximates it best [28].

The corresponding coefficient is given by the same inner product, since it corresponds to the *projection* of the residual onto the chosen atom. In every iteration, the contribution of each atom is subtracted out from the residual, and the whole procedure starts again until the error is small enough, yielding the desired sparse signal representation [9].

[2] For example, a Gaussian pdf $f(x)$ requires $\int_{\mathbb{R}} f(x)\,\mathrm{d}x = 1$, and therefore $f(x) = \frac{1}{\sqrt{2\pi\sigma}}\, G_\sigma(x;\mu)$, where $c = \frac{1}{\sqrt{2\pi\sigma}}$. See also Appendix A.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2014.2346324, IEEE Transactions on Visualization and Computer Graphics

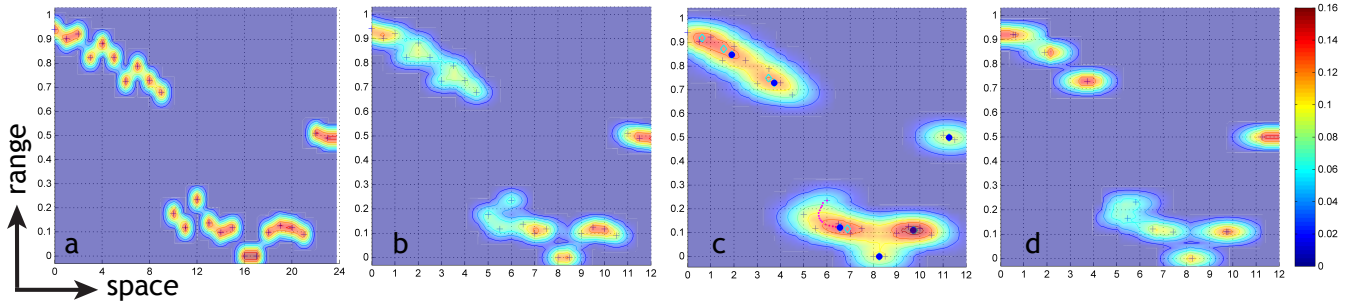

Fig. 3: **Parallel pursuit via mode finding** starts from (a) the input Gaussian mixture (centroids denoted by +), which is the kernel density estimate of the original volume, or a previously simplified mixture. (b) The mixture is low-pass filtered in the spatial domain. (c) The intermediate mixture for mode finding is obtained via convolution with the dictionary kernel. We then perform parallel pursuit iterations in this mixture. We find all its modes (• and ◇) in parallel, but retain only the largest *non-overlapping* modes (•). A fixed-point mode search in the first iteration is illustrated for each step (•) from one example centroid. (d) The output Gaussian mixture consists of one component per retained mode.

Algorithm 1 shows these steps in our framework. Each atom in our dictionary is a Gaussian $h_{\boldsymbol{\mu}}(x)$ as defined in Eq. 17. In 4D, each projection into the dictionary is a tuple $(\boldsymbol{\mu}, c)$, where $\boldsymbol{\mu}$ selects the dictionary atom, and c is the corresponding coefficient. All our 4D Gaussian mixtures then correspond to sets $G = \{(\boldsymbol{\mu}_i, c_i)\}$ of such tuples.

5.4.1 Projection in 4D using mode finding

A fundamental step in Algorithm 1 is finding the atom that results in the largest inner product with the residual (see Eq. 14). Instead of looking at each atom individually—as in standard Matching Pursuit—we exploit the property that the atom we are looking for corresponds to the maximum of the right-hand side of Eq. 14, which is attained at one of its modes. The convolution in Eq. 14 is derived in Appendix A.3.

We can therefore perform the projection step via a *mode finding* procedure in a Gaussian mixture. For mode finding, we employ the fixed-point iteration approach described by Carreira-Perpiñán [3]. Another similar alternative would be to use a mean shift procedure [6].

Fig. 3 depicts the individual steps of our approach for a simplified example illustrated in 2D (1D space × 1D range). We start with a given mixture $\hat{\mathrm{V}}_m$, subject it to spatial low-pass filtering to prevent aliasing (Eq. 35), further convolve it with the dictionary kernel h (Eq. 37), and then find the modes of this intermediate mixture. We start the mode finding procedure at the known modes of the mixture $\hat{\mathrm{V}}_m$, and iterate to stationary points of the intermediate mixture. At this stage, it is quite common that nearby starting points converge to the same mode. The intermediate mixture corresponds to the function $g \star h$ in Eq. 14, and the coefficient that we are looking for (c in Algorithm 1) is the function value at the 4D position of the maximum mode.

5.4.2 Parallel pursuit via mode finding

Standard Matching Pursuit [28] is a completely sequential algorithm, where only one atom is selected in each iteration. In our context, this means determining the *maximum mode* of the intermediate mixture, as described above. However, in order to speed up the fitting process, we want to perform multiple mode searches in parallel in each iteration.

Algorithm 1 Approximating $g(x)$ with Matching Pursuit

Input: $g(x)$ given as Gaussian mixture (Eq. 18) $G = \{(\boldsymbol{\mu}_i, c_i)\}$
Output: Simplified Gaussian mixture $\tilde{g}(x)$ (Eq. 18) $\tilde{G} = \{(\boldsymbol{\mu}_k, c_k)\}$

1: Set output mixture $\tilde{g}(x) = 0$; $[\tilde{G} \leftarrow \emptyset]$;
2: Set residual $r(x) = g(x)$; $[R \leftarrow G]$;
3: Set L_2 approximation error $E_2 = \|r(x)\|_2$;
4: **while** $E_2 > \varepsilon$ **do**
5: Find maximizer $\max_{h_\mu}(r, h_\mu)$ as $c = \max_\mu (r \star h)(\mu)$ (Eqs. 14 and 37 with $r(x)$ instead of $g(x)$; Sec. 5.4.1)
6: Grow mixture $\tilde{g}(x) \leftarrow \tilde{g}(x) + c\, h_\mu(x)$; $[\tilde{G} \leftarrow \tilde{G} \cup \{(\boldsymbol{\mu}, c)\}]$;
7: Reduce residual $r(x) \leftarrow r(x) - c\, h_\mu(x)$; $[R \leftarrow R \cup \{(\boldsymbol{\mu}, -c)\}]$;
8: Re-compute approximation error $E_2 = \|r(x)\|_2$;
9: **end while**

In each iteration, we would like to find several *non-overlapping* modes in parallel, in order to select multiple atoms in parallel. We define non-overlapping modes such that the centroids of the atoms are not within distance $\omega\sigma$ of one another, where σ is the standard deviation of the atoms, and ω is a configurable overlap parameter. Setting ω appropriately ensures that the corresponding coefficients do not interfere with each other, because adding multiple overlapping atoms to the mixture at the same time incurs fitting errors due to incorrect coefficients. For overlapping modes, we always retain the largest one and discard the rest. The non-overlapping modes chosen in each iteration are added to the target mixture, and the residual is updated by subtracting the same atoms (Algorithm 1). The next iteration then continues from the previously known modes plus the newly added modes.

We found that this approach produces good results. However, in order to further speed up the pursuit process, we introduce two simple heuristics. Our first heuristic is to set ω to a value that allows minimal overlap among parallel coefficients. In our computations, we typically set $\omega = 3$ for the spatial dimensions, and $\omega = 2$ for the range. This allows us to put more parallel coefficients in each pursuit iteration with the trade-off of incurring a small error in the coefficient values. Note that setting ω large enough so that the overlap region covers the whole domain is equivalent to standard serial Matching Pursuit.

The second heuristic that we use is to optionally reduce the number of parallel searches that we do after the first pursuit iteration. The reason for this is that as the number of Gaussians in the residual grows, the parallel mode search also becomes slower, since (1) ideally to find all the modes, we have to do more searches starting from each Gaussian centroid in the residual, and (2) there are more Gaussian components in the mixture that need to be evaluated during the search. Conceptually, reducing the number of starting positions for the parallel mode search should have a minimal impact on the method since most of the searches converge to the same mode. Currently we pick around $1/8$ of the possible starting positions in each iteration and just change the starting positions in the next iteration by making them spatially well distributed, i.e., we rotate over even and odd combinations of the x, y, and z positions of the starting points. We only apply this heuristic after doing the first parallel mode search iteration where we use all the possible starting points, i.e., all modes of the initial residual.

6 Sparse PDF Volume Data Structure

Fig. 4 illustrates the data structure that we use to compactly encode the $\hat{\mathrm{V}}_m$ of a sparse pdf volume. We assume a *bricked* volume to facilitate large-scale out-of-core volume rendering. That is, the original volume (resolution level ℓ_0) is subdivided into bricks of fixed size. Our current implementation uses a brick size of 64^3 voxels, plus four ghost voxels in each spatial dimension. Using smaller brick sizes incurs an impractical storage overhead for ghost voxels [11], while larger brick sizes lead to slower pre-processing, since Matching Pursuit has quadratic complexity. All bricks of level ℓ_0 are stored in the usual way, with one scalar per voxel. The bricks of all levels ℓ_m with $m > 0$ are stored in the following sparse pdf volume encoding: We first sort the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2014.2346324, IEEE Transactions on Visualization and Computer Graphics

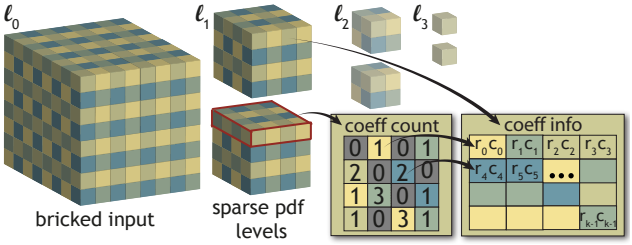

Fig. 4: **The sparse pdf volume data structure** consists of the bricked input volume (ℓ_0) and the coarser resolution levels ℓ_m with $m > 0$. For each of the latter, we store (1) a *coefficient count* brick with one count per voxel $\mathbf{p}$, and (2) a *coefficient info* array. Together, these encode the list of 4D Gaussian mixture components for each coarse level.

set of mixture components $\{(\boldsymbol{\mu}_i, c_i)\} = \{(\mathbf{p}_i, r_i, c_i)\}$ comprising $\hat{\mathrm{V}}_m$ according to *spatial voxel position* $\mathbf{p}$. For each voxel $\mathbf{p}$, we also count how many $(\mathbf{p}_i, r_i, c_i)$ with $\mathbf{p} = \mathbf{p}_i$ there are, and store this count in a *coefficient count* brick at position $\mathbf{p}$. We now drop the $\mathbf{p}_i$ coordinate from each $(\mathbf{p}_i, r_i, c_i)$, and store the tuples (r_i, c_i) in a *coefficient info* array. We do not need to explicitly store the $\mathbf{p}_i$, because the tuples (r_i, c_i) of each $\mathbf{p}$ can be indexed according to the count of tuples of $\mathbf{p}$ (see below). This results in the following data structure per brick (see Fig. 4):

1. A *coefficient count* brick stores one 8-bit count per voxel $\mathbf{p}$. This is the number of coefficients that are associated with the same *spatial* position $\mathbf{p}$ as the voxel, i.e., $\#(\mathbf{p}_i, r_i, c_i)$ with $\mathbf{p} = \mathbf{p}_i$.

2. A *coefficient info* array stores all tuples (r_i, c_i) of the brick, with a 16-bit half float each for r_i and c_i, respectively. All tuples from the same voxel $\mathbf{p}$ are stored contiguously in memory.

Furthermore, for 8-bit data sets we can quantize the r_i to 8 bits. The complete encoding then requires: For level ℓ_0: 1 byte per voxel. For all ℓ_m with $m > 0$: 1 byte per voxel, plus 3 bytes per coefficient. This encoding results in sparse pdf volume sizes that are still similar to standard representations (see Table 3), while encoding full pdfs.

In order to save memory, we do not store indexes to the tuples (r_i, c_i) on disk. However, at run time, for each $\mathbf{p}$ we need to be able to access all (r_i, c_i) that originally came from a tuple $(\mathbf{p}_i, r_i, c_i)$ with $\mathbf{p} = \mathbf{p}_i$. Therefore, after loading the sparse pdf data structure for rendering, we compute the *prefix sum* [18] of the coefficient count brick. This results in an *index* brick, where the entry at each voxel $\mathbf{p}$ points to the corresponding location of the first tuple (r_i, c_i) of $\mathbf{p}$ in the coefficient info array. All other tuples of $\mathbf{p}$ are located in consecutive memory locations, and their count is given by the coefficient count brick entry $\mathbf{p}$.

7 Run Time Classification

This section describes how Eq. 1 can be evaluated using our representation of 4D pdfs as 4D Gaussian mixtures (Eq. 7) instead of requiring individual 1D pdfs $f_{\mathbf{p}}(r)$. We first derive the basic method (Sec. 7.1), and then describe its efficient implementation in practice (Sec. 7.2).

7.1 Basic method

Our goal at run time is to apply a transfer function $t(r)$ to the data at any given voxel position $\mathbf{p}$. According to Eq. 1, we do this by applying $t(r)$ to the 1D voxel footprint pdf $f_{\mathbf{p}}(r)$. The latter is now contained in the 4D Gaussian mixture given by Eq. 7. We therefore insert Eq. 7 into Eq. 1, and evaluate for a fixed 3D voxel position $\mathbf{p} := (x, y, z)$:

$$E\left[t(X_{\mathbf{p}})\right] = \int t(r) f_{\mathbf{p}}(r) \mathrm{d}r, \tag{19}$$

$$= \int t(r) \frac{1}{w_{\mathbf{p}}} \sum_{i=0}^{k-1} c_i \, \mathrm{G}_{(\boldsymbol{\sigma}_s, \sigma_r)}\left((\mathbf{p}, r) - \boldsymbol{\mu}_i\right) \mathrm{d}r, \tag{20}$$

where $\mathrm{G}_{(\boldsymbol{\sigma}_s, \sigma_r)} = \mathrm{G}_{\boldsymbol{\sigma}_s} \otimes G_{\sigma_r}$ is a separable 4D Gaussian kernel, and our expression for $f_{\mathbf{p}}(r)$ is a 1D pdf due to the normalization factor $w_{\mathbf{p}}$:

$$w_{\mathbf{p}} := \int \sum_{i=0}^{k-1} c_i \, \mathrm{G}_{(\boldsymbol{\sigma}_s, \sigma_r)}\left((\mathbf{p}, r) - \boldsymbol{\mu}_i\right) \mathrm{d}r. \tag{21}$$

If we now split up $\boldsymbol{\mu}_i =: (\mathbf{p}_i, r_i)$ into spatial $(\mathbf{p}_i)$ and range (r_i) coordinates, respectively, and exploit that $\mathrm{G}_{(\boldsymbol{\sigma}_s, \sigma_r)}$ is separable, we get:

$$E\left[t(X_{\mathbf{p}})\right] = \frac{1}{w_{\mathbf{p}}} \sum_{i=0}^{k-1} c_i \, \mathrm{G}_{\boldsymbol{\sigma}_s}(\mathbf{p} - \mathbf{p}_i) \int t(r) G_{\sigma_r}(r - r_i) \mathrm{d}r. \tag{22}$$

Noting that all kernels G_{σ_r} are the same, we can write the integral in Eq. 22 as a convolution, which significantly simplifies the equation to:

$$E\left[t(X_{\mathbf{p}})\right] = \frac{1}{w_{\mathbf{p}}} \sum_{i=0}^{k-1} c_i \, \mathrm{G}_{\boldsymbol{\sigma}_s}(\mathbf{p} - \mathbf{p}_i) \tilde{t}(r_i), \tag{23}$$

where the new function $\tilde{t}(r)$ is defined as follows (cf. Eq. 12):

$$\tilde{t}(r) := (t \star G_{\sigma_r})(r) = \int_a^b t(x) G_{\sigma_r}(x - r) \mathrm{d}x, \tag{24}$$

where we have now denoted the interval of integration as $[a, b]$, corresponding to the domain of $t(r)$, e.g., $r \in [0, 1] \subset \mathbb{R}$. (Note that $G_{\sigma_r}(x) = G_{\sigma_r}(-x)$, and therefore $G_{\sigma_r}(x - r) = G_{\sigma_r}(r - x)$).

Similarly, Eq. 21 simplifies to (cf. Eq. 23):

$$w_{\mathbf{p}} = \sum_{i=0}^{k-1} c_i \, \mathrm{G}_{\boldsymbol{\sigma}_s}(\mathbf{p} - \mathbf{p}_i) \bar{G}_{\sigma_r}(r_i), \tag{25}$$

where the new function $\bar{G}_{\sigma_r}(r)$ is defined as (cf. Eq. 24):

$$\bar{G}_{\sigma_r}(r) := \int_a^b G_{\sigma_r}(x - r) \, \mathrm{d}x, \tag{26}$$

which is the integral of $G_{\sigma_r}(\cdot)$ centered at r and clamped to $r \in [a, b]$.

These derivations now yield the crucial observation that $\tilde{t}(r)$ can be pre-computed via a *single* 1D convolution (Eq. 24) for a given transfer function $t(r)$, and that Eq. 23 can be computed as a sum of simple look-ups and 3D (*not* 4D) convolutions. Similarly for $\bar{G}_{\sigma_r}(r)$. This makes run time classification very efficient and easy to implement.

7.2 Practical implementation

We now denote the spatial 3D neighborhood of voxel $\mathbf{p}$ as $\mathcal{N}(\mathbf{p})$, and obtain Gaussian mixture coefficients $(\boldsymbol{\mu}_i, c_i) = (\mathbf{p}_i, r_i, c_i)$ from the data structure described in Sec. 6. We can then write Eq. 23 as:

$$E\left[t(X_{\mathbf{p}})\right] = \frac{1}{w_{\mathbf{p}}} \sum_{\mathbf{q} \in \mathcal{N}(\mathbf{p})} \mathrm{G}_{\boldsymbol{\sigma}_s}(\mathbf{p} - \mathbf{q}) \sum_{\substack{(\mathbf{q}_i, r_i, c_i) \\ \mathbf{q} = \mathbf{q}_i}} c_i \tilde{t}(r_i). \tag{27}$$

We note that the second sum in Eq. 27 can be computed once per voxel $\mathbf{p}$, and then used in all spatial convolutions involving this voxel. This enables evaluating Eq. 27 as follows. We first use simple look-ups and summations to compute two intermediate bricks $T(\mathbf{p})$ and $N(\mathbf{p})$:

$$T(\mathbf{p}) := \sum_{\substack{(\mathbf{p}_i, r_i, c_i) \\ \mathbf{p} = \mathbf{p}_i}} c_i \tilde{t}(r_i), \quad \text{and} \quad N(\mathbf{p}) := \sum_{\substack{(\mathbf{p}_i, r_i, c_i) \\ \mathbf{p} = \mathbf{p}_i}} c_i \bar{G}_{\sigma_r}(r_i). \tag{28}$$

Then, both bricks $T(\mathbf{p})$ and $N(\mathbf{p})$ are convolved in the 3D spatial domain with $\mathrm{G}_{\boldsymbol{\sigma}_s}(\cdot)$, and the classified output brick is obtained by dividing each voxel in T by the corresponding normalization factor in N:

$$E\left[t(X_{\mathbf{p}})\right] = \frac{\left(T \star \mathrm{G}_{\boldsymbol{\sigma}_s}\right)(\mathbf{p})}{\left(N \star \mathrm{G}_{\boldsymbol{\sigma}_s}\right)(\mathbf{p})}. \tag{29}$$

We pre-compute both $\tilde{t}(r)$ (Eq. 24) and $\bar{G}_{\sigma_r}(r)$ (Eq. 26), storing each in a 1D look-up table. Computationally, for each voxel we only have to sum over all look-ups $\tilde{t}(r_i)$ and $\bar{G}_{\sigma_r}(r_i)$ per r_i, perform a single 3D convolution for T and N each, followed by one division per voxel.

In summary, despite its simplicity, this method (using Eqs. 28 and 29) performs accurate classification with any transfer function $t(r)$ for all voxels in a brick, as defined by Eq. 1 and Eq. 20 for each voxel.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2014.2346324, IEEE Transactions on Visualization and Computer Graphics

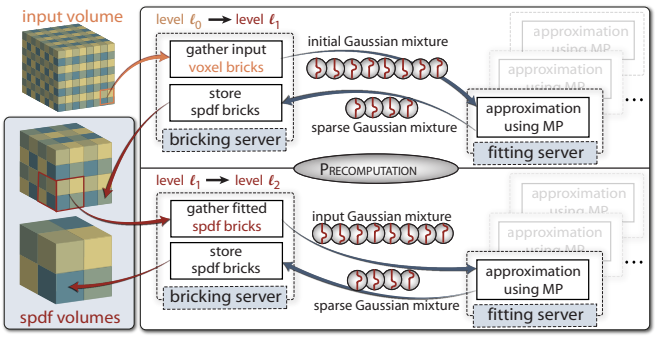

Fig. 5: **Out-of-core distributed pre-processing pipeline.** From the input volume bricks, the bricking server creates a fitting request with the initial 4D Gaussian mixture which it sends to an available fitting server. The fitting server then computes the sparse Gaussian mixture that approximates the input mixture best and sends the result back. The bricking server then compactly encodes the result into the output data structure which is reloaded later on to compute the next level.

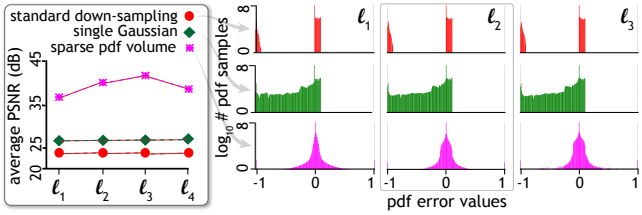

Fig. 6: **Pdf approximation error analysis** for the Visible Human data set. (Left) We plot the peak signal-to-noise ratio (PSNR) of the approximated pdf against the ground truth 256-bin histogram, averaged over all bricks. Sparse pdf volumes preserve the pdf information best (highest PSNR values), and across all resolution levels. (Right) We plot pdf approximation error distributions (histograms of error values, i.e., differences). Sparse pdf volumes have consistently smaller error values in all resolution levels, compared to the standard down-sampled and single Gaussian representation [40], respectively. The latter two exhibit large errors because they can only represent a single value and a single mode, respectively. Each error value is computed as the pointwise difference between ground truth pdf and approximated pdf in 4D.

8 Implementation for Large-Scale Volume Data

This section describes our implementation of the sparse pdf volume pre-computation step that scales to large volume data.

8.1 Scalable out-of-core pre-computation

We have implemented the pre-computation of sparse pdf volumes in a scalable distributed framework illustrated in Fig. 5. This framework is realized entirely out-of-core, so that data sets of arbitrary size can be processed. The framework is split up into two main distributed components: (1) a *bricking server*, and (2) one or several *fitting servers*.

8.1.1 Bricking server

The bricking server first reads the processing parameters and streams through the input volume. During this process, it subdivides the volume into bricks of pre-defined size, which in our case are 64^3 voxels. Each of these bricks is immediately written back to disk into level ℓ_0 of the output data structure. A parallel thread then goes through each set of $2 \times 2 \times 2$ bricks in ℓ_0, i.e., a neighborhood of 128^3 voxels. This neighborhood in turn corresponds to one 64^3 brick in level ℓ_1. The server then creates a corresponding fitting request and pushes it into a request queue. Each fitting request consists of the fitting parameters (σ of Gaussian atoms, target size of output Gaussian mixture), together with the list of 4D Gaussian components that are the kernel density estimate of the voxel data. Once a fitting server finishes processing a fitting request from the request queue, it sends back the resulting output 4D Gaussian mixture to the bricking server. The bricking server then writes it into the output sparse pdf volume on disk. Once level ℓ_1 is complete, the bricking server goes through each $2 \times 2 \times 2$ brick neighborhood in ℓ_1, which likewise now represents the Gaussian mixture of a 128^3 voxel neighborhood in ℓ_1, and thus a 64^3 brick in level ℓ_2. It then creates the corresponding new fitting request. This process is repeated in an identical manner for all remaining resolution levels.

8.1.2 Fitting server

The second component of our framework are the fitting servers, which fit a sparser Gaussian mixture to an input Gaussian mixture using the parallel pursuit described in Sec. 5.4. Each fitting server connects to the bricking server via a TCP/IP link. Once connected, the bricking server assigns a thread to the new fitting server and sends a fitting request from the request queue. After the transmission, the corresponding thread will pause and wait for the result to be returned. The fitting servers take the fitting request, i.e., an input Gaussian mixture, and process it in-core. The input Gaussian mixture is uploaded to the GPU and processed via the CUDA part of our framework (Sec. 8.2). Once the fitting is completed, the result is sent back to the bricking server where the responsible worker thread wakes up and receives the sparse Gaussian mixture data. This data is integrated into the output sparse pdf volume. Should the fitting server not return the result within a pre-defined timeout, or should the TCP/IP link be terminated, then the worker thread also wakes up and pushes the corresponding fitting request back to the queue so that another fitting server can take over.

An almost arbitrary number of fitting servers can connect to one bricking server. Due to the use of simple TCP/IP links, the fitting servers can be located anywhere as long as they can reach the bricking server via the network. These connections can be established dynamically during the run time of the system, i.e., a new fitting server can connect at any time, and existing servers can disconnect or fail without terminating the overall computation.

8.2 Parallel pursuit on the GPU

In order to reduce pre-computation times, we use a CUDA implementation of the parallel Matching Pursuit that is executed in the fitting servers. Other GPU and compute languages such as OpenCL could also be used without much difference. During the whole pursuit process, we maintain two Gaussian mixtures: (1) the residual, and (2) the detected modes in each iteration. Since our Gaussian mixtures are sparse in the whole 4D domain, and in order to facilitate fast neighborhood searches, we use a linked list-based hash table [32] to store each Gaussian mixture. This hash table, with a hashing function based on the work of Teschner et al. [35], allows us to quickly insert multiple Gaussian components using atomic operations on the GPU.

We first copy the input Gaussian mixture into the *residual hash table* by simply assigning a single thread for each component and performing parallel inserts into the residual hash table. Then, in each iteration of the parallel pursuit, we deploy as many threads as there are parallel mode searches to carry out. Each thread performs mode finding using fixed-point iteration [3], starting from the 4D position of the residual Gaussian component assigned to it. Once the modes have been detected, each thread inserts its mode into the *modes hash table*. We then perform a parallel search for overlaps using one thread per entry in the modes hash table. The maximum non-overlapping modes are then inserted into the residual hash table, and the process is repeated until the error threshold is met or the target number of output Gaussian components is reached. Once the pursuit process is finished, the fitting server copies the result from the GPU to the CPU, and sends the obtained Gaussian mixture to the bricking server.

9 Results and Comparisons

This section provides qualitative comparisons and quantitative results. We have used the Shepp-Logan phantom (512^3), the Visible Human ($512^2 \times 1884$), and a rat brain blood vessel (microvasculature) data set (1024^3) acquired using knife-edge scanning microscopy [29]. In this paper, all data sets use 8-bit voxels. However, processing 12- or 16-bit data is straightforward, since we operate in a continuous 4D domain.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2014.2346324, IEEE Transactions on Visualization and Computer Graphics

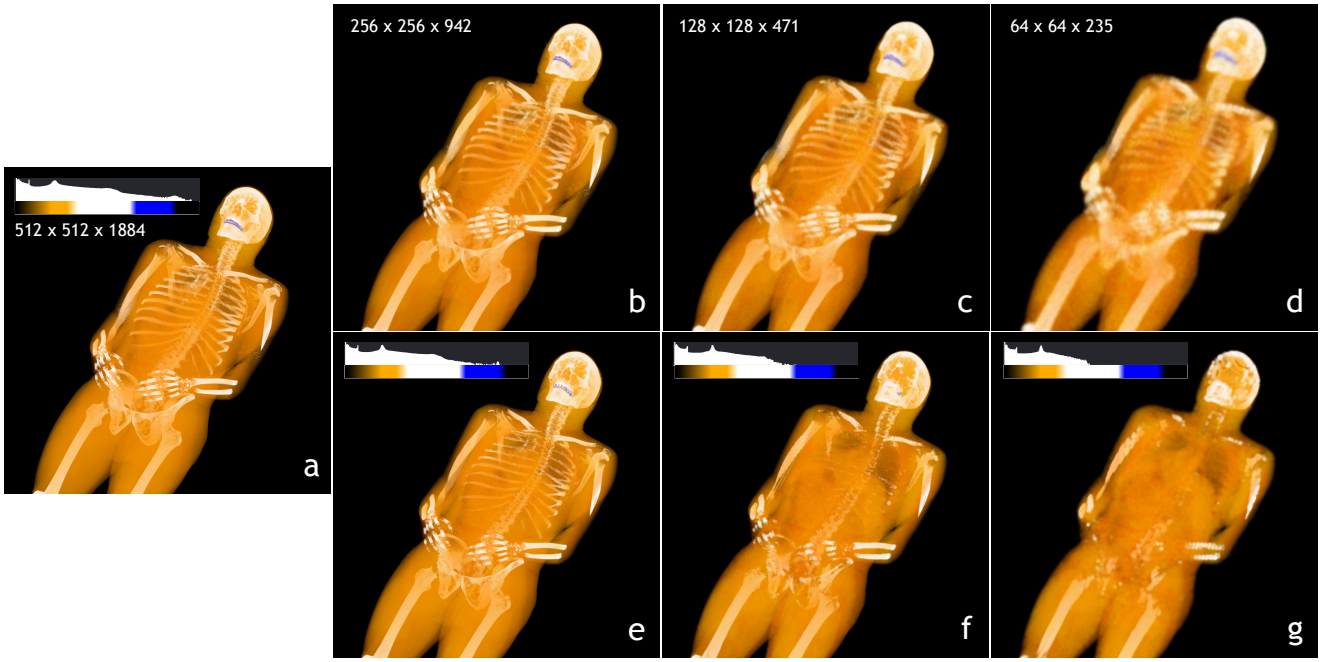

Fig. 7: **Visible Human**. (a) Volume rendering level ℓ_0 ($512^2 \times 1884$). Comparison of volume rendering coarser levels (b, e) ℓ_1 ($256^2 \times 942$), (c, f) ℓ_2 ($128^2 \times 471$), and (d, g) ℓ_3 ($64^2 \times 235$). (Top row) Sparse pdf volumes. (Bottom row) Standard pre-filtering and down-sampling. The logarithmic histograms of the standard data representations are shown on the top along with the transfer function. The standard representation loses more and more of the bones and teeth toward the coarser levels due to low-pass filtering (observe the changing histograms). Sparse pdf volumes are able to maintain much better consistency with the original rendering, keeping the important features of the bones and teeth intact.

9.1 Consistent multi-resolution volume rendering

The pdf approximation error analysis in Fig. 6 shows that sparse pdf volumes preserve the voxel neighborhood pdfs in each resolution level much better than standard low-pass filtering and down-sampling, as well as the single Gaussian approximation [40], respectively. This is the key to consistent volume rendering results even for coarse resolutions. The ground truth for our error analysis is the full 256-bin histogram (smoothed using σ_r in Eq. 6), since this is the function that we want to preserve. PSNR values are computed against this histogram.

Fig. 1 (b) shows that iteratively low-pass filtering and down-sampling the Shepp-Logan phantom introduces new values in the data which leads to inconsistency artifacts in multi-resolution volume rendering. Fig. 1 (c) shows that using a single Gaussian to represent voxel neighborhood pdfs is not sufficient to capture the multi-modal nature of pdfs in lower resolutions, which also results in similar artifacts. In contrast, sparse pdf volumes are able to accurately encode pdfs leading to (d) more consistent multi-resolution volume renderings that are almost equivalent to (e) the ground truth using a full histogram with 256 bins. Similarly, Figs. 7 (e,f,g) show that the coarser resolutions of the visible human increasingly lose important values such as the bones and teeth due to spatial pre-filtering and down-sampling, leading to inconsistent results across levels. In comparison, sparse pdf volumes are able to retain the information of the bones and teeth, leading to reduced inconsistency artifacts, as shown in the smooth transitions from level to level in Figs. 7 (b,c,d). Even small details as in the blood vessel data set depicted in Fig. 8 are consistently retained in the multi-resolution volume rendering using sparse pdf volumes (Figs. 8 a,d,e). This is in contrast to standard pre-filtering and down-sampling, where details dramatically disappear in the coarser resolution levels (Figs. 8 c,f,g).

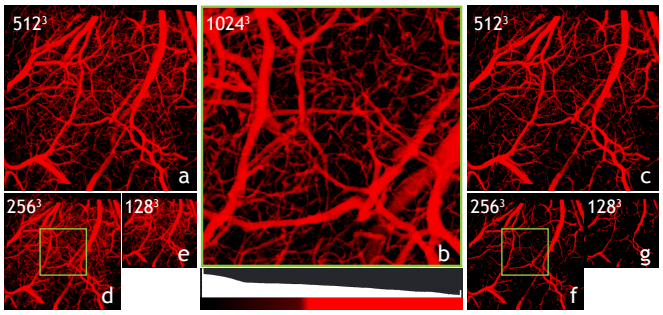

Fig. 8: **Blood Vessels.** (Left) Consistent multi-resolution volume rendering using sparse pdf volumes preserves details in the data for levels (a) ℓ_1, (d) ℓ_2, (e) ℓ_3 in contrast to (Right) standard pre-filtering and down-sampling for the same levels (c) ℓ_1, (f) ℓ_2, (g) ℓ_3. The boxes in (d) and (f) correspond to the zoom-in into level ℓ_0 (1024^3) in (b).

9.2 Performance and scalability

Pre-computation scalability. Table 1 gives pre-computation times for the data sets that we have used. We have used 16 fitting server processes, each with a 3.07 GHz Intel Xeon CPU and NVIDIA Tesla M2070Q (Fermi) GPU. On a single node, these times would roughly be larger by a factor of 16. Very importantly, despite the quadratic complexity of Matching Pursuit in general, our pre-computation times scale linearly with the data size, since we are subdividing the data into bricks and perform the pursuit on each brick independently. On average, a fitting server takes 5 minutes to process each 64^3 brick.

Our pre-computation step also scales well in terms of memory requirements with respect to the dimensionality of the domain (4D), since we do not need to re-sample the pdf information for the fitting process. This is in contrast to our earlier work [17]. Moreover, our pre-computation method would also effortlessly scale to 12- or 16-bit data, since mode finding always operates in a continuous 4D domain.

Classification performance. Using the optimized classification scheme described in Sec. 7.2, we achieve high classification performance at run time, as illustrated in Table 2 (timings measured on NVIDIA Geforce Titan Black (Kepler) GPU). We execute the classification step only when the transfer function changes, and also classify only the currently visible bricks. In order to do this, we have integrated volume classification and rendering with sparse pdf volumes into an existing large-scale out-of-core ray-caster [16]. Apart from the modified classification of bricks, the volume renderer did not need to be modified at all. Based on the classification approach described in Sec. 7.2, the volume renderer performs *pre-classified* volume render-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2014.2346324, IEEE Transactions on Visualization and Computer Graphics

Table 1: **Pre-computation times** using parallel pursuit (with 16 parallel fitting server processes) and the average time for one 64^3 brick. Note that # bricks is given for level ℓ_0, but fitting starts at level ℓ_1.

Data set	Resolution	# Bricks	1 Brick	Total
D1: Shepp-Logan	512^3	512	5 min	30 min
D2: Visible Human	$512^2 \times 1884$	1920	4.7 min	84 min
D3: Blood Vessels	1024^3	4096	5 min	188 min

Table 2: **Classification times.** We report average times per 64^3 brick (*total/brick*), consisting of the time for computing Eq. 28 (c_i*'s/brick*) and Eq. 29 ($\star$*/brick*). We also state the total time for classifying all bricks of resolution level ℓ_1 (D1: 256^3; D2: $256^2 \times 942$; D3: 512^3).

Data set	total ℓ_1	total / brick	c_i's / brick	$\star$ / brick
D1: Shepp-Logan	130 ms	2.03 ms	0.24 ms	1.79 ms
D2: Visible Human	526 ms	2.19 ms	0.37 ms	1.82 ms
D3: Blood Vessels	1,106 ms	2.16 ms	0.36 ms	1.80 ms

Table 3: **Storage requirements comparison.** The total storage requirements of sparse pdf volumes (*ours*) are comparable to standard representations. Column (μ,σ) is [40]; *histogram* stores 256 bins.

Data set	octree	(μ,σ)	ours	histogram
D1: Shepp-Logan	175 MB	219 MB	241 MB	11,361 MB
D2: Visible Human	659 MB	826 MB	909 MB	43,257 MB
D3: Blood Vessels	1,404 MB	1,755 MB	1,930 MB	91,044 MB

ing [10]. That is, the ray-caster fetches pre-classified RGBA values from the volume bricks, which are stored in an RGBA cache texture. An important consequence is that volume rendering frame rates are independent of the sparse pdf volume method. Rendering performance is exactly the same as for standard pre-classified volume rendering.

Storage requirements. Table 3 summarizes the storage required for each of our test data sets in comparison with standard volume representations. For a fair comparison, we use a brick size of 64^3 and four ghost (boundary) voxels in each spatial dimension throughout. We compare with an *octree* with 8 bits per voxel, single Gaussian (μ,σ) [40] with 24 bits per voxel (8-bit quantized μ, 16-bit float σ), and the full histogram representation with 256 bins and a 16-bit float per bin. Note that these values include the fixed storage requirement of level ℓ_0 (8 bits per voxel). The coarsest levels for D1, D2, and D3 that we have computed are levels ℓ_3 (D1), ℓ_4 (D2), and ℓ_4 (D3), respectively. Coarser resolution levels would already be smaller than a single brick for the entire volume. All sparse pdf volume representations used in this paper use 64^3 coefficients in each 64^3 voxel brick, i.e., on average (over the brick) there is a single coefficient per each voxel. This fact together with Table 3 shows that the sparse pdf volume representation is able to find a very good balance between storage overhead and the quality of the encoded pdf information. In all results, we have used $\sigma_s = 0.6$ and $\sigma_r = 4/256$ for the dictionary Gaussian h_μ.

Volume compression. We have not performed a direct comparison with volume compression methods, since we believe that our goals are quite different. It would, however, be possible to use a general compression method on top of the sparse pdf volume representation, i.e., performing an additional compression step after the sparse pdf volume has been computed. This, however, would require de-compressing before volume classification. In contrast, an important property of sparse pdf volumes is that classification is performed *directly* on the sparse representation, i.e., without any explicit "de-compression" step.

10 Limitations and Future Work

Pre-classification. Our fast classification method (Sec. 7.2) is limited to pre-classified volume rendering. While our general approach (Sec. 7.1) in principle also supports post-classification, the required spatial 3D convolution (Eq. 23) could pose a performance bottleneck.

Shading. While we have not described volume shading in this work, gradients for shading can be estimated via one of two ways, e.g., using central differences: (1) Estimating gradients on the alpha channel of the classified RGBA voxels, i.e., estimating gradients *after* applying the transfer function [8]; or (2) Computing the expected value of the volume intensity itself and estimating the gradient there.

Data properties. Our method may break down for data without spatial coherence, e.g., random volumes. We are currently exploring additional strategies to adapt to special cases in the data and allow a hybrid mixture of pdf representations in the same data structure, which could then better adapt to data properties and user requirements.

Other grid types and mixed-resolution rendering. While our main goal were volume data on regular grids, our representation could be extended to other grid types. The major requirements would be to define the voxel *footprint* in the multi-resolution hierarchy, and to be able to gather samples in the corresponding neighborhood. In addition to other regularly sampled grids, such as BCC grids, computing a sparse pdf volume for irregular or adaptive grids, e.g., AMR data, could be particularly interesting. We are also planning to explore the reduction of *transition artifacts* in mixed-resolution volume rendering.

11 Conclusions

Our sparse pdf volume representation for large-scale volume data compactly and accurately encodes the hierarchical pdf information of large multi-resolution volumes, which facilitates consistent multi-resolution volume rendering. In contrast to standard volume compression methods, our goal is not to reduce the memory size of the volume itself, but to encode much more information without requiring an impractical amount of additional storage, as well as avoiding explicit decompression for volume rendering. Our representation supports fast direct classification using the sparse encoding. We have shown that the representation of pdfs as a mixture of 4D Gaussians, and the corresponding parallel simplification, make pre-processing fast enough to achieve practical pre-computation times for large volume data.

Nevertheless, the pre-computation times are still the main bottleneck of our method and would be important to reduce in future work. However, it is a crucial property of our method that once a sparse pdf volume has been pre-computed, it does not require much more storage than standard methods that encode much less information, and facilitates real-time volume rendering with interactive transfer function changes with minimal changes to existing volume renderers.

Acknowledgments

This work was partially supported by King Abdullah University of Science and Technology (KAUST). We would also like to thank Thomas Theußl and Johanna Beyer; and John Keyser for the data set in Fig. 8.

References

[1] E. W. Bethel, H. Childs, and C. Hansen. *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*. CRC Press/Francis–Taylor Group, 2012.

[2] J. Beyer, M. Hadwiger, T. Möller, and L. Fritz. Smooth mixed-resolution GPU volume rendering. In *IEEE/Eurographics Symposium on Volume and Point-Based Graphics*, pages 163–170, 2008.

[3] M. A. Carreira-Perpiñán. Mode-finding for mixtures of Gaussian distributions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(11):1318–1323, 2000.

[4] W. Cheney and W. Light. *A Course in Approximation Theory*. American Mathematical Society, 2009.

[5] H. Childs, M. Duchaineau, and K.-L. Ma. A scalable, hybrid scheme for volume rendering massive data sets. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 153–161, 2006.

[6] D. Comaniciu and P. Meer. Mean shift: a robust approach toward feature space analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(5):603–619, 2002.

[7] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. GigaVoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 15–22, 2009.

[8] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. In *Proceedings of ACM SIGGRAPH 1988*, pages 65–74, 1988.

[9] M. Elad. *Sparse and Redundant Representations: From Theory to Applications in Signal and Image Processing*. Springer, 2010.

[10] K. Engel, M. Hadwiger, J. Kniss, C. Rezk-salama, and D. Weiskopf. *Real-Time Volume Graphics*. A. K. Peters, Ltd., 2006.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2014.2346324, IEEE Transactions on Visualization and Computer Graphics

[11] T. Fogal, A. Schiewe, and J. Krüger. An analysis of scalable GPU-based ray-guided volume rendering. In *IEEE Symposium on Large Data Analysis and Visualization*, pages 43–51, 2013.
[12] T. Gagie. Compressing probability distributions. *Information Processing Letters*, 97(4):133–137, 2006.
[13] E. Gobbetti, J. Iglesias Guitián, and F. Marton. COVRA: A compression-domain output-sensitive volume rendering architecture based on a sparse representation of voxel blocks. *Computer Graphics Forum*, 31(34):1315–1324, 2012.
[14] E. Gobbetti, F. Marton, and J. Iglesias Guitián. A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer*, 24(7):787–806, 2008.
[15] N. Greene and P. Heckbert. Creating raster omnimax images from multiple perspective views using the elliptical weighted average filter. *IEEE Computer Graphics and Applications*, 6(6):21–27, 1986.
[16] M. Hadwiger, J. Beyer, W.-K. Jeong, and H. Pfister. Interactive volume exploration of petascale microscopy data streams using a visualization-driven virtual memory approach. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2285–2294, 2012.
[17] M. Hadwiger, R. Sicat, J. Beyer, J. Krüger, and T. Möller. Sparse PDF maps for non-linear multi-resolution image operations. *ACM Transactions on Graphics*, 31(6):133:1–133:12, 2012.
[18] M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with CUDA. In H. Nguyen, editor, *GPU Gems 3*. Addison-Wesley Professional, 2007.
[19] P. Heckbert. Fundamentals of texture mapping and image warping. Master's thesis, U.C. Berkeley, 1989.
[20] E. Lamar, B. Hamann, and K. I. Joy. Multiresolution techniques for interactive texture-based volume visualization. In *Proceedings of IEEE Visualization*, pages 355–362, 1999.
[21] M. Levoy and R. Whitaker. Gaze-directed volume rendering. In *Proceedings of Symposium on Interactive 3D Graphics*, pages 217–223, 1990.
[22] X. Li and H.-W. Shen. Time-critical multiresolution volume rendering using 3d texture mapping hardware. In *Proceedings of the 2002 IEEE Symposium on Volume Visualization and Graphics*, pages 29–36, 2002.
[23] P. Lindstrom and M. Isenburg. Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1245–50, 2006.
[24] S. Liu, J. A. Levine, P.-T. Bremer, and V. Pascucci. Gaussian mixture model based volume visualization. In *IEEE Symposium on Large Data Analysis and Visualization*, pages 73–77, 2012.
[25] P. Ljung. Adaptive sampling in single pass, GPU-based raycasting of multiresolution volumes. In *Volume Graphics*, pages 39–46, 2006.
[26] K.-L. Ma, C. Wang, H. Yu, K. Moreland, J. Huang, and R. Ross. Next-generation visualization technologies: Enabling discoveries at extreme scale. In *SciDAC Review*, pages 12–21, 2009.
[27] S. Mallat. *A Wavelet Tour of Signal Processing: The Sparse Way*. Academic Press, 3rd edition, 2008.
[28] S. Mallat and Z. Zhang. Matching pursuits with time-frequency dictionaries. *IEEE Transactions on Signal Processing*, 41(12):3397–3415, 1993.
[29] D. Mayerich, J. Kwon, C. Sung, L. C. Abbott, J. Keyser, and Y. Choe. Fast macro-scale transmission imaging of microvascular networks using KESM. *Biomedical Optics Express*, 2:2888–2896, 2011.
[30] A. V. Oppenheim, R. W. Schafer, and J. R. Buck. *Discrete-time Signal Processing*. Prentice-Hall, 2nd edition, 1999.
[31] F. Reichl, M. Treib, and R. Westermann. Visualization of big SPH simulations via compressed octree grids. In *IEEE Big Data*, pages 71–78, 2013.
[32] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.
[33] B. W. Silverman. *Density Estimation for Statistics and Data Analysis*. Chapman and Hall/CRC, 1986.
[34] S. K. Suter, M. Makhynia, and R. Pajarola. TAMRESH: Tensor approximation multiresolution hierarchy for interactive volume visualization. *Computer Graphics Forum*, 32(8):151–160, 2013.
[35] M. Teschner, B. Heidelberger, M. Müller, D. Pomerantes, and M. Gross. Optimized spatial hashing for collision detection of deformable objects. In *Proceedings of Vision, Modeling, and Visualization*, pages 47–54, 2003.
[36] D. Thompson, J. A. Levine, J. C. Bennett, P.-T. Bremer, A. Gyulassy, V. Pascucci, and P. P. Pébay. Analysis of large-scale scalar data using hixels. In *IEEE Symposium on Large Data Analysis and Visualization*, pages 23–30, 2011.
[37] M. Weiler, R. Westermann, C. Hansen, K. Zimmermann, and T. Ertl. Level-of-detail volume rendering via 3D textures. In *IEEE Symposium on Volume Visualization*, pages 7–13, 2000.
[38] R. Westermann. A multiresolution framework for volume rendering. In *Proceedings of Symposium on Volume Visualization*, pages 51–58, 1994.
[39] X. Xu, E. Sakhaee, and A. Entezari. Volumetric data reduction in a compressed sensing framework. *Computer Graphics Forum*, 33(3):111–120, 2014.
[40] H. Younesy, T. Möller, and H. Carr. Improving the quality of multi-resolution volume rendering. In *Eurographics/IEEE-VGTC Symposium on Visualization (EuroVis)*, pages 251–258, 2006.

A Derivations Involving Weighted Gaussians

This appendix summarizes the most important basic computations involving weighted Gaussians that are used in the main part of the paper.

A.1 Convolution of weighted Gaussians

We define a weighted Gaussian as $c\,G_\sigma(x;\mu)$, with $G_\sigma(x;\mu)$ as defined by Eq. 16 (note that our G_σ are not normalized). The convolution of two weighted Gaussians results again in a weighted Gaussian:

$$c_n\,G_{\sigma_n}(x;\mu_n) = (c_0\,G_{\sigma_0}(x;\mu_0)) \star (c_1\,G_{\sigma_1}(x;\mu_1))\,, \tag{30}$$

with weights c_n, standard deviations σ_n, and positions μ_n, where:

$$\mu_n = \mu_0 + \mu_1, \tag{31}$$

$$\sigma_n = \sqrt{\sigma_0^2 + \sigma_1^2}, \tag{32}$$

$$c_n = \frac{\sqrt{2\pi}\sigma_0\sigma_1}{\sigma_n} c_0 c_1. \tag{33}$$

We note that for two Gaussian pdfs ($c_0 = \frac{1}{\sqrt{2\pi}\sigma_0}$, $c_1 = \frac{1}{\sqrt{2\pi}\sigma_1}$), Eq. 30 results again in a Gaussian pdf, because then Eq. 33 gives $c_n = \frac{1}{\sqrt{2\pi}\sigma_n}$.

A.2 Low-pass filtering $g(x)$

We compute the low-pass filtered $g(x)$ from our definition of $g(x)$ as a Gaussian mixture (Eq. 18) via convolution with a Gaussian low-pass filter kernel $w(x) := c_w G_{\sigma_w}(x)$ with $c_w = \frac{1}{\sqrt{2\pi}\sigma_w}$ so that $\int_{\mathbb{R}} w(x)\,\mathrm{d}x = 1$:

$$(g \star w)(x) = c_s \sum_{i=0}^{k-1} c_i\, G_{\sigma_{\text{filt}}}(x;\mu_i), \tag{34}$$

which again is a Gaussian mixture, with $\sigma_{\text{filt}} = \sqrt{\sigma^2 + \sigma_w^2}$ (cf. Eq. 32). For a down-sampling factor of 2, we desire $\sigma_{\text{filt}} := 2\sigma$. Therefore, we use $\sigma_w := \sqrt{3}\sigma$. Convolution with this filter therefore scales each c_i by a factor $c_s = 1/2$ (cf. Eq. 33). We therefore get:

$$(g \star w)(x) = \frac{1}{2} \sum_{i=0}^{k-1} c_i\, G_{2\sigma}(x;\mu_i). \tag{35}$$

A.3 All inner products of $g(x)$ as a convolution

Similarly, the convolution in Eq. 14 can be computed from our definition of $g(x)$ as a Gaussian mixture (Eq. 18) convolved with the dictionary Gaussian $h(x) := c_h G_{\sigma_h}(x)$ with $c_h = \frac{1}{\sqrt{\sqrt{\pi}\sigma_h}}$ so that $\|h(x)\|_2 = 1$:

$$\max_x (g \star h)(x) = \max_x c_p \sum_{i=0}^{k-1} c_i\, G_{\sigma_{\text{proj}}}(x;\mu_i), \tag{36}$$

which again is a Gaussian mixture, with $\sigma_{\text{proj}} = \sqrt{\sigma^2 + \sigma_h^2}$, and each c_i scaled by the factor $c_p = \frac{\sqrt{2}\pi^{\frac{1}{4}}\sigma\sqrt{\sigma_h}}{\sigma_{\text{proj}}}$ (cf. Eq. 33). If we use $\sigma_h := \sigma$, i.e., we project into a dictionary of Gaussians of the same width as the mixture Gaussians in Eq. 18, we get $\sigma_{\text{proj}} = \sqrt{2}\sigma$, and thus:

$$\max_x (g \star h)(x) = \max_x \sqrt{\sigma}\pi^{\frac{1}{4}} \sum_{i=0}^{k-1} c_i\, G_{\sqrt{2}\sigma}(x;\mu_i). \tag{37}$$