To appear in IEEE Transactions on Visualization and Computer Graphics (IEEE Vis 2020).

The Mixture Graph—A Data Structure for Compressing, Rendering, and Querying Segmentation Histograms

Khaled Al-Thelaya Marco Agus Jens Schneider (*IEEE Member*)

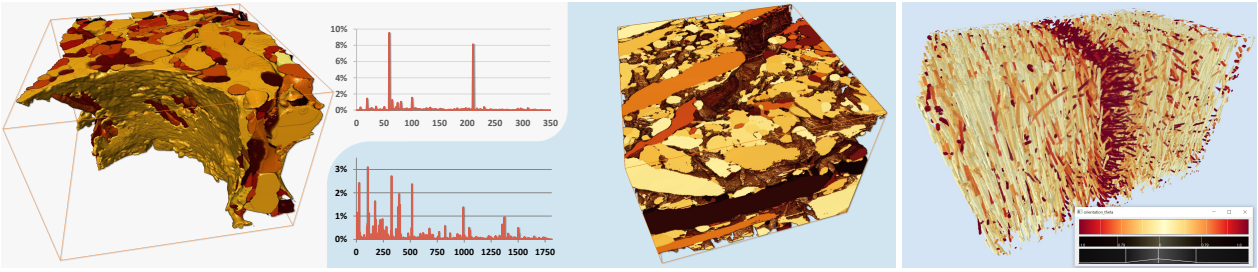

Fig. 1: The *Mixture Graph* allows us to interactively render and query segmented volumes while maintaining pre-filtered anti-aliasing across multiple scales. Segmented volumes store nominal data and therefore pose challenges if traditional rendering is attempted. **Left to right:** Hippocampus ($1178 \times 1125 \times 789$, 353 segments), Neocortex ($2048 \times 2048 \times 300$, 1,182 segments), fiber-reinforced Polymer ($614 \times 961 \times 600$, 15,917 segments).

Abstract—In this paper, we present a novel data structure, called the *Mixture Graph*. This data structure allows us to compress, render, and query segmentation histograms. Such histograms arise when building a mipmap of a volume containing segmentation IDs. Each voxel in the histogram mipmap contains a convex combination (*mixture*) of segmentation IDs. Each mixture represents the distribution of IDs in the respective voxel's children. Our method factorizes these mixtures into a series of linear interpolations between exactly two segmentation IDs. The result is represented as a directed acyclic graph (DAG) whose nodes are topologically ordered. Pruning replicate nodes in the tree followed by compression allows us to store the resulting data structure efficiently. During rendering, transfer functions are propagated from sources (leafs) through the DAG to allow for efficient, pre-filtered rendering at interactive frame rates. Assembly of histogram contributions across the footprint of a given volume allows us to efficiently query partial histograms, achieving up to $178\times$ speed-up over naïve parallelized range queries. Additionally, we apply the Mixture Graph to compute correctly pre-filtered volume lighting and to interactively explore segments based on shape, geometry, and orientation using multi-dimensional transfer functions.

Index Terms—Segmented Volumes, Data Structures, Sparse Data

1 Introduction

Segmentations of volume data have traditionally been used in a medical context to separate different semantic objects in the data. Typically, one or more segment IDs are computed per voxel that represent the probability of the voxel belonging to the specific segment. Such segmentations have then been used either to modulate the transfer function or to extract 3D representations for each semantic object. Since then, segmented volumes have also become popular in other disciplines, such as engineering, e.g., in the form of topological segmentation of flow fields. In all these examples, however, segmentations have been primarily treated as annotations to the original data.

This is changing due to the extensive use of segmentations in fields such as neuroscience, connectomics (see also Fig. 1, left & middle), computational neurology and material science (see also Fig. 1, right). Here, raw data is typically imaged using an electron microscope (EM). This data, however, primarily serves as a means to compute the segmentation. Once the raw volume has been successfully partitioned into semantic objects, the raw data is used less and less. Reasons for this procedure include that the raw EM data is noisy and often lacks a direct optical interpretation. Some of the data in the OpenConnectome Project [32], for instance, is imaged using a high-resolution scanning (non-transmissive) EM [24]. Such data does represent neither opacity nor color. Therefore, this traditional input for volume rendering can be assigned using transfer functions [27] prior to rendering. Alternatively, 3D surface representations of semantic objects may be used. Still, volume rendering of the raw data [23] is highly regarded as a valuable tool in order to obtain, author, and annotate [4, 5], or validate [1] the resulting segmentation. However, as segmentation volumes are now a *primary* modality rather than a derived quantity, interactive, direct rendering of segmentations gains importance [6].

Such segmentation volumes store *nominal* data, posing challenges to traditional rendering attempts such as: data cannot be interpolated or pre-filtered before assigning optical properties to voxels. This implies that, traditionally, mipmaps [56] need to be recomputed from scratch whenever a change in the transfer function alters these voxel properties. Adding to the challenges, volumes generated in connectomics are among the largest volumetric data sets currently available, making pre-filtered anti-aliasing highly desireable. Moreover, the nominal character of the data also makes reducing data size while maintaining direct renderability difficult, ruling out virtually all existing lossy compression methods.

Contributions. In this paper, we address some of the most dire of the aforementioned needs. In particular, we present the Mixture Graph, a novel, graph-based data structure to represent hierarchical segmentation histograms. We show that these histograms can be understood as computational instruction for building a mipmap to support pre-filtered anti-aliasing *before* the optical properties at voxels are known. Compared to naïve use of such histograms, the Mixture Graph presents an efficient and compact alternative and allows us to propagate transfer function updates efficiently and in parallel through the graph. This is achieved by a symbolic factorization that breaks the aforementioned computational instruction into a sequence of linear interpola-

- *All authors are with the College of Science and Engineering (CSE), Hamad Bin Khalifa University (HBKU), Education City, Doha, Qatar.*
- *Jens Schneider is the corresponding author and can be reached at jeschneider@hbku.edu.qa.*

tions. These interpolations can be compressed using well-established, lossy compression techniques. This leads to a prune-and-recycle operation on the underlying graph that exploits sparsity in the histogram's range. The resulting data structure naturally supports pre-filtered rendering. We also describe how the Mixture Graph can be used to store quantized normals to provide simple pre-filtered shading. Finally, we present a footprint assembly algorithm to efficiently compute partial segment histograms across a given sub-volume.

2 Related Work

In connectomics and material sciences, segmentation volumes have ceased to be mere annotations to the semantic objects in the data. Instead, the segmentation has become a first-class modality to be analyzed and visualized. Reasons include that connectomics data is typically imaged with an electron microscope, an imaging modality prone to noise which does not necessarily result in data lending itself to interpretation in terms of optical absorption and opacity. For instance, unlike transmission EMs, the more commonly used scanning EMs measure the backscattering of the electron beam [42]. Still, direct volume rendering is used [29, 23], often in combination with other techniques, to proofread [17, 1] and analyze or explore [2, 5, 4, 21] automatically [25, 50, 39] or semi-automatically [22] generated segmentations.

As more and more automated segmentation algorithms become available, large and densely segmented data sets emerge. Unlike the quantized real values stored by their EM input, these data sets store nominal integer IDs. Rendering such modalities was addressed, e.g., by two-level volume rendering [18, 16], which combines multiple volume rendering techniques to highlight the various semantic objects in combination with the original input data. Other approaches seek to reconstruct a smooth surface for purposes of rendering [26].

While all this demonstrates the interest in segmented data, compression of *segmented volume data* gained relatively little attention. The reason is that, albeit, being the most popular and successful choice in the last decades, lossy compression methods [20, 33, 46] cannot be readily used to compress nominal integer data. On the other hand, lossless compression methods face challenges with respect to implementation and performance on parallel architectures [35]. This is due to their inherently sequential view of the data, and, while such methods exist [13, 54, 53, 47], they generally do offer neither the random access nor the bandwidth required to interactively render directly from the compressed representation. Aside from raw integer volumes [32], PNG-compressed RGB or RGBα slices storing the segment IDs in multiple 8-bit channels are still a de-facto format for exchanging columns of neuronal tissue [24, 7]. PNG's compression ratio for this type of data is generally significant (e.g., < 0.3 bits per pixel for the Hippocampus [7], Fig. 1 left), reflecting the sparse nature of segmentation data. However, such a compression does not address the generation of hierarchies from the data. Such hierarchies (mipmaps) are crucial in providing pre-filtered anti-aliasing for high resolution volumes. Furthermore, PNG stacks need to be decompressed prior to rendering, e.g., to synthesize views oblique to the slices. Unlike traditional medical segmentations, the number of segment IDs in connectomics and material sciences that individual binary segmentations have little to no practical relevance.

Traditional compression methods for *scalar volume data* focus on reducing the size of opacity values. Wavelets [10, 8], while resulting in good compression rates [31, 20, 33, 3] are usually not well suited for decoding on the GPU. The reason is that they derive their efficiency not only from the actual wavelet transform, but from the coding back-end of the transform coefficients. Arithmetic codes [38] or variations of embedded zero tree codes (e.g., SPIHT [41]) are traditionally used. These do not trivially support random parallel access. GPU-based wavelets are employed in the field of terrain rendering [49] and octrees [37] are used to compress volume data. All these methods, however, do not support lossless encoding of nominal data such as the segmentation volumes we are concerned with. While lossless coding may also be driven by a wavelet transform, such as the fully invertible LeGall integer basis [8]. To the best of our knowledge, however, such lossless transforms have not yet been applied to volume data.

Vector quantization [14] has been applied successfully to the lossy compression of volume data [34, 46, 12]. Vector quantization is similar to our method in that a palette or codebook is learned from the data. Each entry in the codebook stores a vector, whereas each vector in the input data is replaced by an index into the codebook. In this work, we utilize vector quantization to derive a nominal volume from a normal map to apply the Mixture Graph for pre-filtered shading. The related field of sparse coding and sparse dictionary learning [40] can be seen as a generalization of vector quantization: instead of referencing the codebook with a single index, a sparse weight vector is stored. Decoding consists of computing a linear combination of codebook entries. More recently, Wang et al. [51, 52] propose to apply sparse 3D Gaussian Mixture Models to handle massive scalar simulation volumes.

In this work, we consider the efficient, hierarchical storage of segmented volume data. Our method considers a mipmap of attributes, such as segment color, that would arise under application of a transfer function to the input segmentation. However, unlike a traditional mipmap that is built *after* a transfer function is applied, our data structure stores the *computations* necessary to arrive at a mipmap. These computations are factorized into simple linear interpolations that can be compressed efficiently and by lossy methods. Our data representation is essentially a paletted texture, in which the palette can be updated efficiently and in parallel. Unlike traditional paletted textures, however, our palette can store mixtures of multiple colors, somewhat similar to two-colored pixels [36]. Our method is most closely related to double sparsity and sparse coding [40], although the methodology differs substantially: We perform a greedy factorization of the *computations* necessary to compute a palette, whereas sparse coding is usually formulated as an (orthogonal) matching pursuit [30] optimization problem to represent data in a potentially overcomplete basis.

3 Algorithmic Overview

Our method first computes a normalized histogram mipmap (Sec. 3.1), in which each voxel stores a "mixture" of segment IDs. Mixtures are convex combinations of segment IDs reflecting the relative number of occurrences of each ID within each voxel of the mipmap. Clearly, such a histogram may have significant storage requirements and typically results in heavily unbalanced workloads during rendering.

In a second step, we factorize the histogram mipmap into a set of linear interpolations (Sec. 3.2). Made possible by embedding mixtures in $\mathbb{R}^\infty$, this step results in a directed, acyclic graph (DAG) representing both the histogram *and* the computations necessary to reconstruct or render the original segmented volume at different scales (Sec. 3.4).

We use a scalar quantization step to prune redundant nodes in the DAG. The result is then compressed at a fixed bitrate to facilitate fast, random access to the original histogram (Sec. 3.3).

Since the quantization step is lossy, we also store quantization errors for each quantization bin to estimate the reconstruction error in later stages (Sec. 3.5), such as running fast, approximate queries of partial histograms across any given sub-volume. This feature allows domain scientists to quickly count IDs in a volumetric range and assess their distribution (Sec. 3.6).

Notation. In this paper, we make heavy use of convex combinations that we call "mixtures". A mixture is described by a vector $\mathbf{m} \in \mathbb{R}^\infty$ with the following properties.

$$\|\mathbf{m}\|_1 = 1 \tag{1}$$

$$[\mathbf{m}]_n \geq 0 \; \forall n \tag{2}$$

$$\|\mathbf{m}\|_0 < \infty, \tag{3}$$

where we used the ℓ_0 pseudo-norm to count non-zero elements in $\mathbf{m}$ and used $[\cdot]_n$ to denote the n^{th} element in a vector. The scalar product $\langle \mathbf{m}, \mathbf{x} \rangle$ between a mixture vector $\mathbf{m}$ and a vector $\mathbf{x} \in \mathbb{R}^\infty$ storing data thus computes a convex combination (Eq. (1,2)) of a finite number of elements (Eq. (3)) in $\mathbf{x}$. We further define the set of all mixtures

$$\mathcal{M} := \{\, \mathbf{m} \in \mathbb{R}^\infty \;:\; \|\mathbf{m}\|_1 = 1 \;\wedge\; \|\mathbf{m}\|_0 < \infty \;\wedge\; [\mathbf{m}]_n \geq 0 \; \forall n \,\}.$$

Finally, we generally assume that the greatest position of a non-zero element in $\mathbf{m}$ is known and finite, denoted

$$\exists \widehat{k}(\mathbf{m}) \in \mathbb{N}_0 : [\mathbf{m}]_k = 0 \quad \forall k > \widehat{k}(\mathbf{m}).$$

3.1 Normalized Histogram Mipmap

Given a compact, discrete domain, $\mathcal{D} \subseteq \mathbb{N}_0^3$, and a volume of segment IDs, $\mathcal{S} : \mathcal{D} \to \mathbb{N}_0$, we construct a hierarchical segmentation histogram with $l_{\max}+1$ levels, $\mathcal{H} : \mathcal{D} \times \{0,\dots,l_{\max}\} \to \mathcal{M}$ as follows. Let $i,j,k \in \mathcal{D}$ denote a voxel position in 3D and $l \in \{0,\dots,l_{\max}\}$ denote a hierarchy level. Let $l = 0$ refer to the level with the finest resolution and let $\widehat{\mathbf{e}}_i$ denote the i^{th} unit vector over $\mathbb{R}^\infty$. We then compute

$$\begin{aligned}\mathcal{H}(i,j,k,0) &= \widehat{\mathbf{e}}_{\mathcal{S}(i,j,k)} \quad \forall i,j,k \in \mathcal{D} \\ \mathcal{H}(i,j,k,l) &= \frac{1}{N} \sum_{u=2i}^{2i+1} \sum_{v=2j}^{2j+1} \sum_{w=2k}^{2k+1} \mathcal{H}(u,v,w,l-1), \end{aligned} \tag{4}$$

where N is the number of voxels in the support of $\mathcal{H}(i,j,k,l)$ (typically 8, but potentially less than 8 at the borders). If the input data has non-power-of-two resolution, we round up the resolution of each subsequent level, adjusting summation limits and N in Eq. (4) accordingly. We continue computing additional levels in $\mathcal{H}$ in this way until we reach $l = l_{\max}$ with a resolution of 1^3 voxels.

Our assumption that the input volume $\mathcal{S}$ contains exactly one segment ID per voxel merely served the exposition of this section. If voxels in $\mathcal{S}$ already contain mixtures (i.e., $\mathcal{S} : \mathcal{D} \to \mathcal{M}$), we set $\mathcal{H}(\cdot,0) = \mathcal{S}$ and proceed as described above. In this paper, we will only discuss the traditional *average-of-eight* mipmap filter [56], but other low-pass filters can be used in the construction of $\mathcal{H}$, as long as the low-pass filter can be normalized to a mixture itself.

3.2 Factorization

High-resolution levels of $\mathcal{H}$ are very sparse for most real-world segmented volumes. This is particularly true for volumes in which only one segment ID is provided per voxel. In contrast, $\mathcal{H}(\cdot, l_{\max})$ is the dense, normalized histogram of all segment IDs in the volume. This poses challenges for processing and rendering such histograms, since the workload per voxel is highly inhomogeneous: In order to render a segmented volume with 1,024 IDs using an RGBα transfer function, only one fetch is sufficient for each voxel in $l = 0$, whereas higher levels require up to 1,024 fetches to compute the color of a single voxel. To balance this workload, we propose to factorize each mixture into a set of "simpler" mixtures. In this paper, we consider mixtures of the form $\lambda \in \mathcal{M} : \|\lambda\|_0 \leq 2$, that is, we restrict the factorization to either linear interpolation between two elements or identity of one element.

Such a factorization is always possible in the $\mathbb{R}^\infty$ embedding. We start by populating a mixture list Λ with N trivial mixtures, one for each input segment: $\Lambda = [\,\widehat{\mathbf{e}}_1,\dots,\widehat{\mathbf{e}}_N\,]$. We then examine one of the remaining mixtures $\mathbf{m}$ "over Λ" (that is, $[\mathbf{m}]_n \neq 0 \Leftrightarrow n \in \{1,\dots,N\}$) with $\|\mathbf{m}\|_0 > 2$. We pick two non-zero positions i,j (i.e., $[\mathbf{m}]_i \neq 0$ and $[\mathbf{m}]_j \neq 0$). After that, we compute a new mixture

$$\lambda_{N+1} := \frac{[\mathbf{m}]_i \widehat{\mathbf{e}}_i + [\mathbf{m}]_j \widehat{\mathbf{e}}_j}{[\mathbf{m}]_i + [\mathbf{m}]_j}, \tag{5}$$

which is appended to the mixture list

$$\Lambda \leftarrow\!\shortmid \Lambda \oplus \lambda_{N+1}. \tag{6}$$

Finally, we update $\mathbf{m}$,

$$\mathbf{m} \leftarrow\!\shortmid \mathbf{m} - [\mathbf{m}]_i \widehat{\mathbf{e}}_i - [\mathbf{m}]_j \widehat{\mathbf{e}}_j + \Big([\mathbf{m}]_i + [\mathbf{m}]_j\Big) \widehat{\mathbf{e}}_{N+1}. \tag{7}$$

This update removes one non-zero entry from $\mathbf{m}$ in total, and creates a linear interpolation λ_{N+1}. Here, we used the embedding of mixtures in $\mathbb{R}^\infty$ to add mixtures that, informally speaking, use positions "behind" those dimensions of $\mathbb{R}^\infty$ that previously carried information. We therefore use the embedding in $\mathbb{R}^\infty$ to the same effect described in Hilbert's Grand Hotel thought experiment [11], in which a fully booked hotel with infinitely many rooms can always accommodate a countable, potentially infinite number of new guests. Unlike the original thought experiment, in which guests move rooms to free up the room with the smallest index, we use the embedding to append mixtures with larger and larger $\widehat{k}(\mathbf{m})$ to a countable set of mixtures with finite norms which, in the limit, may span $\mathbb{R}^\infty$.

When repeated until $\|\mathbf{m}\|_0 = 1$, we obtain a series of linear interpolations λ_i with maximum non-zero positions

$$\widehat{k}(\lambda_i) < i, \tag{8}$$

which is a direct consequence of "appending" linear interpolations to previously used dimensions in $\mathbb{R}^\infty$. Figure 2 shows an example for the factorization of mixtures.

Input:	$\mathbf{m} = (\,0.1, 0.2, 0.3, 0.4, \dots\,)$
Mixture List:	$\Lambda = [\,\widehat{\mathbf{e}}_1, \widehat{\mathbf{e}}_2, \widehat{\mathbf{e}}_3, \widehat{\mathbf{e}}_4\,]$
Pick **1**,**2**:	$\mathbf{m} = (\,\mathbf{0.1}, \mathbf{0.2}, 0.3, 0.4, \dots\,)$
Insert new λ:	$\Lambda = [\,\widehat{\mathbf{e}}_1, \widehat{\mathbf{e}}_2, \widehat{\mathbf{e}}_3, \widehat{\mathbf{e}}_4, \lambda_5\,]$, $\lambda_5 = \frac{1}{3}(\,\mathbf{1}, \mathbf{2}, \dots\,)$
Update:	$\mathbf{m} = (\,\mathbf{0}, \mathbf{0}, 0.3, 0.4, \mathbf{0.3}, \dots\,)$
Pick **3**,**5**:	$\mathbf{m} = (\,0, 0, \mathbf{0.3}, 0.4, \mathbf{0.3}, \dots\,)$
Insert new λ:	$\Lambda = [\,\widehat{\mathbf{e}}_1, \widehat{\mathbf{e}}_2, \widehat{\mathbf{e}}_3, \widehat{\mathbf{e}}_4, \lambda_5, \lambda_6\,]$, $\lambda_6 = \frac{1}{2}(\,0, 0, \mathbf{1}, 0, \mathbf{1}, \dots\,)$
Update:	$\mathbf{m} = (\,0, 0, \mathbf{0}, 0.4, \mathbf{0}, \mathbf{0.6}, \dots\,)$
Pick **4**,**6**:	$\mathbf{m} = (\,0, 0, 0, \mathbf{0.4}, 0, \mathbf{0.6}, \dots\,)$
Insert new λ:	$\Lambda = [\,\widehat{\mathbf{e}}_1, \widehat{\mathbf{e}}_2, \widehat{\mathbf{e}}_3, \widehat{\mathbf{e}}_4, \lambda_5, \lambda_6, \lambda_7\,]$, $\lambda_7 = \frac{1}{5}(\,0, 0, 0, \mathbf{2}, 0, \mathbf{3}, \dots\,)$
Update:	$\mathbf{m} = (\,0, 0, 0, \mathbf{0}, 0, \mathbf{0}, \mathbf{1}, \dots\,)$
Result:	$\mathbf{m} = (\,0, 0, 0, 0, 0, 0, 1 \dots\,)$ $\Lambda = [\,\widehat{\mathbf{e}}_1, \widehat{\mathbf{e}}_2, \widehat{\mathbf{e}}_3, \widehat{\mathbf{e}}_4, \lambda_5, \lambda_6, \lambda_7\,]$.

Fig. 2: Factorization of a mixture $\mathbf{m}$, $\|\mathbf{m}\|_0 = 4$. In each step, two non-zero positions in $\mathbf{m}$ are selected, a new linear interpolation λ is added to the mixture list Λ, and $\mathbf{m}$ is updated (... denote trailing zeros).

As depicted in Fig. 3, such a series of linear interpolations can be represented as a binary tree with edge weights. Leaf nodes represent the input volume's segments and the root represents the final mixture $\mathbf{m}$. Each internal node represents a linear interpolation λ with exactly two children corresponding to the two non-zero entries i,j in λ. Edge weights are given by $[\,\lambda\,]_i$ and $[\,\lambda\,]_j$.

Carrying out the factorization for a set of mixtures while reusing identical nodes results in a directed acyclic graph (DAG) $\mathcal{G} = (N,E,W)$ as depicted in Figure 4. The node set N contains nodes with an in-degree of 0 (sources) representing the original segment IDs, internal nodes with an in-degree of 2, and sinks with an out-degree of

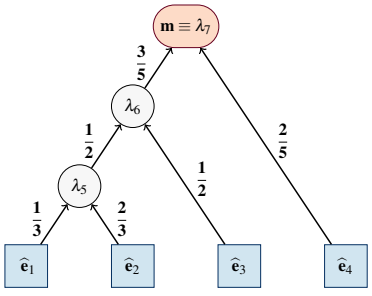

Fig. 3: Factorization tree of the example in Fig. 2. Leaf nodes (boxes) store unit vectors representing the input segment IDs, internal nodes (circles) perform linear interpolations between two children, and the root (rounded rectangle) corresponds to the original mixture $\mathbf{m}$.

0. The final mixture for each voxel is represented by either sources or sinks. We define the edge direction of $(i,j) \in E$ as "from i to j", with the notion of mixture i "contributes to" mixture j:

$$\exists (i,j) \in E : w_{ij} \neq 0 \in W \;\Leftrightarrow\; \exists \boldsymbol{\lambda}_\mathbf{j} \in \Lambda, i \in \mathbb{N}_0 : \left[\boldsymbol{\lambda}_\mathbf{j}\right]_i = w_{ij}. \quad (9)$$

Since each node except for sources has an in-degree of exactly two, we store the connectivity information for each node as incoming edges. For each edge (i,j), we call node i a *predecessor* of j and, conversely, j a *successor* of i.

3.3 Compression

From the previous section, it is intrinsically clear that the factorization into linear interpolations is not unique: in each step *any* two non-zero elements of $\mathbf{m}$ can be picked. Our compression method exploits this degree of freedom to generate as many redundant nodes as possible. For instance, Fig. 4 shows that $\boldsymbol{\lambda}_5$ can be re-used in the factorization of both $\mathbf{m}$ and $\mathbf{m}'$, since they both mix $\widehat{\mathbf{e}}_1$ and $\widehat{\mathbf{e}}_2$ in the same ratio $1:2$, albeit with different weights of 0.5 and 0.6.

Finding the factor with the highest re-usability is a hard problem. In the first step, we have N choose 2 possible picks i,j, where N is the number of segments in the input. Picking i,j removes this combination from subsequent picks. However, we add a new option to pick from ($N-1$ choose 2). To maximize overall re-use of nodes, we thus cannot process the factorization steps independently of one another. The full search space offers a total of P choices, with

$$P = \prod_{i=2}^{N} \binom{i}{2} = \frac{1}{2}\prod_{i=2}^{N} i(i-1) = \frac{1}{2N}\left(N!\right)^2. \quad (10)$$

The problem is further exacerbated by the sheer number of mixtures to be considered and the fact that only nodes with the same ratio between components i and j can be re-used. Our method therefore relies on a greedy algorithm to find a candidate pick i,j that has a good re-use probability. We consider two greedy strategies. The first strategy, which we call **max-occurrence**, picks the pair of indices i,j that most frequently corresponds to non-zero components of mixtures in $\mathcal{H}$. The rationale behind this strategy is that, even though we are ultimately interested in recycling nodes representing mixtures $\lambda = (1-w)\widehat{\mathbf{e}}_i + w\widehat{\mathbf{e}}_j$, frequent combinations of non-zero indices i,j are good candidates. The reason is that $w \in (0,1)$ can be quantized to increase re-use. Formally, we define a counting function ϕ_1,

$$\begin{aligned} \mathcal{C} &:= \{i,j \in \{1,\dots,N\} : i \neq j\} \\ \phi_1 :&\quad \mathcal{C} \to \mathbb{N}_0 \\ \phi_1(i,j) &:= \left|\left\{\mathbf{h} \in \mathcal{H} : [\mathbf{h}]_i \neq 0 \wedge [\mathbf{h}]_j \neq 0\right\}\right|, \end{aligned} \quad (11)$$

and compute

$$i,j = \underset{k,l \in \mathcal{C}}{\arg\max}\ \phi_1(k,l). \quad (12)$$

We call our second strategy **max-reduction**. It exploits that after factoring out a mixture λ in $\mathbf{m}$ with $\|\lambda\|_0 = 2$ and $\|\mathbf{m}\|_0 = N$, a mixture $\mathbf{m}'$ with $\|\mathbf{m}'\|_0 = N-1$ remains. Thus, picking i,j reduces our choices from N choose 2 to $N-1$ choose 2, a reduction by $N-1$ if $N > 2$ and by 2 if $N = 2$. Since this decrease of choices is directly equivalent to shrinking the search space, our strategy is to find a pair i,j minimizing the size of the remaining search space. Formally, we define a second counting function ϕ_2,

$$\begin{aligned} \phi_2 :&\quad \mathcal{C} \to \mathbb{N}_0 \\ \phi_2(i,j) &:= \sum_{\mathbf{h} \in \mathcal{H}} \begin{cases} 0 & \text{if } [\mathbf{h}]_i = 0 \vee [\mathbf{h}]_j = 0 \\ \|\mathbf{h}\|_0 - 1 & \text{else if } \|\mathbf{h}\|_0 > 2 \\ 2 & \text{else if } \|\mathbf{h}\|_0 = 2 \end{cases}, \end{aligned} \quad (13)$$

and compute

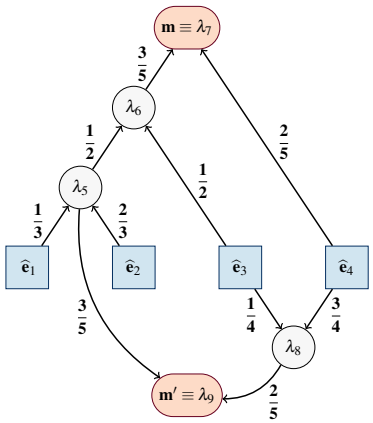

Fig. 4: Example directed acyclic graph (DAG) corresponding to the factorization of $\mathbf{m} = (\,0.1, 0.2, 0.3, 0.4, \dots\,)$ and $\mathbf{m}' = (\,0.2, 0.4, 0.1, 0.3, \dots\,)$. Mixture $\boldsymbol{\lambda}_5$ can be re-used, since the ratio between the first two components is the same ($0.1:0.2 \;\equiv\; 0.2:0.4$).

$$i,j = \underset{k,l \in \mathcal{C}}{\arg\max}\ \phi_2(k,l). \quad (14)$$

These two strategies are compared in the Results section. Once we have found the best pair i,j according to one of the strategies, we proceed by factoring all occurrences of i,j. We repeat this process until $\mathcal{H}$ is fully factorized, i.e., $\forall \mathbf{h} \in \mathcal{H} : \|\mathbf{h}\|_0 = 1$. To improve re-use of the nodes thus generated, we use scalar quantization on the ratios of the linear interpolations. Note in this context that linear interpolation ratios cannot be 0 or 1, since this would exactly replicate a previous mixture. Consequently, our scalar quantizer partitions the open range $(0,1)$ into 2^b bins, where b is the bitrate of the quantization. Each bin is represented by a single floating point codeword that best represents the weights in the bin. In addition, we store an empirical standard deviation σ for each bin to facilitate estimating the error of the quantization throughout the entire reconstruction process. Since the diversity of interpolation weights is greatly reduced by the quantization step, more nodes in the Mixture Graph become identical and can be re-used.

The final output of the compression step is a bitstream storing the compressed Mixture Graph. Sources are not stored explicitly since they correspond to input segmentation IDs. The number of sources, however, is stored in the bitstream's header, along with the volume's dimensions, a flag indicating whether the input volume contained a fractional segmentation (i.e., input segments are already mixtures), the bitrate of the scalar quantizer, etc. Table 1 provides an overview of the output format. For each node, we store a triple of values consisting of two node IDs and one index into the scalar quantizer's

Table 1: Overview of the output bitstream.

ID	description	# bits	range/comments
(1)	# sources	32bits	$[0,\dots,2^{32}-1]$
(2)	fractional input ?	1bit	$\exists \mathbf{h} \in \mathcal{H}(\cdot,0) :\ \|\mathbf{h}\|_0 > 1$?
(3)	bits per node	6bits	$[1,\dots,64]$
(4)	$\sigma_{\max}$	32bits	max. σ of quantization
(5)	3D resolution	3×32bits	resolution of $\mathcal{H}(\cdot,0)$
(6)	quantization bitrate	4bits	$[1,\dots,16]$
(7)	# internal nodes	var bits	# bits defined at (3)
For each level in $\mathcal{H}$			
(8)	# voxel bits	var bits	# bits defined at (3)
(9)	$i_{\min}(l)$	var bits	# bits defined at (3)
Bulk data			
(A)	one mixture ID per voxel		# bits defined at (8)
(B)	description of mixtures		# bits = 2×(3) + (6)
(C)	scalar quantizer codebook		20bits for each weight and σ

codebook of interpolation weights. This is done at a fixed bitrate of $2\lceil \log_2(\#\text{nodes})\rceil + b$. For each level of $\mathcal{H}$, we compute the minimum and maximum ID of all nodes referenced in this level, i.e.,

$$\begin{aligned} i_{\min}(l) &:= n : [\mathbf{h}]_n \neq 0 \wedge [\mathbf{h}]_k = 0 \quad \forall \mathbf{h} \in \mathcal{H}, k < n, \\ i_{\max}(l) &:= \max_{\mathbf{h} \in \mathcal{H}(\cdot,l)} \widehat{k}_{\mathbf{h}}. \end{aligned} \tag{15}$$

We store $i_{\min}(l)$ in the bitstream's header for each l. For each voxel, we then store its node ID minus the minimum node ID of that level, again, using a fixed bitrate of

$$r(l) = \lceil \log_2(i_{\max}(l) - i_{\min}(l) + 1) \rceil \tag{16}$$

for each voxel in this level. Finally, we store the scalar quantizer's code values and standard deviations in 20 bits each.

3.4 Reconstruction

In order to reconstruct the original normalized histogram mipmap, we first perform a topological sorting of the nodes Λ in the Mixture Graph. Despite the factorization step resulting in a "soft" topological order (children-before-parents), this step is necessary in order to establish synchronization points for parallel reconstruction. Topological sorting resolves dependencies when reconstructing the original mixtures (also see Fig. 5). We begin by assigning a value of 0 to each source in the graph. Each remaining node with predecessor values v_1, v_2 is assigned a value of $\max(v_1, v_2) + 1$. We call these values v_i the *topological level* of node i and $\max_{v_i}$ the topological depth of the graph. Exploiting the children-before-parents relation described above, this amounts to a linear scan of all nodes.

On parallel architectures, such as GPUs, nodes with identical values can be processed without synchronization. Conversely, this also means that the topological depth is equivalent to the required number of synchronization barriers. Furthermore, since we only need to ascertain that all nodes with smaller values are processed before the first node with a larger value, we can defer evaluating certain nodes. Consider a node with topological level v and a successor level of v_s. If $v_s - v > 1$, then the node can be evaluated in parallel together with node levels $v, v+1, \ldots, v_s - 1$. Since in our experience much more nodes are generated with small values than with large ones, this can be used for load-balancing. We start by processing all nodes with a value of 0 and a successor value of 1. We then continue processing nodes with value of 0 but a successor value of 2, until we have processed at least $\lceil N/p \rceil$ nodes, where N is the total number of nodes and p is the topological depth of the DAG. We then repeat this procedure for the next level, until we reached the sinks in the DAG.

The Mixture Graph not only stores a factorization of the mixtures in $\mathcal{H}$, but also represents the computations necessary to reconstruct mixtures of segment attributes at all scales of a mipmap. Therefore, it is also possible to assign each segment an RGBα color and rebuild the resulting RGBα mipmap efficiently. Traditionally, this requires to assign colors first and then to rebuild the mipmap from scratch. Using the Mixture Graph, only all potentially unique RGBα colors in the resulting mipmap are computed by propagating colors through the Mixture Graph. The mipmap itself is essentially a paletted texture. In the Results section we show that the 1,046 megavoxel Hippocampus data set with 353 segments and a mipmap comprising more than 1,195 million voxels can be factored into a little more than 1 million nodes. Consequently, instead of re-computing the color values of 1,195 million voxels, we only need to update 1 million colors by linear interpolations. A partial update of the transfer function would trigger a "push" scattering propagation of through the graph. Since scattering is much harder to parallelize than gathering, our system only performs full updates using parallel gathering. As we show in the Results section, the performance of such updates is still well within real-time requirements. Note that, apart from other benefits, assigning an RGBα value to each voxel of the aforementioned data set would amount to more than 4.6GB, whereas our data representation requires only 1.36GB.

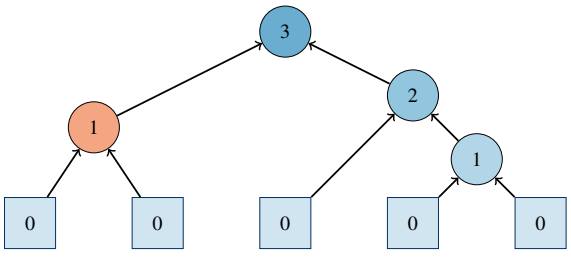

Fig. 5: Topological sorting of the DAG. Node numbers indicate the distance to the closest source. During parallel reconstruction smaller numbers have to be processed before larger numbers. The largest number corresponds sto the number of synchronization steps required. Here, the orange node has only successors with a value of 3. It can be processed together with nodes labelled '1' or with nodes labelled '2'.

Finally, we would like to point out that, as long as the input data contains only one segment ID per voxel, we can always reconstruct the lowest level (input resolution) without loss from the Mixture Graph. Higher-level mixtures are encoded with a small loss, and we would like to refer the reader to the Results section for details.

3.5 Error Propagation

To estimate the error in the reconstruction, the scalar quantizer stores the empiric standard deviation in addition to the code value per bin. The quantization error can be propagated following the standard rules for error propagation during the reconstruction:

$$\begin{aligned} Q = a + b \quad &\rightarrow \delta Q = \sqrt{(\delta a)^2 + (\delta b)^2} \\ Q = a \times b \quad &\rightarrow \delta Q = \sqrt{Q^2 \left(\left(\frac{\delta a}{a} \right)^2 + \left(\frac{\delta b}{b} \right)^2 \right)}. \end{aligned} \tag{17}$$

Sources in the Mixture Graph represent certain input segment IDs $\widehat{\mathbf{e}}_i, \delta\widehat{\mathbf{e}}_i = 0$. Successive linear interpolations in the reconstruction using uncertain weights introduce uncertainties into the mixtures of the input segments. In particular, we consider the linear interpolation between two potentially uncertain vectors $\mathbf{a} \pm \delta\mathbf{a}$, $\mathbf{b} \pm \delta\mathbf{b}$ using an uncertain weight $w \pm \delta w$. Since $\mathbf{a}, \mathbf{b}$ model mixtures of segments, we treat the error of each component i in these vectors separately. We obtain

$$\begin{aligned} [\mu]_i &= (1-w)[\mathbf{a}]_i + w[\mathbf{b}]_i \\ [\delta\mu]_i &= \sqrt{[\mu]_i^2 \left(\left(\frac{[\delta\mathbf{a}]_i}{[\mathbf{a}]_i} \right)^2 + \left(\frac{[\delta\mathbf{b}]_i}{[\mathbf{b}]_i} \right)^2 + 2 \left(\frac{\delta w}{w} \right)^2 \right)}. \end{aligned} \tag{18}$$

Despite the fact that segments are typically spatially coherent, it is worth noting that the above independent treatment of the error propagation is correct since the uncertain weights are independent of the mixture vectors. We also validated this experimentally by sampling weights from standard distributions during the reconstruction. The above error propagation equations provide us with the means to quantify the error of volume region queries.

3.6 Fast Sub-Volume Queries

To compute the histogram over a sub-volume $[x, x+\Delta x] \times [y, y+\Delta y] \times [z, z+\Delta z]$, we follow a footprint assembly [44, 43] approach. First, we decompose the 1D ranges along x, y and z according to their alignment with the mip levels in the normalized histogram mipmap $\mathcal{H}$. Then, the tensor product of these ranges is comprised of rectangular boxes as depicted in Fig. 6. Each of these boxes is tesselated by cubes of a side length l equal to the shortest of the three box extents. Each cube represents a single sample from $\mathcal{H}$ at level $\log_2(l)$. The contribution of each sample to the final result is denormalized (multiplication by l^3) before being accumulated. In order to evaluate samples from $\mathcal{H}$ efficiently, a lookup table is pre-computed on the CPU that stores one mixture per node. We then use error propagation (Eq. 18) to track the

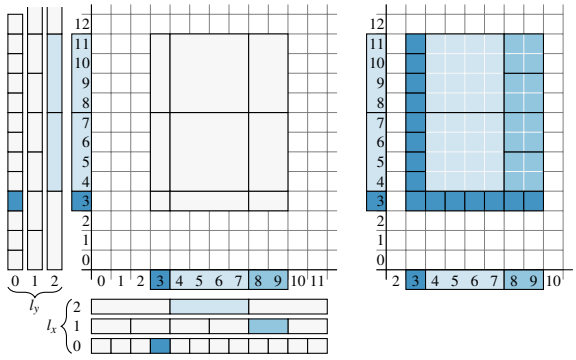

Fig. 6: Footprint assembly in 2D. **Left:** The 2D footprint formed by the tensor product of 1D footprints consists of rectangles. Alignments and extents of samples of the mimmap are shown next to each axis. **Right**: Rectangles in the 2d footprint are tesselated using squares matching the smaller side of each rectangle. Each square corresponds to one sample from the normalized histogram mipmap.

uncertainty of the interpolation weights through the lookup table. This allows us to predict the error made in range queries. Typical errors using 512 bins for quantization are about 0.3%, with the maximum error for a segment not exceeding 0.5%. As a direct benefit of the Mixture Graph only considering convex combinations of nodes, the sum of the expectations always sum up exactly to the volume of the query region. Our error estimate is thus unbiased.

The task of querying ranges of the segmented volume is not well suited for the GPU, since the number of non-zero entries in the sparse mixtures is hard to predict. While it is generally possible to naïvely reconstruct a dense histogram, a dense approach is unlikely to scale well for volumes with a high number of segment IDs, since the sparsity in the range is not used. Furthermore, while we only discussed axis-aligned sub-volumes in this section, footprint assembly can also be performed for "oddly-shaped" regions—albeit at a substantially higher computational cost.

3.7 Implementation Details

Maximum Search. Our compression algorithm relies on quickly finding the best pair of indices i, j with respect to one of the greedy strategies defined in Eq. (12) and Eq. (14). We note that at first the best and second-best pair tend to be well-separated in terms of their score ϕ_1 or ϕ_2. In later stages, progress slows down as choices become more ambiguous. We therefore introduce a threshold τ. In the first stage, we factorize all mixtures $\mathbf{m}$, $\|\mathbf{m}\|_0 > \tau$, skipping over many entries in lower levels of the mipmap. Once we have successfully exhausted options in this search space, we factorize the remaining mixtures with only τ or less components in scanline order. Using a cutoff $\tau = 3$, about half of all mixtures can be exempted in the first stage, resulting in about 2.73× speed increase when compared to a naïve exhaustive search when ϕ_1 is used as a scoring function, and in about 1.79× improvement for ϕ_2. The reason is that ϕ_2 reduces the search space quicker and results in a better performance overall.

Interpolation Symmetries. To find node redundancies, we consider triples (i, j, w) to denote linear interpolations between segments s_i and s_j: $(1-w)s_i + ws_j$. Thus, each triple (i, j, w) is equivalent to $(j, i, 1-w)$. In our implementation, we use this symmetry to improve the accuracy of the quantized weight. To do so, we quantize both w and $1-w$ using the codebook generated by the scalar quantizer and keep the one resulting in the lower error. We then store either (i, j, w) or $(j, i, 1-w)$ in the output stream.

Sparse vectors over $\mathbb{R}^\infty$. While it may seem appealing to implement the data structure for sparse vectors over $\mathbb{R}^\infty$ using a generic hash implementation (e.g., std::unordered_map in C++), we found that the storage requirements of such implementations quickly become prohibitive. We therefore implemented the underlying data structure in C++ without the STL as a vector of pairs (index, value) that we keep sorted with respect to the index.

4 Rendering

To render the segmented volume directly from its Mixture Graph representation, we upload the entire binary representation to the GPU. Specifically, voxel data is stored in shader storage buffer objects (SSBOs); index structures such as node offsets, bit allocation counts etc. are stored as shader uniform constants. In addition, we allocate an SSBO with one vec4 color entry per node to hold the decoded color lookup table. The lookup table is updated with each transfer function update using a compute shader, as outlined before. Since our binary representation is accessed based on bit addresses, we use the uint64 data type provided to shaders by the GPU shader5 extension whenever necessary. For each voxel, we ultimately retrieve some index information (level node offset, bit allocation per voxel, etc.) plus the node ID of the voxel. The node ID is a direct reference into the color lookup table. In essence, rendering from the Mixture Graph is the same as rendering from a paletted texture, with the minor complications that the palette has to be reconstructed prior to rendering and address computations have to be performed in the shader.

While the Mixture Graph allows the efficient computation of any kind of numeric segmentation attribute, we shall for now assume that each segment is assigned an RGBα color through a transfer function. On the GPU, we traverse the Mixture Graph's nodes in topological order. For each node, we gather two colors and linearly interpolate between them using the quantized weight stored at the node. Once the palette is computed, rendering proceeds by identifying the address of any given voxel's raw data in a buffer containing the voxel IDs. We provide the number of levels as a uniform shader constant. Note that while the Mixture Graph stores a full mipmap and thus provides pre-filtered anti-aliasing, we still have to perform the computation of the mip level as well as the interpolation in the shader "manually". This is a drawback shared with virtually all modern paletted compression and rendering methods. To estimate the correct level of detail (LOD), we use the observation that the pixel coverage of a voxel is linear in viewspace depth z [9]. The corresponding equations can also be found in Section 8.14 of the OpenGL 4.5 specifications. We therefore upload viewport, camera parameters, volume dimensions as well as a uniform LOD bias to the shader. In addition, the step size of a volume raymarcher can be adjusted to the LOD, which requires opacity correction to be performed in case of semi-transparent structures.

Empty Space Skipping. The hierarchical nature of the Mixture Graph makes it possible to implement various empty space skipping schemes. Although a full evaluation of different empty space skipping methods is beyond the scope of this paper, our current raymarcher implements a GPU-based scheme that: (i) assigns opacity values of the current transfer function to the leafs, (ii) propagates these values through the Mixture Graph, and, (iii) uses the hierarchy for skipping longer segments. As long as the opacity value in parent voxels falls below a certain threshold, step (iii) traverses the hierarchy upwards to establish larger and larger blocks that can be skipped. This simple yet flexible scheme is enabled by the Mixture Graph's ability to quickly recompute entire mipmaps. It results in an average 3× speedup when compared to the same implementation without mipmaps using a very low threshold of 10^{-8}.

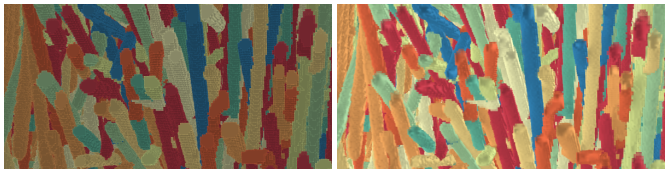

Fig. 7: Using on-the-fly opacity gradients with local ambient occlusion (left) vs. pre-computed normals stored in a second Mixture Graph (right) for shading the Polymer data set.

Lighting. Since our input are volumes of categorical labels, the gradient of the volume cannot be used as normal for shading [26]. Instead, we estimate normals as follows. We start by tagging voxels at the boundary between two segments as *features* and compute an unsigned distance transform to these features using the method of Schneider et al. [45]. We erode our feature mask by two voxels and smooth the distance transform inside a thin band around the segment surfaces. This is necessary to compensate for artifacts due to the finite voxel resolution. We then compute gradients using centered differences, but the resulting gradients will be discontinuous across ridges of the medial axis. To alleviate this, we finally flip the feature mask and smooth the remaining gradients in the interior of the segments using isotropic diffusion. Once we have normal estimates, we quantize the normals [48] using vector quantization to 9 bit. To alleviate banding effects, we use 3D error diffusion [28]. This results in yet another volume of categorical values that we store in another Mixture Graph. Since the leafs of this Mixture Graph store normals, we can evaluate a shading model at each leaf and propagate the resulting light contributions for diffuse and specular component through the graph to obtain a lookup table for each voxel, across each LOD. Note that this also results in properly pre-filtered lighting contributions, since we do not average normals (as is traditionally done) but light contributions. We would also like to note that estimating smooth normals from voxel-based segmentations is challenging; even more so since data sets from neurosciences typically have strongly anisotropic voxels (up to 1:1:7 for the Neocortex). Fig. 7 shows the two lighting modes currently supported by our renderer: on-the-fly computed opacity gradients using discrete differences paired with local ambient occlusion [19] (3-rays, left) and quantized, pre-computed normals stored in a second Mixture Graph (right).

5 Results

Our benchmark configuration comprises a i9-9820X CPU clocked at 3.30GHz and 64GB of RAM, running Windows 10. Although the CPU has 10 cores, our current factorization is mostly sequential and does not utilize the GPU, an NVIDIA RTX Titan with 24GB VRAM.

5.1 Factorization

Table 2: Mixture graph results for the Hippocampus [7], Neocortex [24], and Polymer [55] data sets using the max-reduction criterion. Voxel numbers reported include voxels in higher mip-levels.

Input	**Hippocampus**	**Neocortex**	**Polymer**
resolution	$1178 \times 1125 \times 789$	$2048^2 \times 300$	$614 \times 961 \times 600$
voxels/million	1,195	1,438	404.7
mip levels	12	12	11
segments	353	1,182	15,917
background	52.24%	20.70%	86.34%
Histo Mipmap	**Hippocampus**	**Neocortex**	**Polymer**
non-zeros/voxel	1.006	1.025	1.021
raw size	9.00 GB	11.10 GB	3.10GB
input node pairs	4,976,378	23,551,565	184,642,703
Mixture Graph	**Hippocampus**	**Neocortex**	**Polymer**
nodes	1,034,963	5,202,070	1,930,257
size	1.47 GB	2.14 GB	0.707 GB
bits/voxel	11.92	14.52	17.16
quant. bins	512	512	512
max error	0.00488	0.00523	0.00493
topol. depth	11	22	26
encoding time	3.51 min	2.05 h	82.3 h

Table 2 lists our results for three test data sets. The Hippocampus data set is provided by Calì [7]. It comprises about 1,045 million voxels stemming from electron microscopy imaging, partitioned into 353 segments. Building the normalized histogram mipmap of this data set results in 1.006 non-zero entries in the mixtures per voxel. Storing the histogram requires 9.00 GB. Subsequent factorization of the normalized histogram mipmap into the Mixture Graph results in slightly more then 1 million nodes and 1.47 GB of raw data. This corresponds to a bitrate of 11.92 bits per input voxel, not considering additional voxels in the mipmap hierarchy.

The second data set is a section of a Neocortex data set [24]. It was imaged using electron microscopy as well, but contains a much denser segmentation that uses significantly more IDs. Consequently, the total number of node pairs (i.e., the search space for the factorization) in the input is substantially larger. This results in a significantly longer processing time.

The third data set is a micro CT scan of a fiber-reinforced Polymer [55]. Its high segmentation count poses a computational challenge for our current greedy implementation that we plan to address in the near future by relaxing the greed of the algorithm in favour of faster heuristics. Furthermore, spatially independent factorizations using data windows could lead to increased parallelism. On the other hand, the data set also shows the greatest potential for the Mixture Graph, since we are able to reduce the initial set of over 184.6 million node pairs to just under 2 million, while still maintaining virtually lossless reconstruction.

We quantify the error introduced by our method as the maximum ℓ_1 distance between the original $\mathcal{H}$ and a full reconstruction from the Mixture Graph. We obtain the reconstruction by assigning unit vectors $\widehat{\mathbf{e}}_i$ to leafs i followed by propagation. Errors are thus relative to a unit of 1 and dimensionless. The reason for that choice is that all mixtures are ℓ_1-normalized. Note that, unlike the error estimate discussed in Section 3.5, the error listed in Table 2 (max error) is an actual error based on a encoding-decoding round trip. In contrast, the error estimate quantifies the uncertainty in each ID as required for volumetric queries.

In addition to the number of nodes in the Mixture Graph (including sources and sinks), we also list the topological depth of the graph, since it is equivalent to the number of synchronization steps during parallel reconstruction. We provide timings for all data sets including generation of the histogram mipmap and the factorization into the Mixture Graph.

For all data sets, we observe a significant reduction in the computational workload required to propagate transfer function updates through the entire mipmap. This is reflected by the ratio between voxels in the input mipmap (Hippocampus: 1,195M, Neocortex: 1,438M, Polymer: 404.7M) and the generated nodes (Hippocampus: 1M, Neocortex: 5.2M, Polymer: 1.9M). While the number of generated nodes depends on the number of possible pairs to choose from (Hippocampus: 5M, Neocortex: 23.6M, Polymer: 180.6M), it also depends on the number and location of segments. It is therefore no surprise that the Polymer data with its many fibrous segments shows the best reduction since mixtures involving many segments are quite rare. In any case, the reduction is substantial ($> 4:1$ and almost $100:1$ for the Polymer). Moreover, without the Mixture Graph, hundreds of millions or even billions of voxels have to be revisited on every transfer function change,

Max-occurrence criterion. Both the max-occurrence criterion (Eq. 12) and the max-reduction criterion (Eq. (14)) result in a similar result regarding output size and max error. However, the max-reduction criterion is significantly faster in all three cases (around $2.5\times$ for the Hippocampus and around $2.7\times$ for Neocortex and Polymer), which we attribute to the fact that it prunes the search space quicker. However, to our surprise, the max-reduction criterion also results in a slightly better error, albeit negligibly so. The max-reduction criterion also generates a negligibly ($\leq$ 1 permille) smaller number of nodes.

Quantization parameter. We also conduct experiments to assess the impact of the scalar quantization stage on the overall performance of our method. Our findings are summarized in Table 3. The error reported there is obtained by taking the maximum of the each

Table 3: Impact of the bitrate of the scalar quantizer.

		Hippocampus		
rate/bits	nodes	size	time	error
4	460,210	1.447 GB	2.21 min	0.17475
5	655,225	1.467 GB	2.51 min	0.09384
6	809,324	1.469 GB	2.75 min	0.04213
7	898,406	1.469 GB	2.92 min	0.02525
8	968,962	1.469 GB	3.21 min	0.01228
9	1,029,188	1.470 GB	3.51 min	0.00488
		Neocortex		
rate/bits	nodes	size	time	error
4	3,286,086	2.102 GB	90.25 min	0.23748
5	3,980,659	2.107 GB	95.36 min	0.10286
6	4,441,354	2.132 GB	102.70 min	0.05272
7	4,758,860	2.134 GB	106.37 min	0.02639
8	5,038,103	2.136 GB	116.26 min	0.01288
9	5,300,258	2.139 GB	127.20 min	0.00523
		Polymer		
rate/bits	nodes	size	time	error
4	1,071,148	713.6 MB	71.6 h	0.18248
5	1,315,833	718.3 MB	73.2 h	0.07603
6	1,519,865	721.5 MB	76.6 h	0.04066
7	1,684,252	722.6 MB	79.5 h	0.01932
8	1,812,057	723.1 MB	80.7 h	0.01001
9	1,930,257	724.0 MB	82.3 h	0.00493

quantization bin's individual rmse. Since quantization bins store interpolation weights, the error is in $[0,1]$. Lower bitrates in the scalar quantization stage lead to higher ambiguities between nodes, and, thus, to substantially fewer nodes. To see this, consider that two interpolations $\lambda(i,j,w)$, $\lambda(i,j,w')$ are merged if the quantization bin of w is identical to that of w'. Fewer nodes translate to faster encoding and reconstruction times. The cost to pay is that the reconstruction error roughly doubles for each reduction by one bit. The reason is that each such reduction reduces the precision of the quantized interpolation weight by half. Seemingly surprising, the output size stays more or less constant. This is a consequence of the fact that the bulk of the output size is allotted to voxel data at the finest level. Interestingly, the encoding time decreases a little faster than the number of nodes. We attribute this to our observation that, irrespective of the bitrate, similar choices are made in the early stages of our greedy algorithm. That means that for any bitrate, the search space is reduced by roughly the same absolute amount early on, resulting in fewer factorization steps at lower bitrates.

5.2 Reconstruction

The CPU in our benchmark configuration supports bit manipulation instructions for counting leading and trailing zeros that result in an easier implementation of the footprint gathering algorithm.

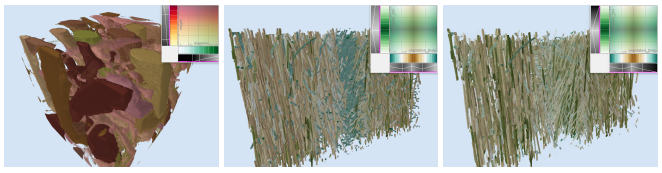

Fig. 8: The Mixture Graph allows for real-time transfer function editing across all scales of the mipmap, and can be used for multidimensional color mapping of different segment attributes. These schemes can be used for various real time visual analysis applications in neuroscience (left), or material science (center and right).

Color lookup table. We evaluate the time it takes to assemble a color lookup table on both CPU and GPU. To do so, we assign colors to the source leaf nodes in the DAG and traverse all remaining nodes in parallel. For each node, we fetch data from its children and perform one interpolation. We measure the execution time of 1,000 repetitions and report the average time of one repetition and GPU timings include the data upload of one float4 color per source. On the CPU, we used OpenMP to parallelize the code, whereas on the GPU, a compute shader is used. This code is executed whenever the user changes the color transfer function such as shown in Fig. 8. For the Hippocampus data set (353 sources, topological depth 11) our code takes 22.7ms on the CPU and 3.5ms on the GPU for computing color information for the more than 1M nodes. For the Neocortex (1,182 sources, topological depth 22) our code takes 129.7ms (CPU) and 11.3ms (GPU). Finally, for the polymer (15,917 sources, topological depth 26) our code takes 57.0ms (CPU) and 12.1ms (GPU). These timings indicate that the limiting factor on the CPU is the number of nodes whereas on the GPU the need to synchronize affects computation times. Enabled by this performance, we implement several interactive multidimensional transfer function editing schemes that can be easily integrated into a wide range of visual exploration workflows. For example, neuroscientists can classify and select neural structures according to geometric features such as shape, volume, length and diameter (Fig. 8 left), and material scientists can cluster fibers into groups based on length or orientation (Fig. 8 middle and right).

Mixture lookup table. We also bench the time it takes to assemble a mixture lookup table on the CPU. We use this lookup table for range queries (both naïve and our footprint assembly algorithm), such as depicted in Fig.9. We start by assigning a mixture $\widehat{\mathbf{e}}_i$ with variance 0 to each leaf node i. In the same fashion as for the color lookup, we propagate this information through the DAG, including the error described in Eq. (18). Each node thus stores two sparse vectors over $\mathbb{R}^\infty$, the expected value μ and its variance $\delta\mu$. For the Hippocampus, computing this lookup takes 68ms, 389ms for the Neocortex, and 171ms for the Polymer. Building this lookup table does not only depend on the number of nodes and the topological depth of the DAG, but also on the initial number of segments as well as the number of non-zero entries in the sparse vectors. This operation will typically be performed only once when loading the data set. Due to the pruning performed during the factorization, the lookup tables easily fit into main memory, with the Neocortex lookup at around 251MB being the largest of the three.

Range queries. We then use this lookup table to measure randomly-aligned range queries covering between 16^3 to 256^3 voxels. For each of the three data sets, we compare a parallel naïve approach that adds the contributions of each voxel at the finest level with our footprint assembly (FA) method that exploits the data hierarchy. Our findings are summarized in Table 4. As can be seen, FA significantly reduces the number of samples necessary to count the segment volumes in a given range. At ranges larger than 8^3, we amortize the setup overhead of FA over the naïve method. Beyond this point, we observe substantial speed-ups that are enabled by our hierarchical data structure.

It is worth noting that the number of fetches made by the FA method depends solely on the domain resolution, as well as on the footprint size and alignment. These numbers are therefore similar for the three

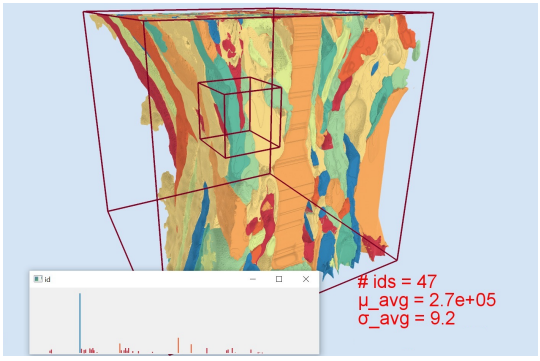

Fig. 9: The Mixture Graph allows for real-time computation of approximate histograms over axis-aligned bounding boxes. The above small box contains segments with an average volume of $270K \pm 9.2$ voxels per segment.

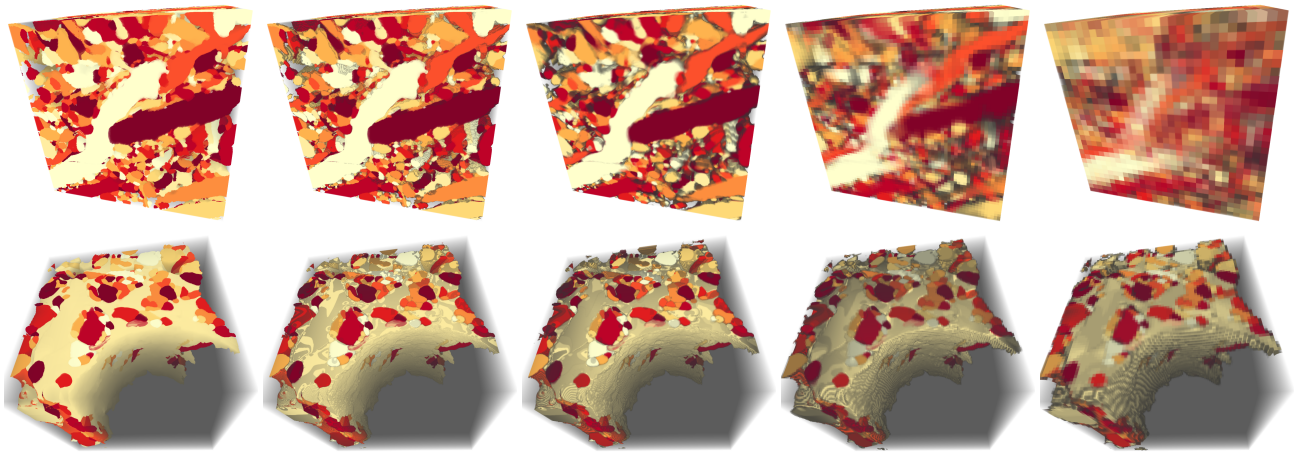

Fig. 10: **Top, left to right:** Direct volume rendering of levels 0,2,4,5, and 6 of the 1.2G voxel Neocortex data set. The data sets strong anisotropy can be seen in the middle images. **Bottom, left to right:** Direct volume rendering of levels $0,\ldots,4$ of the 1.0G voxel Hippocampus data set. For both rows, empty regions were set to a semi-transparent gray, and shading was disabled for a cleaner illustration. These images were rendered directly from the binary Mixture Graph representation at interactive rates on the GPU.

Table 4: Performance of the footprint assembly (FA) algorithm: **Hippocampus** (top), **Neocortex** (center), and **Polymer** (bottom). Left to right: size of the query region, fetches made by the FA algorithm, percentage of fetches saved, time in ms for FA and 10-core parallel naïve approach, and speedup (bold times in seconds).

			time/ms		
size	fetches	saved	footprint	naïve	speed-up
16^3	768.6	81.24%	0.1136	0.4058	3.57×
32^3	3.92K	88.04%	0.3597	1.8547	5.16×
64^3	17.45K	93.34%	1.3577	12.8862	9.49×
128^3	73.9K	96.48%	6.0933	110.5094	18.14×
256^3	264.7K	98.42%	25.2157	**1.2s**	57.60×
16^3	961.40	76.53%	0.1463	0.4015	2.74×
32^3	3.9K	87.96%	0.4091	1.8542	4.53×
64^3	12.7K	95.15%	1.2531	14.5791	11.63×
128^3	63.1K	96.99%	7.7496	160.4324	20.70×
256^3	246.8K	98.53%	89.5613	**2.6s**	28.98×
16^3	925.70	77.40%	0.1344	0.4359	3.24×
32^3	3.9K	88.09%	0.3912	2.4141	6.17×
64^3	17.9K	93.15%	2.1309	29.5320	13.86×
128^3	69.4K	96.69%	15.8873	652.8276	41.09×
256^3	289.6K	98.27%	393.9230	**70.5s**	178.95×

data sets. In contrast, the time is also affected by the number of segmentation IDs and their spatial distribution. For large, contiguously labeled regions, the resulting mixture remains very sparse throughout its computation, whereas for regions with less coherence, many more terms have to be accumulated.

5.3 Rendering

A drawback of our method to derive pre-computed normals are artifacts in the form of *caps* around the medial axis where the segments leave the domain. A second limitation is that we are restricted to light and camera at infinity since we compute lighting contributions without knowledge of position. Note that only the second is a limitation of the Mixture Graph since our method is orthogonal to the normal estimation. Fig. 10 depicts several levels of the Hippocampus and Neocortex data sets, all rendered at the same resolution, to illustrate the downsampling capabilities of our data structure. In the accompanying video, we demonstrate real-time performance using a direct volume raymarcher, sampling at 0.5 voxels and rendering to a 1600×1200 viewport.

5.4 Conclusion & Future Work

In this paper, we have presented the Mixture Graph. This data structure allows us to factorize and compress normalized histogram mipmaps, essentially resulting in a paletted mipmap that provides fast updates of transfer functions for rendering. In our experiments, we observe a decrease from the order of gigavoxels in the mipmap down to the order of mega-nodes in the Mixture Graph. Both quantities reflect the amount of computational work required to re-compute the mipmap. We demonstrate the usefulness of our method for segmented volumes, whose integer nature otherwise prohibit direct filtering, interpolation, and lossy compression. Additionally, we evaluate and present various trade-offs between encoding speed and reconstruction fidelity to guide future users of our method.

Our current implementation considers a single volume and builds the Mixture Graph using serial code. In the future, we would like to explore bricked volumes, since they offer potential for parallelization and out-of-core processing. However, building the Mixture Graph of a bricked volume in parallel is not straightforward. The reason is that each factorization step alters the remaining search space and, consequently, the order of execution may matter significantly. Another direction for future research is to consider larger mixtures (e.g., barycentric interpolation etc.) as the building blocks of the graph. While each mixture would require more storage, we expect that fewer nodes will be generated. Also, the longest path in the graph is expected to become shorter, which would require fewer synchronization operations during parallel transfer function updates. Our current empty space skipping method is efficient but we also plan to investigate how more sophisticated schemes [15] can be combined with out data structure in the future. Finally, we demonstrate that the choice of the greedy scoring function has significant impact on the performance of the construction of the mixture graph. In this context, more research is needed to determine better scoring functions.

Acknowledgments

All authors are funded by the College of Science and Engineering (CSE) at Hamad Bin Khalifa University (HBKU). The data sets used in this work were generated by Calì et al. [7] (Hippocampus) and Kasthuri et al. [24] (Neocortex). The fiber-reinforced polymer data set is provided by Christoph Heinzl [55] .

REFERENCES

[1] A. K. Al-Awami, J. Beyer, D. Haehn, N. Kasthuri, J. W. Lichtmann, H. Pfister, and M. Hadwiger. NeuroBlocks - visual tracking of segmentation and proofreading for large connectomics projects. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):738–746, 2015. https://doi.org/10.1109/TVCG.2015.2467441.

[2] A. K. Al-Awami, J. Beyer, H. Strobelt, N. Kasthuri, J. W. Lichtmann, H. Pfister, and M. Hadwiger. NeuroLines: A subway map metaphor for visualizing nanoscale neuronal connectivity. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2369–2378, 2014. https://doi.org/10.1109/TVCG.2014.2346312.

[3] C. Bajaj, I. Ihm, and S. Park. 3D RGB image compression for interactive applications. *ACM Transactions on Graphics*, 20(1):10–38, 2001. https://doi.org/10.1109/PacificVis.2014.52.

[4] J. Beyer, A. K. Al-Awami, N. Kasthuri, J. W. Lichtmann, H. Pfister, and M. Hadwiger. ConnectomeExplorer: Query-guided visual analysis of large volumetric neuroscience data. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2868–2877, 2013. https://doi.org/10.1109/TVCG.2013.142.

[5] J. Beyer, M. Hadwiger, A. K. Al-Awami, W.-K. Jeong, N. Kasthuri, J. W. Lichtmann, and H. Pfister. Exploring the connectome: Petascale volume visualization of microscopy data streams. *IEEE Computer Graphics and Applications*, 33(4):50–61, 2013. https://doi.org/10.1109/MCG.2013.55.

[6] J. Beyer, H. Mohammed, M. Agus, A. K. Awami, H. Pfister, and M. Hadwiger. Culling for extreme-scale segmentation volumes: A hybrid deterministic and probabilistic approach. *IEEE Transactions on Visualization and Computer Graphics*, 25:1132–1141, 2019. https://doi.org/10.1109/TVCG.2018.2864847.

[7] C. Calì, J. Baghabra, D. Boges, G. Holst, A. Kreshuk, F. Hamprecht, M. Srinivasan, H. Lehväslaiho, and P. Magistretti. Three-dimensional immersive virtual reality for studying cellular compartments in 3D models from EM preparations of neural tissues. *Computational Neurology*, 524(1):23–38, 2016. https://doi.org/10.1002/cne.23852.

[8] A. Cohen, I. Daubechies, and J.-C. Feauveau. Biorthogonal bases of compactly supported wavelets. *Communications on Pure and Applied Mathematics*, 45(5):485–560, 1992. https://doi.org/10.1002/cpa.3160450502.

[9] J. Cohen, M. Olano, and D. Manocha. Appearance-preserving simplification. In *Proceedings of ACM SIGGRAPH*, pages 115–112, 1998. https://doi.org/10.1145/280814.280832.

[10] I. Daubechies and W. Sweldens. Factoring wavelet transforms into lifting steps. *Journal on Fourier Analysis and Applications*, 4(3):247–269, 1998. hhttps://doi.org/10.1007/BF02476026.

[11] W. Ewald and W. Sieg, editors. *David Hilbert's Lectures on the Foundations of Arithmetics and Logic 1917–1933*. Springer Verlag, 2013. https://www.springer.com/la/book/9783540205784.

[12] N. Fout and K.-L. Ma. Transform coding for hardware-accelerated volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1600–1607, 2007. https://doi.org/110.1109/TVCG.2007.70516.

[13] S. Funasaka, K. Nakano, and Y. Ito. Adaptive loss-less data compression method optimized for GPU decompression. *Concurrency and Computation Practice and Experience*, 24(24):1–15, 2010. https://doi.org/10.1002/cpe.

[14] A. Gersho and R. M. Gray. *Vector Quantization and Signal Compression*. Kluwer Academic Press, 1992. https://link.springer.com/book/10.1007/978-1-4615-3626-0.

[15] M. Hadwiger, A. K. Al-Awami, J. Beyer, M. Agus, and H. Pfister. SparseLeap: Efficient empty space skipping for large-scale volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):974–983, 2018. https://doi.org/10.1109/TVCG.2017.2744238.

[16] M. Hadwiger, C. Berger, and H. Hauser. High-quality two-level volume rendering of segmented data sets on consumer graphics hardware. In *IEEE Visualization*, pages 301–308, 2003. https://doi.org/10.1109/VISUAL.2003.1250386.

[17] D. Haehn, S. Knowles-Barley, M. Roberts, J. Beyer, N. Kasthuri, J. W. Lichtmann, and H. Pfister. Design and evaluation of interactive proofreading tools for connectomics. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2466–2475, 2014. https://doi.org/10.1109/TVCG.2014.2346371.

[18] H. Hauser, L. Mroz, G. Bischi, and E. Gröller. Two-level volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 7(3):242–252, 2001. https://doi.org/10.1109/2945.942692.

[19] F. Hernell, P. Ljung, and A. Ynnermann. Local ambient occlusion in direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 16(4):548–559, 2010. https://doi.org/10.1109/TVCG.2009.45.

[20] I. Ihm and S. Park. Waveletbased 3D compression scheme for interactive visualization of very large volume data. *Computer Graphics Forum*, 18(1):3–15, 1999. https://doi.org/10.1111/1467-8659.00298.

[21] W.-K. Jeong, J. Beyer, M. Hadwiger, R. Blue, C. Law, A. Vázquez-Reina, R. C. Reid, J. Lichtmann, and H. Pfister. Ssecrett and NeuroTrace: Interactive visualization and analysis tools for large-scale neuroscience data sets. *IEEE Computer Graphics and Applications*, 30(3):58–70, 2010. https://doi.org/10.1109/MCG.2010.56.

[22] W.-K. Jeong, J. Beyer, M. Hadwiger, A. Vázquez, H. Pfister, and R. T. Whitaker. Scalable and interactive segmentation and visualization of neural processed in EM datasets. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1505–1514, 2009. https://doi.org/10.1109/TVCG.2009.178.

[23] W.-K. Jeong, J. Schneider, S. Turney, B. E. Faulkner-Jones, D. Meyer, R. Westermann, R. C. Reid, J. Lichtmann, and H. Pfister. Interactive histology of large-scale biomedical image stacks. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1386–1395, 2010. https://doi.org/10.1109/TVCG.2010.168.

[24] N. Kasthuri, K. Hayworth, D. Berger, R. Schalek, J. Conchello, S. Knowles-Barley, D. Lee, A. Vázquez-Reina, V. Kaynig, T. Jones, M. Roberts, J. Morgan, J. Tapia, H. Seung, W. Roncal, J. Vogelstein, R. Burns, D. Sussman, C. Priebe, H. Pfister, and J. Lichtmann. Saturated reconstruction of a volume of neocortex. *Cell*, 162(3):648–661, 2015. https://doi.org/10.1016/j.cell.2015.06.054.

[25] V. Kaynig, A. Vázquez-Reina, S. Knowles-Barley, M. Roberts, T. R. Jones, N. Kasthuri, E. Miller, J. Lichtmann, and H. Pfister. Large-scale automatic reconstruction of neuronal processes from electron microscopy images. *Medical Image Analysis*, 22(1):77–88, 2015. https://doi.org/10.1016/j.media.2015.02.001.

[26] V. Lempitsky. Surface extraction from binary volumes with higher-order smoothness. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1197–1204, 2010. https://doi.org/10.1109/CVPR.2010.5539832.

[27] P. Ljung, J. Krüger, E. Gröller, M. Hadwiger, C. Hansen, and A. Ynnermann. State of the art in transfer functions for direct volume rendering. *Computer Graphics Forum*, 35(3):669–691, 2016. https://doi.org/10.1111/cgf.12934.

[28] Q. Lou and P. Stucki. Fundamentals of 3D halftoning. *Springer LNCS*, 1375:224–239, 1998. https://doi.org/10.1007/BFb0053273.

[29] H. M, 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. https://doi.org/10.1109/TVCG.2012.240.

[30] S. Mallat and Z. Zhang. Matching pursuits with time-frequency dictionaries. *IEEE Transactions on Signal Processing*, 12:33973415, 1993. https://doi.org/10.1109/78.258082.

[31] S. Muraki. Volume data and wavelet transforms. *IEEE Computer Graphics and Applications*, 13(4):50–56, 1993. https://doi.org/10.1109/38.219451.

[32] Neurodata.io. The OpenConnectome Project. www.openconnectomeproject.org, 2020. accessed 29 July 2020.

[33] K. G. Nguyen and D. Saupe. Rapid high quality compression of volume data for visualization. *Computer Graphics Forum*, 20(3):49–57, 2001. https://doi.org/10.1111/1467-8659.00497.

[34] P. Ning and L. Hesselink. Vector quantization for volume rendering. In *Proceedings of the Workshop on Volume Visualization*, pages 69–74, 1992. https://doi.org/10.1145/147130.147152.

[35] R. Patel, Y. Zhang, J. Mak, A. Davidson, and J. Owens. Parallel lossless data compression on the GPU. In *Innovative Parallel Computing (InPar)*, pages 1–9, 2012. https://doi.org/10.1109/InPar.2012.6339599.

[36] D. Pavić and L. Kobbelt. Two-colored pixels. *Computer Graphics*

Forum, 29(2):743–752, 2010. https://doi.org/10.1111/j.1467-8659.2009.01644.x.

[37] F. Reichl, M. Treib, and R. Westermann. Visualization of big SPH simulations via compressed octree grids. In *IEEE International Conference on Big Data*, pages 71–78, 2013. https://doi.org/10.1109/BigData.2013.6691717.

[38] J. Rissanen and G. G. L. Jr. Arithmetic coding. *IBM Journal of Research and Development*, 23(2):149–162, 1979. https://doi.org/10.1147/rd.232.0149.

[39] W. G. Roncal, M. Pekala, V. Kaynig-Fittkau, D. M. Kleissas, J. T. Vogelstein, H. Pfister, R. Burns, R. J. Vogelstein, M. A. Chevillet, and G. D. Hager. VESICLE: Volumetric evaluation of synaptic interfaces using computer vision at large scale. In *Proc. British Machine Vision Conference*, pages 81.1–81.13, 2015. https://doi.org/10.5244/C.29.81.

[40] R. Rubinstein, M. Zibulevsky, and M. Elad. Double sparsity: Learning sparse dictionaries for sparse signal approximation. *IEEE Transactions on Signal Processing*, 58(3):1553–1564, 2010. https://doi/org/10.1109/TSP.2009.2036477.

[41] A. Said and W. A. Pearlman. A new fast and efficient image codec based on set partitioning in hierarchical trees. *IEEE Transactions on Circuits and Systems for Video Technology*, 6(3):243–250, 1996. https://doi.org/10.1109/76.499834.

[42] R. Schalek, D. Lee, N. Kasthuri, A. Peleg, T. Jones, V. Kaynig, D. Haehn, H. Pfister, D. Cox, and J. Lichtmann. Imaging a $1mm^3$ volume of rat cortex using multibeam SEM. In *Microscopy and Microanalysis*, pages 582–583, 2016. https://doi.org/10.1017/S1431927616003767.

[43] A. Schilling and G. Knittel. System and method for mapping textures onto surfaces of computer-generated objects, May 2001. US Patent 6,236,405: https://portal.unifiedpatents.com/patents/patent/US-6236405-B1.

[44] A. Schilling, G. Knittel, and W. Strasser. Texram: A smart memory for texturing. *IEEE Computer Graphics and Applications*, 16(3):32–41, 2002. https://doi.org/0.1109/38.491183.

[45] J. Schneider, M. Kraus, and R. Westermann. GPU-based Euclidean distance transforms and their application to volume rendering. *Springer CCIS*, 68:215–228, 2009. https://doi.org/10.1007/978-3-642-11840-1_16.

[46] J. Schneider and R. Westermann. Compression domain volume rendering. In *Proceedings of IEEE Visualization*, pages 293–300, 2003. https://doi.org/10.1109/VISUAL.2003.1250385.

[47] E. Sitardi, R. Mueller, T. Kaldewey, G. Lohmann, and K. A. Ross. Massively-parallel lossless data decompression. In *International Conference on Parallel Processing (ICPP)*, pages 242–247, 2016. https://doi.org/10.1109/ICPP.2016.35.

[48] M. Tarini, P. Cignoni, C. Rocchini, and R. Scopigno. Real time, accurate, multi-featured rendering of bump mapped surfaces. *Computer Graphics Forum*, 19(3):119–130, 2000. https://doi.org/10.1111/1467-8659.00404.

[49] M. Treib, F. Reichl, S. Auer, and R. Westermann. Interactive editing of gigasample terrain fields. *Computer Graphics Forum*, 31(2):383–392, 2012. https://doi.org/10.1111/j.1467-8659.2012.03017.x.

[50] A. Vázquez-Reina, M. Gelbart, D. Huang, J. Lichtmann, E. Miller, and H. Pfister. Segmentation fusion for connectomics. In *IEEE International Conference on Computer Vision (ICCV)*, pages 177–184, 2011. https://doi.org/10.1109/ICCV.2011.6126240.

[51] K.-C. Wang, K. Lu, T.-H. Wei, N. Shareef, and H.-W. Shen. Statistical visualization and analysis of large data using a value-based spatial distribution. In *IEEE Pacific Visualization Symposium*, pages 161–170, 2017. https://doi.org/10.1109/PACIFICVIS.2017.8031590.

[52] K.-C. Wang, N. Shareef, and H.-W. Shen. Image and distribution based volume rendering for large data sets. In *IEEE Pacific Visualization Symposium*, pages 26–35, 2018. https://doi.org/10.1109/PacificVis.2018.00013.

[53] A. Weißenberger and B. Schmidt. Massively parallel Huffman decoding on GPUs. In *International Conference on Parallel Processing (ICPP)*, pages 27.1–27.10, 2018. https://doi.org/10.1145/3225058.3225076.

[54] A. Weißenberger and B. Schmidt. Massively parallel ANS decoding on GPUs. In *International Conference on Parallel Processing (ICPP)*, pages 100.1–100.10, 2019. https://doi.org/10.1145/3337821.3337888.

[55] J. Weissenböck, A. Amirkhanov, W. Li, A. Reh, A. Amirkhanov, E. Gröller, J. Kastner, and C. Heinzl. FiberScout: An interactive tool for exploring and analyzing fiber reinforced polymers. In *IEEE Pacific Visualization Symposium*, pages 154–160, 2014. https://doi.org/10.1109/PacificVis.2014.52.

[56] L. Williams. Pyramidal parametrics. In *Proceedings of ACM SIGGRAPH*, pages 1–11, 1983. https://doi.org/10.1145/964967.801126.