Interactive Volume Exploration of Petascale Microscopy Data Streams Using a Visualization-Driven Virtual Memory Approach

Markus Hadwiger, Johanna Beyer, Won-Ki Jeong, Hanspeter Pfister

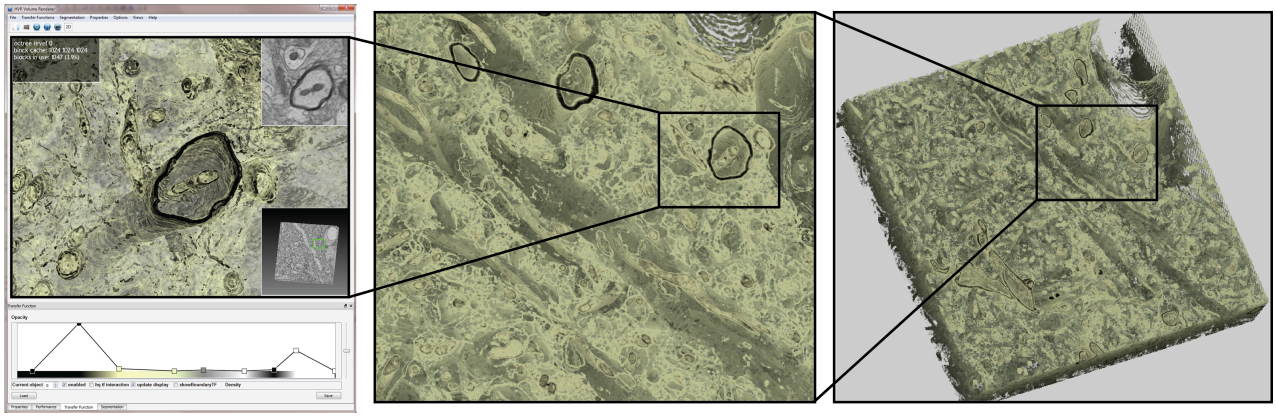

Fig. 1. **Our system** is the first to enable neuroscientists to interactively explore petascale volume data resulting from high-throughput electron microscopy data streams. The volume can be visualized while the acquisition is still in progress, and without pre-processing all data into a 3D multi-resolution hierarchy as required by all previous systems. Shown here: $21,494 \times 25,790 \times 1,850$ mouse cortex.

Abstract— This paper presents the first volume visualization system that scales to petascale volumes imaged as a continuous stream of high-resolution electron microscopy images. Our architecture scales to dense, anisotropic petascale volumes because it: (1) decouples construction of the 3D multi-resolution representation required for visualization from data acquisition, and (2) decouples sample access time during ray-casting from the size of the multi-resolution hierarchy. Our system is designed around a scalable multi-resolution virtual memory architecture that handles missing data naturally, does not pre-compute any 3D multi-resolution representation such as an octree, and can accept a constant stream of 2D image tiles from the microscopes. A novelty of our system design is that it is *visualization-driven:* we restrict most computations to the visible volume data. Leveraging the virtual memory architecture, missing data are detected during volume ray-casting as cache misses, which are propagated backwards for on-demand out-of-core processing. 3D blocks of volume data are only constructed from 2D microscope image tiles when they have actually been accessed during ray-casting. We extensively evaluate our system design choices with respect to scalability and performance, compare to previous best-of-breed systems, and illustrate the effectiveness of our system for real microscopy data from neuroscience.

Index Terms—petascale volume exploration, high-resolution microscopy, high-throughput imaging, neuroscience.

1 Introduction

Recent advances in high-resolution microscopic imaging result in volume data of extreme size. In neuroscience, electron microscopy (EM) volumes of brain tissue are produced by physically cutting very thin sections of about 25-50nm, and imaging each section at 3-5nm pixel resolution [4]. This resolution is necessary in order to be able to trace neural connections in the area of Connectomics [4, 23], which is the main target domain of the visualization system presented in this paper. However, even sub-millimeter tissue blocks imaged at such resolutions comprise terabytes of raw data, and neuroscientists target much larger volumes of several petabytes. A further complication is the acquisition time itself. Measuring a single terabyte of data, even by using fully automated Scanning Electron Microscopes (SEM), may take half a week [4]. As neuroscientists strive to measure large blocks of brain tissue to enable the analysis of brain function, a *high-throughput* acquisition process has to continuously stream data over months or even years. In this scenario, existing visualization approaches fail. Traditional algorithms require complete knowledge of all data, for example in order to pre-compute an octree or kD tree multi-resolution representation. For high-throughput microscopy this is infeasible, since pre-processing the data into a hierarchical representation incurs an unacceptably large gap between acquisition and visualization. Therefore, it is necessary to develop novel visualization paradigms and systems in order to facilitate the interactive exploration and analysis of large-scale microscopy data streams.

We have identified the following major system design goals:

(G1) Scalability to the petascale, even for dense instead of sparse data.

(G2) Accommodate high-throughput acquisition of 2D microscopic image tile streams. New image data can arrive continuously.

(G3) Visualize incomplete, incompletely registered, and changing data. Image alignment information can be updated dynamically. Data can be re-imaged dynamically with different parameters.

(G4) Accommodate highly anisotropic (e.g., 1:10) data.

(G5) Avoid pre-computation that requires knowledge of all data.

(G6) Avoid computation and storage for data that are not visible in the visualization as much as possible.

- *Markus Hadwiger and Johanna Beyer are with King Abdullah University of Science and Technology (KAUST), E-mail: {markus.hadwiger | johanna.beyer}@kaust.edu.sa.*
- *Won-Ki Jeong is with Ulsan National Institute of Science and Technology (UNIST), E-mail: wkjeong@unist.ac.kr.*
- *Hanspeter Pfister is with School of Engineering and Applied Sciences at Harvard University, E-mail: pfister@seas.harvard.edu.*

Manuscript received 31 March 2012; accepted 1 August 2012; posted online 14 October 2012; mailed on 5 October 2012.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

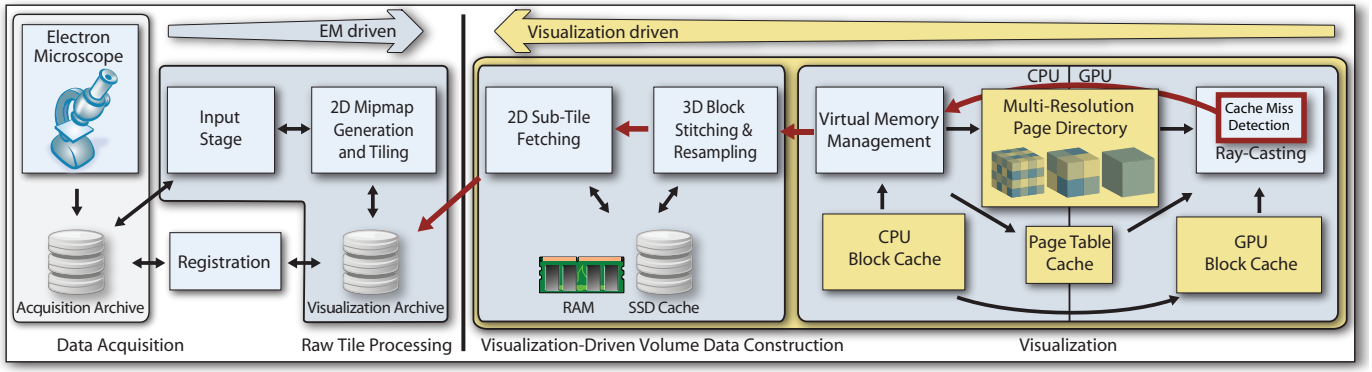

Fig. 2. **System overview.** Petascale volumes are acquired as a stream of image tiles from the microscope. Each raw image tile is processed individually in the input stream. Everything else is *visualization-driven*: Ray-casting operates in virtual volume space, detecting cache misses for visible volume blocks. Only these blocks are then constructed in 3D by stitching and resampling the corresponding tiles from the 2D input stream.

This paper introduces the first visualization system that fulfills these goals, enabling the interactive exploration of petascale microscopy data streams. Our system combines the following contributions:

- A novel *visualization-driven* paradigm for out-of-core volume construction and visualization. Only visible 3D volume blocks are constructed from the raw 2D image data (G1, G2, G5, G6).
- A novel *multi-resolution virtual memory* scheme, where all data comprise a *virtual multi-resolution volume*, naturally handling missing data and dynamic updates (G2, G3). Our architecture achieves scalability via a multi-level page table hierarchy (G1). It naturally accommodates highly anisotropic data (G4), in contrast to an octree that would become unbalanced and less efficient.
- Visible image data are resampled directly into the resolution required for visualization, avoiding the pre-computation of a 3D multi-resolution hierarchy (G5, G6).

This visualization-driven pipeline is enabled by a GPU-based ray-caster that detects visible data in virtual volume space, generating *cache misses* for missing physical data. These cache misses are then propagated backwards to construct, fetch, and cache only visible data.

2 Related Work

Our system is related to a lot of prior work, and we can only highlight the most important connections here. Our visualization stage uses GPU volume ray-casting [20], which has become the most common approach for GPU volume rendering. However, GPU volume renderers are often restricted by GPU memory size. In order to accommodate large volumes, out-of-core and multi-resolution approaches have been developed. LaMar et al. [21] and Weiler et al. [31] were among the first to use hierarchical octree bricking schemes for hardware-assisted volume rendering. Other hierarchical approaches were proposed by Boada et al. [3] and Guthe et al. [11], who use a hierarchical wavelet representation and screen-space error estimation for LOD selection. All previous multi-resolution volume renderers require the multi-resolution hierarchy to be built in a pre-process, which is not feasible for our scenario of dynamically streaming image data. A pre-processing step is also required by all previous systems that support streaming of volume data for progressive rendering, such as the *ViSUS* system [28]. The *ImageVis3D/Tuvok* system [19] supports both sliced-based volume rendering and ray-casting. Large data are subdivided using a kD tree, which is also exploited for distributed rendering [8]. Each brick in the tree is rendered in one rendering pass. There is relatively little published work on single-pass GPU octree ray-casting. Gobbetti et al. [10] determine the potential visibility of octree nodes together with the corresponding partial tree on the CPU, which is then downloaded to the GPU. Octree traversal on the GPU follows explicitly stored rope links between adjacent nodes. Actual visibility is only taken into account in an indirect manner by using occlusion queries. Octree traversal without rope links is usually performed by adapting kD tree traversal algorithms to octrees [6, 7], most of which were developed for ray-tracing geometry [9, 27, 15] or iso-surfaces [17]. The visualization stage of our system performs single-pass ray-casting, detecting the visibility of small blocks on-the-fly. This is also done by the *Gigavoxels* [6] and *CERA-TVR* [7] systems. In contrast to our virtual memory scheme, however, these systems perform explicit octree traversal using the kd-restart algorithm [9]. The data resolution is refined or coarsened by iteratively changing the set of nodes/bricks that are resident in node/brick pools stored in GPU memory. This requires holding the entire path from every leaf to the root in GPU memory, and can result in large numbers of updates per frame. Our system avoids both of these drawbacks. Detailed comparisons of our system with the approaches of these systems are given in Section 8. Clipmaps [29] are an approach for rendering very large mipmaps that essentially also uses a virtual memory space. However, clipmaps have to use a fixed toroidal updating scheme. In contrast, our architecture can address small 3D blocks that are packed arbitrarily into a larger cache texture. This packing is similar to adaptive texture maps [18], but with fully dynamic updates. Virtual 2D texturing approaches have been proposed for adaptive shadow map rendering [22], as well as for state-of-the-art game engines [30]. In contrast to our system, these approaches combine virtual memory management with a tree structure. Another class of large-scale volume rendering systems is purely CPU-based, in order to avoid GPU memory limitations altogether. Much research has been devoted to volume rendering on large supercomputers [5, 16, 26]. This is especially useful in the context of *in-situ* visualization of large-scale simulations, where the visualization is computed on the same machine as the data, avoiding the need to move large data. However, this is not a feasible approach for microscopy data. Our data streams do not originate from large-scale simulations, but from acquisition setups that are not directly connected to a supercomputer. Our system streams data to the GPU-based visualization, but only as required by actual visibility.

3 System Overview

Figure 2 depicts an overview of our system. Although image data propagate from left to right, the majority of the pipeline (Fig. 2, right half) is *visualization-driven* by the actual visibility of small 3D blocks (32^3 voxels) determined during ray-casting, which is propagated from right to left. After introducing basic terminology, we give an overview of each pipeline stage from left to right. The subsequent sections then discuss each stage in reverse order, following the visualization-driven nature starting on the right with visualization and going backwards.

Terminology. Our system is inspired by standard virtual memory architecture [13], and throughout the paper we use similar terminology (see Fig. 3). We *virtualize* 3D volumes by subdividing them into small 3D *blocks* (e.g., 32^3 voxels). Only the *working set* of currently required blocks is *resident* in a large 3D *cache* (texture), which is updated dynamically. The original volume becomes a *virtual volume* that is accessed via a *page table*: a 3D index texture where each voxel is a *page table entry* that maps the corresponding block's position in the cache, or is flagged as *unmapped*, i.e., not resident in the cache. The same approach can be used to virtualize page tables, referring to the top-level page table in the resulting hierarchy as the *page directory*.

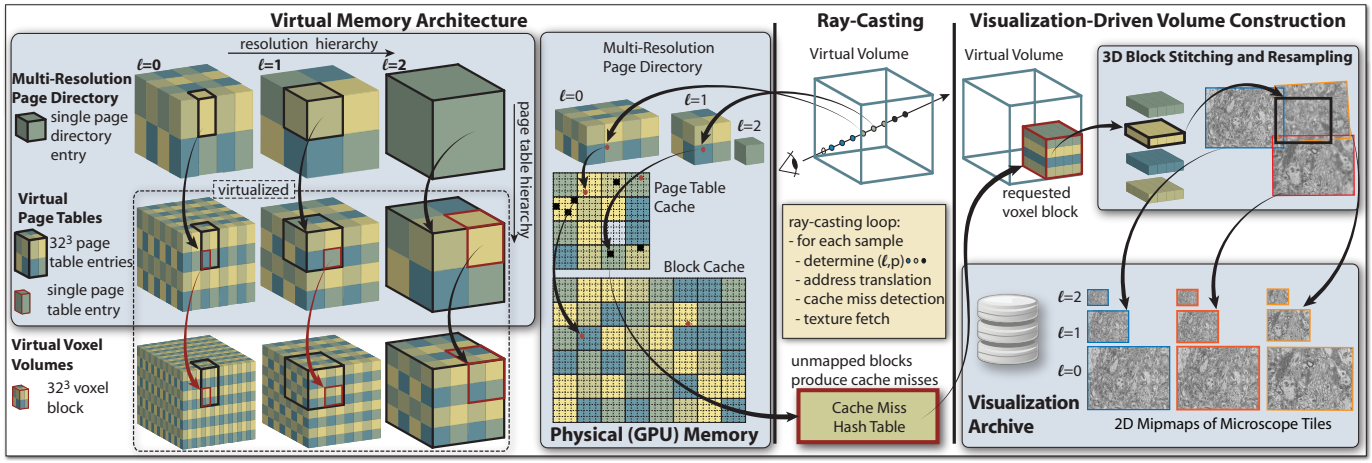

Fig. 3. **Virtual memory architecture and visualization-driven pipeline.** A *virtual multi-resolution volume* comprises a *hierarchy of resolution levels* ℓ (left, horizontal), each of which is virtualized via a *hierarchy of page tables* (vertical). Ray-casting accesses virtual multi-resolution addresses $(\ell, \mathbf{p})$ (center right), performing on-the-fly translation to addresses in the *block cache* via the *multi-resolution page directory* and the *page table cache* (center left). Accesses to blocks in virtual memory that are *unmapped*, i.e., not mapped to a cache block, generate *cache misses* (center right). These enable *visualization-driven* construction of volume data (right), as well as updating page table cache and block cache only for visible blocks.

Data acquisition (Section 7). The microscope deposits each acquired *image tile* (e.g., $12,000 \times 12,000$ pixels) into the *acquisition archive*. This archive stores the acquired raw data together with additional meta information including magnification, position, and orientation (the *alignment matrix*) of acquired image tiles, and is either co-located with the microscope or is a central shared file system.

Raw tile processing (Section 6). This stage continuously polls the acquisition archive for new image tiles, and automatically processes each new tile for storage in the *visualization archive*. This can be the same actual storage used for acquisition, but for organizational reasons it is often better to separate the two archives. Processing comprises construction of a 2D mipmap for each tile, subdivision of each mipmap level into smaller *sub-tiles* for optimized disk access, and optional image compression. Each new raw image tile is processed immediately, independent of tile registration/alignment and visualization.

Registration. Computing the alignment of raw image tiles is an independent component outside the scope of this paper that asynchronously updates the alignment matrices associated with raw image tiles. However, in order to facilitate coarse-to-fine registration, this component can access the 2D tile mipmaps stored in the visualization archive. No image data are changed by registration. Actual stitching of tiles is performed only on-demand in the volume construction stage of our pipeline. Updates of alignment matrices are picked up by the input stage of *raw tile processing* and are propagated forward.

Visualization-driven volume data construction (Section 5). This stage is driven entirely by the visualization stage, which requests voxel data for visible 3D blocks from a resolution level matching the current display resolution. In order to fulfill these requests, first all 2D image sub-tiles that intersect the 3D target block are determined and fetched. The requested 3D voxel data are then created by stitching and resampling these 2D sub-tiles directly into the 3D target grid and resolution. Stitching is determined by the alignment matrix associated with each image tile. Fast stitching and resampling to any target resolution is facilitated by the 2D tile mipmaps stored in the visualization archive.

Visualization (Section 4). The visualization component of our pipeline performs GPU-based ray-casting. However, its design differs from previous systems in several important aspects. Instead of creating and traversing a tree structure—such as an octree or a kD tree—our system employs a multi-level, multi-resolution virtual memory architecture (Fig. 3) that scales well to extremely large volume sizes. Volume space comprises a *virtual multi-resolution volume* that is sampled directly during ray-casting (Section 4.2). Missing data generate *cache misses* at the granularity of small 3D blocks, which by definition are visible data. These cache misses generate requests for data that are propagated backwards in the pipeline, triggering the visualization-driven construction of volume data from 2D image tiles (Section 5).

4 Visualization

The visualization stage of our pipeline has been designed to operate in a fully virtualized volume space, which is a crucial design goal of our system. We achieve this by considering the 3D bounding box of the tissue block to be imaged to comprise a *virtual multi-resolution volume*. This structure is similar to a virtualized 3D mipmap. However, the down-sampling ratio between successive resolution levels can be different for each axis in order to accommodate anisotropic voxel aspect ratios. We access this virtual volume space via an efficient multi-level, multi-resolution virtual memory management architecture. Compared to a traditional octree or kD tree multi-resolution representation, using a virtual multi-resolution volume has the following main advantages:

- It is more efficient for deep resolution hierarchies. No tree traversal is required, and no tree structure needs to be maintained. Maximum efficiency is achieved for *any* desired resolution when all visible data of that resolution are resident in the cache. Rendering arbitrary slices also becomes a straightforward operation.
- Any sample can be fetched directly from any resolution level. This enables jumping between resolutions without constructing intermediate lower resolutions, which greatly helps to reduce the latency incurred by on-demand volume construction (Section 5).
- The down-sampling ratio between resolution levels can be arbitrary. For the anisotropic voxel aspect ratios of most EM data, this avoids the problem of an octree becoming unbalanced when the resolution is not decreased along all three axes in every level.

4.1 Virtual Multi-Resolution Volume Representation

In order to determine a 3D reference space for our entire pipeline, the extents of the 3D tissue block to be imaged are queried from the electron microscope at startup. This space then constitutes *virtual volume space*, which is addressed via normalized floating-point coordinates $\mathbf{p} = (x, y, z) \in [0,1]^3$. Furthermore, this space is extended to form a *virtual multi-resolution volume* by introducing a *hierarchy of resolution levels* (Fig. 3 left, horizontal). This facilitates addressing voxels of a desired size via a *virtual multi-resolution address* $(\ell, \mathbf{p})$, where the integer ℓ is the resolution level. During ray-casting, samples are fetched at positions $(\ell, \mathbf{p})$, computing ℓ from the desired level of detail (Section 4.2). However, before actual voxel data can be accessed, $(\ell, \mathbf{p})$ must be translated to a coordinate in cache (texture) space, which is done on the fly using a *hierarchy of page tables* (Fig. 3 left, vertical).

4.1.1 Resolution Hierarchy

The virtual multi-resolution volume is comprised of discrete resolution levels, from $\ell = 0$ (highest resolution) to $\ell = \ell_{max}$ (coarsest resolution). Fig. 3 (left) depicts this hierarchy horizontally. In contrast to a mipmap

or an octree, our hierarchy does not enforce a fixed down-sampling ratio between levels. Every resolution level can be imagined as comprising its own volume in the normalized $[0,1]^3$ volume space, but without constraining the amount of voxels to be any particular fraction of the next finer level. Addressing any voxel via a normalized coordinate $\mathbf{p}$ is straightforward when the resolution of each level in voxels is known. Being able to choose an arbitrary resolution for each level enables arbitrary resolution reduction between levels. This would also facilitate the efficient representation of sampled scale space, such as with a hybrid pyramid scheme [24]. There, the scale parameter increases by less than a factor of two between levels, and the grid resolution is only reduced (halved) every cumulative increase of scale by a factor of two.

Access to resolution levels. The volume corresponding to a level ℓ can be accessed by using virtual multi-resolution addresses $(\ell, \mathbf{p})$. The access is performed via the page table hierarchy of level ℓ. An important trait of our architecture is that, although any level ℓ can be accessed directly, the only multi-resolution structure that is required to do this is the small *multi-resolution page directory* described below. Apart from this, the individual volumes comprising the resolution hierarchy are fully *virtualized* and only represented implicitly.

Resolution reduction. We start with the full resolution, and choose each axis in the next level to be either half or the same resolution, depending on voxel anisotropy, so that anisotropy is gradually reduced. Fig. 3 (left) shows a simple version of this process, where the depicted anisotropy is $1:2$ from $\{x,y\}:z$ in $\ell=0$, which is reduced to $1:1$ from $\ell=1$ onward. This is achieved by defining the same resolution in z for $\ell=0$ and $\ell=1$. Section 8.1.1 discusses more details.

4.1.2 Page Table Hierarchy

The individual virtualized volumes comprising the resolution hierarchy described above are accessed via a hierarchy of page tables, which is depicted vertically in Fig. 3 (left). The basic idea is to virtualize a large volume for out-of-core visualization using a 3D page table containing indices for addressing small voxel blocks (e.g., 32^3) stored in the 3D *block cache* (Fig. 3, center) with a single page table entry. The block cache is implemented as a 3D texture that serves as "physical" memory in the virtual memory scheme. In addition to the index, each page table entry can be marked as either *unmapped* or *empty* (see below). However, for very large volumes, the page table itself becomes too large. We therefore apply the basic concept recursively, obtaining a hierarchy of page tables with t levels, where all levels except the root of the hierarchy (top row of Fig. 3, left) are also virtualized. We call this root page table the *multi-resolution page directory*.

```
1:  float3 p, ray; int4 pDirEntry, pTableEntry;
2:  function SampleCurrentPosition( int ℓ )
3:      float sample = 0.0;
4:      if !PageDirectoryIndexSameAsPrevious( ℓ, p ) then
5:          int3 pDirAddress = pageDirBase[ ℓ ].xyz + p * r_pd(ℓ).xyz;
6:          pDirEntry = texture3D( texPageDir, pDirAddress );
7:      end if
8:      int pagingFlag = pDirEntry.w;
9:      if pagingFlag != UNMAPPED && pagingFlag != EMPTY then
10:         if !PageTableIndexSameAsPrevious( ℓ, p ) then
11:             int3 pTableAddress = pDirEntry.xyz + ( p * r_pt(ℓ).xyz ) % b_pt.xyz;
12:             pTableEntry = texture3D( texPageTableCache, pTableAddress );
13:         end if
14:         pagingFlag = pTableEntry.w;
15:         if pagingFlag != UNMAPPED && pagingFlag != EMPTY then
16:             int3 voxelAddress = pTableEntry.xyz + ( p * r_v(ℓ).xyz ) % b_vox.xyz;
17:             sample = texture3D( texBlockCache, voxelAddress );
18:         else
19:             p += SkipEmptySpacePageTableEntryRecursive( ray, ℓ, p );
20:         end if
21:     else
22:         p += SkipEmptySpacePageDirectoryEntry( ray, ℓ, p );
23:     end if
24:     if pagingFlag == UNMAPPED then
25:         int blockID = ComputeBlockIdForPosition( ℓ, p );
26:         ReportCacheMiss( blockID );
27:     end if
28:     return sample;
29: end function
30: function CastRayForImagePixel( int x, int y )
31:     p = FetchRayStart( x, y ); ray = FetchRayDirection( x, y );
32:     repeat
33:         int ℓ = ComputeDesiredSampleResolutionLevel( p );
34:         float sample = SampleCurrentPosition( ℓ );
35:         ApplyTransferFunctionAndCompositing( sample );
36:         p += AdvanceToNextSample( ray, ℓ );
37:     until early ray termination or volume exit
38: end function
```

Fig. 4. **Address translation and ray-casting.** Each virtual multi-resolution address $(\ell, \mathbf{p})$ is translated to a block cache texture coordinate on-the-fly.

Multi-resolution page directory. Since the page directory is not virtualized, it is always completely resident in GPU memory. For a virtual multi-resolution address $(\ell, \mathbf{p})$, the parameter ℓ determines which resolution level should be accessed. Each directory entry references a block of page table entries in the *page table cache* (Fig. 3), which stores blocks of the same size from any hierarchy or resolution level.

Page tables. Each page table entry references either another block of page table entries in the hierarchy level below it, which is then also stored in the *page table cache*, or a block of voxels in the *block cache*. The former case is only relevant if there are more than two page table hierarchy levels ($t > 2$), i.e., more than the page directory and the page table below it. Figure 3 shows only two hierarchy levels ($t = 2$).

Page table entry flags. Every page table entry has a flag field that can be set to either one of two special values (see also Fig. 4):

Unmapped entries correspond to blocks in the virtual multi-resolution volume that are not mapped to a block in the block cache. (Mapped blocks simply have the flag cleared.) If such a page table entry is accessed during ray-casting, a *cache miss* is generated. Additionally, the extent of the block will either be skipped over as empty space, or a lower resolution will be selected dynamically (Section 4.2.3). The cache miss then results in a request for the volume construction stage (Section 5) to asynchronously construct the block's data. After the data have arrived at the visualization stage, the page table entry will be mapped. This resets the unmapped flag. If, however, the volume construction stage instead replies that the data for the requested block do not exist (yet), the unmapped flag will instead be changed to *empty*.

Empty entries correspond to blocks in the virtual multi-resolution volume that are known to be empty. Blocks can be classified as empty for two different reasons: (1) The voxel data of the block are either all zero or do not exist, and will thus be classified as empty irrespective of the current transfer function. This case is reported by the volume construction stage (Section 5). (2) The block's data are available, but currently invisible (transparent) given the current transfer function. This case is identified whenever the transfer function changes. Ray-casting performs empty space skipping for empty blocks (Section 4.2.3). However, no cache miss will be reported for empty blocks, because they never need to be mapped to a block in the block cache.

4.1.3 Address Translation

Figure 3 (center) illustrates the individual steps of translating a virtual multi-resolution address $(\ell, \mathbf{p})$ to a physical address in the block cache using the page table hierarchy. The implementation is described by the pseudo code in Fig. 4. We show this process for two hierarchy levels ($t = 2$), but the extension to more levels is straightforward.

Due to anisotropic down-sampling, the multi-resolution page directory cannot be stored as a regular mipmap. Instead, we pack all resolution levels $\ell \in [0, \ell_{max}]$ into a single 3D texture *texPageDir* (Fig. 4). The origin of each level is retrieved from a small array *pageDirBase*. In order to compute look-up indexes from normalized coordinates $\mathbf{p}$, we use the following constants: $r_{pd}(\ell)$ is the total number of entries in the page directory for resolution level ℓ. $r_{pt}(\ell)$ is the total number of entries in the virtualized page table of level ℓ. $r_v(\ell)$ is the total number of voxels in the virtualized volume of level ℓ. Each page table block comprises b_{pt} entries, and each voxel block comprises b_{vox} voxels.

4.1.4 Page Table and Cache Management

The multi-resolution page directory, the page table cache, and the block cache are each implemented with one 3D texture whose content is managed by the CPU using a standard least/most-recently used

(LRU/MRU) scheme [13]. However, updates are initiated by the cache misses reported by the ray-caster (Section 4.2.2). For each cache miss, the CPU first checks if it can be fulfilled from the *CPU block cache*, which is simply a larger version in CPU memory of the block cache in GPU memory (Fig. 2). If the required block is resident in the CPU block cache, it is downloaded into the GPU block cache, and the page directory and page table cache are updated accordingly. Otherwise, a request for the block of 3D volume data is issued asynchronously to the volume data construction stage (Section 5). In this case, the download of block data, as well as the corresponding page table updates, are deferred until the 3D block constructed by that stage has arrived.

4.2 Ray-Casting Virtual Multi-Resolution Volumes

The pseudo code in Fig. 4 illustrates the major parts of the ray-casting loop for a given view ray. The ray-caster marches along the ray from sample to sample, performing hierarchical address translation for each sample as described above. The sample position on the ray is given by the corresponding normalized coordinate $\mathbf{p}$ in virtual volume space. In order to obtain a virtual multi-resolution address $(\ell, \mathbf{p})$, a suitable resolution level ℓ must be chosen. A good choice of ℓ depends on the desired level of detail, i.e., the resolution level to be sampled.

Computing a floating-point level of detail value $lod(\mathbf{p})$ for each sample can be done using existing strategies, e.g., estimating the projected screen space size of the corresponding voxel [6]. The integer resolution level ℓ can then either be determined from $lod(\mathbf{p})$ by rounding to the nearest integer, and sampling a single resolution level with tri-linear interpolation, or computing the two adjacent resolution levels, sampling both, and interpolating linearly in-between.

4.2.1 Exploiting Spatial Coherence for Page Table Look-Ups

Many successive samples along a ray will map to the same page directory/table entries. The corresponding look-up overhead can therefore be reduced significantly by exploiting this spatial coherence. The closer a page table entry is to the root of the hierarchy (the page directory), the less frequently it needs to be fetched. For example, for $b_{pt} = b_{vox} = 32$ and $t = 2$, for an axis-aligned ray the page table is accessed only every 32 voxels, the page directory every 1024 voxels.

The ray-casting loop tracks the indices of the last used page directory/table entries. When the next sample along a ray maps to the same entry, the result of the look-up from the previous sample is re-used instead of fetching the entry again (see Fig. 4). Figures 6 and 7 illustrate that this simple optimization in practice indeed reduces the required number of page table accesses per ray to a very small amount.

4.2.2 Cache Miss and Usage Reporting

Cache misses are reported during ray-casting for *unmapped* page table entries. A cache miss simply consists of a block ID that uniquely identifies the voxel block that caused the miss. Depending on the desired maximum size of resolution level $\ell = 0$, the block ID is either a 32-bit or a 64-bit integer. For small blocks of 32^3 voxels, 32-bit IDs are sufficient to address up to 128 teravoxels. Using 64-bit integers enables scaling up to the exascale. In our implementation, every ray tracks cache misses from front to back along the ray only up to a limit of M cache misses (e.g., $M = 4$). Cache misses farther back the ray are simply not reported in the same frame. This strategy not only makes cache miss reporting scalable, but it also distributes cache misses and the corresponding updates over multiple frames, which overall leads to smoother frame rates. As soon as arriving block data and the corresponding page table update fulfill a cache miss, no further miss will be reported for this block, and the next ray-casting pass will report cache misses farther back. This strategy also ensures that blocks will be updated in approximately front to back order, which makes the latency until data arrives less noticeable to the user. The list of cache misses is cleared before every frame, which prevents data requests from becoming stale when the visibility of blocks changes.

Cache miss hash tables. Each ray stores cache miss block IDs into a hash table shared with other rays, using atomic writes. We allocate one hash table for each pixel tile of size $N \times N$ in screen space (e.g., $N = 64$). Each hash table is implemented as an array with a fixed maximum number of entries per row. When a target row is already full, the cache miss will simply be dropped. As before, this distributes cache misses over multiple frames. Dropped misses will simply be reported in a later frame if the corresponding block is still visible.

Cache usage reporting. In order to allow the LRU caching strategy to track which cache blocks are in use, the ray-caster also tracks an *in use* bit for every block in the block cache. The required number of bits is equal to the number of available blocks in the block cache, which is independent of the volume resolution and thus scalable.

Per-frame read-back. The cache miss hash tables and the cache usage are read back from the GPU to the CPU once per rendering frame. Due to the comparatively small size, the corresponding performance impact is very small. Our current implementation reads back 300 KB of state for a screen resolution of 1024×1024 and a block cache size of 1GB, which takes less than 1ms on our test system.

4.2.3 Dynamic Resolution Hierarchy Traversal

Our system targets displaying volume data at exactly the desired level of detail, with respect to the display resolution, instead of substituting data of lower resolution. This goal is facilitated by being able to access any virtual multi-resolution volume address $(\ell, \mathbf{p})$ directly. However, in the following cases, we adapt the choice of ℓ dynamically.

Handling *unmapped* blocks. When an address $(\ell, \mathbf{p})$ is accessed whose corresponding page table entry is *unmapped*, we employ one of two user-selectable strategies: (1) The block is simply skipped as though it would be empty. It will be displayed as soon as the corresponding data have been constructed and the block has been mapped. (2) The resolution level ℓ is increased dynamically, in turn testing each successively lower resolution whether the corresponding block is mapped. The maximum number of steps for this strategy is limited to a small number k, e.g., $k = 4$. The rationale for this is that higher resolution reductions than typically 2^k are of limited benefit to the user. This scheme greatly improves the user experience, especially during zoom-ins, whereas the performance impact is only small due to exploiting spatial coherence in the same fashion as described before. That is, the reduced level $\ell + i$ with $i \leq k$ will be re-used for all subsequent samples that map to the same page table entry.

Exploiting *empty* blocks. For empty blocks, we maximize the amount of empty space that can be skipped with a single step via the following strategy. If the page table entry corresponding to $(\ell, \mathbf{p})$ is *empty*, the ray-caster also iteratively accesses coarser levels $(\ell + i, \mathbf{p})$, starting with $i = 1$. If the block of $\ell + i$ is also *empty*, i is increased further. The amount of empty space that can be skipped is then determined by $\ell + i - 1$, which is the largest block surrounding $(\ell, \mathbf{p})$ that is known to be empty. The performance overhead of this strategy is at most one unnecessary increase of ℓ, because all other increases are guaranteed to improve performance by skipping larger empty areas.

Dynamic data streaming. In order to handle dynamically streaming data, the raw tile processing stage notifies all later stages whenever new data have arrived, or data have been modified, which includes the change of alignment matrices. The visualization stage then unmaps the corresponding blocks if they are mapped. They will then simply be requested again, which ensures that they contain the correct data.

5 Visualization-Driven Volume Data Construction

The main responsibility of this pipeline stage (Fig. 2, center) is the *visualization-driven* construction of 3D volume blocks via on-the-fly stitching and resampling when the visualization stage requests them. These requests originate from cache misses during ray-casting that cannot be fulfilled from the caches in the visualization stage itself (Section 4.1.4). The visualization-driven design of this stage is a crucial property of our system that: (1) constructs 3D volume blocks only when they are visible, and (2) directly constructs 3D blocks only at the requested resolution. Instead of performing stitching and resampling at a higher resolution and then down-sampling to a lower resolution, we resample directly to the grid and resolution of the requested 3D volume block (Section 5.2). This approach is possible because our virtual memory architecture does not require lower (or higher) resolutions to be available (Fig. 3)—in contrast to octree-based systems [6, 7].

Efficient resampling of the 2D image tiles stored in the visualization archive is facilitated by retrieving only the sub-tiles intersecting

the 3D target block, from a resolution level matching the target resampling resolution (Fig. 5). A request for volume data is fulfilled in four main steps: (1) all 2D sub-tiles that intersect the 3D target block are determined. (2) for each sub-tile, the mipmap level matching the desired resampling resolution is chosen. (3) for each 2D slice comprising the 3D target block, the corresponding 2D sub-tile images are resampled directly into the corresponding location in the 3D target grid. (4) the resampled 3D block is transmitted to the visualization stage.

5.1 2D Sub-Tile Fetching

In order to quickly determine the 2D sub-tiles that intersect the 3D target block, we maintain an index structure to find all 2D image tiles comprising a z-slice in the final volume in $\mathcal{O}(\log_2 N)$, where N is the number of slices. The problem of finding all 2D tiles intersecting the 3D target block, and then all sub-tiles, is thus reduced to a 2D problem.

Efficiently retrieving the sub-tiles overlapping the 3D target block requires a compact index structure that easily fits into main memory and can still be searched efficiently. Since some of the 2D sub-tiles are potentially empty, we first compute 2D coordinates per sub-tile. Then we traverse these sub-tile coordinates in Morton order (z order) [25], and perform a run-length encoding on the indices, thus exploiting spatial coherence. Since only entry- and exit-points of a space-filling curve into a contiguous area are stored, the memory consumption of this data structure is only $\mathcal{O}(\sqrt{k})$, for k sub-tiles per tile. Retrieving a specific set of sub-tile indices is then implemented efficiently by first expanding indices by skipping non-contributing runs, followed by a single read from the visualization archive for all relevant sub-tiles.

The index structure is updated via notifications sent by the raw tile processing stage (Section 6). In case of an update event, the associated alignment matrix is read, the associated bounding rectangle is computed, and the index structure is updated accordingly. This does not require actual image data. These are only fetched when the visualization requests a 3D block. The same notification mechanism is also used whenever the alignment matrix of an image tile has changed.

5.2 3D Block Stitching and Resampling

Stitching and resampling are performed directly on the 3D target grid of the requested block. In order to avoid aliasing when resampling, we have to apply a 3D pre-filter [12]. Because of the 2D nature of our source data, we consider a separable 3D filter and convolution over a neighborhood $\mathbf{x}_i$ of a voxel $\mathbf{x}$, using a tensor product kernel $W_{x,y} \otimes W_z$ consisting of a 2D filter $W_{x,y}$ in the (x,y) plane, and a 1D filter W_z along the z axis. We define the pre-filtered 3D volume I^{3d} as:

$$I^{3d}(\mathbf{x}) = \sum_i W_{x,y} \otimes W_z\,(\mathbf{x}-\mathbf{x}_i) \cdot S_{2d}^{3d}\left((\tilde{T}_1, \mathbf{I}_1^{2d}), \ldots, (\tilde{T}_N, \mathbf{I}_N^{2d})\right)(\mathbf{x}_i). \quad (1)$$

The operator S_{2d}^{3d} performs stitching of the N required 2D source image tiles $\mathbf{I}_k^{2d}$ into the 3D grid of I^{3d}, where the pre-filter is then evaluated. This implicitly includes the reconstruction filter necessary for resampling at an arbitrary position $\mathbf{x}_i$. The $\tilde{T}_k$ perform the affine transformations for tile alignment. In our current implementation, the operator S_{2d}^{3d} chooses pixels according to their Euclidean distance to the center of the corresponding image tile (see below). We enable this in Eq. (1) by using 2-vector images $\mathbf{I}_k^{2d}$, which are built from the scalar source images I_k^{2d} as $\mathbf{I}_k^{2d}(\mathbf{x}) = \left(I_k^{2d}(\mathbf{x}), D_k^{2d}(\mathbf{x})\right)$, where $D_k^{2d}(\mathbf{x})$ is the distance between $\mathbf{x}$ and the center of tile I_k^{2d}.

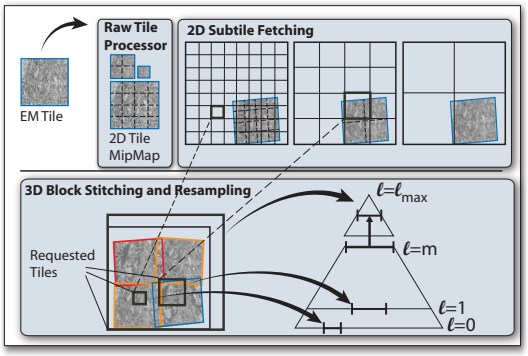

Fig. 5. **Visualization-driven stitching and resampling.** Only visible 3D blocks in the virtual multi-resolution volume are stitched and resampled, computing the result at the requested resolution ($\ell \in [0,m]$). Only the few coarsest resolutions ($\ell > m$) are down-sampled after stitching level $\ell = m$.

In order to make stitching followed by pre-filtering fast enough for visualization-driven processing, we split up Eq. (1) into a full-resolution pre-processing step for the pre-filter $W_{x,y}$, and a visualization-driven step for evaluation of S at the requested target resolution. Instead of applying $W_{x,y}$ in the volume construction stage, we apply it to each source image tile in the raw tile processing stage to compute the levels of a 2D mipmap, which is then stored in the visualization archive. We compute these pre-filtered 2D images $\tilde{I}_k^{2d}$ as:

$$\tilde{I}_k^{2d}(\mathbf{x}) = \sum_i W_{x,y}\,(\mathbf{x}-\mathbf{x}_i)\, I_k^{2d}(\mathbf{x}_i). \quad (2)$$

The volume construction step then only has to perform stitching of pre-filtered image tiles, followed by application of the 1D pre-filter W_z:

$$I^{3d}(\mathbf{x}) = \sum_i W_z\,(\pi_z(\mathbf{x}-\mathbf{x}_i)) \cdot S_{2d}^{3d}\left((\tilde{T}_1, \tilde{\mathbf{I}}_1^{2d}), \ldots, (\tilde{T}_N, \tilde{\mathbf{I}}_N^{2d})\right)(\mathbf{x}_i), \quad (3)$$

where π_z means projection onto the z-coordinate and the $\mathbf{x}_i$ now range over a 1D neighborhood of $\mathbf{x}$ in z-direction. $\tilde{\mathbf{I}}_k^{2d}$ is computed from $\tilde{I}_k^{2d}$ as described above. This approach only approximates the result of Eq. (1), because Eq. (2) applies the pre-filter $W_{x,y}$ before stitching is performed in Eq. (3), instead of afterward. However, it decouples image stitching from pre-filtering and sub-sampling input image tiles.

In principle, an arbitrary filter W_z can be used as long as one is willing to perform 2D stitching of all required input slices followed by 1D pre-filtering. Our current implementation uses a box filter of size one for W_z, i.e., nearest-neighbor interpolation along the z axis in Eq. (3). This allows a 3D target block to be resampled by simply stitching the image sub-tiles in 2D without performing actual 3D filtering, and storing the result into the correct 3D location. However, the general formulation above allows additional pre-filters to be integrated, which could even include the interpolation of intermediate slices in order to reduce the quality degradation caused by highly anisotropic data.

Implementation. In order to avoid inefficient disk accesses of small size, we request and construct 3D blocks larger than 32^3 voxels, from which the actual 32^3 blocks are then extracted. We currently construct 3D blocks of size $512 \times 512 \times 32$ (see Sections 8.1.3 and 8.4.1).

Resampling is performed using texture mapping and fragment shaders. After fetching all required sub-tiles from the correct resolution level of the 2D image tile mipmaps, each sub-tile is downloaded into a pool of 2D texture maps that is sampled by the fragment shader. As reconstruction filter we either use GPU bi-linear filtering, or a higher-order filter implemented in the fragment shader.

Stitching is performed during resampling using a GPU-based method akin to Hoff et al.'s Voronoi computation [14]. In regions where multiple tiles overlap, our domain experts prefer a selection of one actual measurement value over a blend between all valid values. A heuristic commonly used in this context is to select the pixel with the minimum distance to its respective tile center. We use the alignment matrix as texture matrix and render all contributing sub-tiles to an off-screen buffer. Then we set the depth of each fragment to the distance from the respective tile's center and perform a depth test.

Stitched and resampled 3D blocks are then transmitted to the visualization stage, which fulfills its block request. If the volume construction and visualization stages are running on two separate nodes, the construction stage also caches resampled 3D blocks in memory, and optionally on fast local SSD storage as well (Fig. 2).

Strategy for coarse resolution levels. Although the resolutions of our target data result in deep resolution hierarchies (Table 1), the tip of the hierarchy always corresponds to coarse resolutions. We define the corresponding resolution levels as $\ell \in [m, \ell_{max}]$, as depicted in Fig. 5. We currently choose m such that it corresponds to the resolution of one sub-tile times the number of image tiles comprising the full volume. For all levels $\ell > m$, we then use a different strategy. In order to fulfill a 3D block request, we first compute the entire level $\ell = m$ via stitching and resampling as described above. The block for $\ell > m$ is then computed via simple down-sampling. The resulting hybrid strategy is

similar in spirit to clipmaps [29], where the coarsest mipmap levels are always kept in memory, whereas only a clipped area of each finer level is resident. However, our goal in this stage is dynamic stitching and resampling of multiple image tiles, not rendering a single clipmap.

6 Raw Image Tile Processing

Our pipeline generates 2D mipmaps for the image tiles emitted by the EM as soon as they arrive, by continuously polling the acquisition archive. Each image tile has an affine transformation matrix attached to it which corresponds to the movement of the EM. This matrix can be iteratively refined by an external registration process to reflect image tile alignment both in 2D and 3D. The image tiles are chopped into 128×128 sub-tiles. This allows for more efficient disk storage, as well as more efficient resampling in the volume construction stage (Section 5), since the image tiles emitted by the EM are rather large (Section 7). Sub-tiles are optionally compressed using JPEG at 2bpp and stored in the visualization archive. Furthermore, they inherit the transformation matrix from the image tile (stored only once), augmenting it by an offset. Note that this stage is completely independent of registration, which we regard as an external process, and subsequent image alignment and stitching. It can thus be fully interleaved with both dynamic registration and data acquisition by the EM. Only meta data are forwarded to the volume construction stage. Actual data are only transmitted in case the volume construction stage forwards an actual request from the visualization stage. Processing in this stage is fast enough to keep up with the acquisition rate of the EM (Section 8.4.2).

7 Image Data Acquisition

The image tiles acquired by the EM are stored on a shared file system archiving all acquired raw data. Our neuroscience collaborators use a Scanning EM (SEM) apparatus with the following specifications. Tiles comprising $12,000 \times 12,000$ luminance pixels at 8bit precision are emitted every 15 seconds ($\approx$10 Mpixels/s). The SEM traverses a section of the physical data sample in a scan-line fashion with 6%–15% overlap between tiles. Both positional and rotational jitter are artifacts of the apparatus. Positional jitter is on the order of a few dozens of pixels, while rotational jitter does not exceed $\pm 5^\circ$. Both can be safely assumed to be normal-distributed. The overlap between tiles is a requirement for the registration. In the future, our collaborators would like to reach a scanning speed of first 20, and then 40 Mpixels/s. The evaluation in Section 8.4.2 shows that our processing of raw image tiles (Section 6) matches even these expected future numbers.

For testing different parameters of our system, we have implemented an *EM simulator* to simulate continuous image acquisition without requiring access or changes to the microscope setup. The simulator reads already stitched and registered sections measured by the EM. From these sections, rotated and translated tiles with mutual overlap as per the above specifications are cut out and forwarded to subsequent stages of our pipeline. The EM Simulator also supports simulation of registration updates.

8 Evaluation, Comparison, and Discussion

This section evaluates our system and compares it to previous approaches. Our ray-caster uses a GLSL fragment shader where each fragment casts a single ray, together with the `NV_shader_buffer_store` extension for writing out cache misses and usage. Tile processing also uses CUDA, OpenMP, and multithreading. Our test setup are three 12-core dual-CPU 3GHz machines, 48GB, with NVIDIA Quadro 6000. The visualization stage and GUI were running on the first machine, the volume construction stage on the second one, and the raw tile processing stage, as well as the EM simulator, on the third one. The communication between stages is implemented via TCP/IP and Winsocks2, on a 1Gb network. The block cache was a 1GB texture, and the page table cache a 64MB texture.

Comparison to other systems. Our system is the first one that targets interactive volume visualization and constructs 3D blocks of volume data dynamically from high-resolution image streams. Our visualization stage is most similar to the following two previous systems: The *Gigavoxels* system of Crassin et al. [6] performs ray-guided streaming of voxel data, which only downloads voxel blocks to the GPU when they become visible. However, these blocks have to come from a pre-computed octree. Its main target are entertainment applications, for rendering opaque (voxelized) surfaces or pre-defined opacity distributions such as static clouds. The interactive use of transfer functions is not supported, and empty space skipping is only possible for pre-determined homogeneous areas. The newer *Siemens CERA-TVR* system by Engel [7] uses an approach similar to [6], but supports dynamic transfer functions and the corresponding empty space skipping. It targets teravoxel volumes, but also requires a pre-computed octree and its efficiency decreases for anisotropic voxel aspect ratios. Both systems perform octree traversal during rendering, using the *kd-restart* algorithm [9]. In the following sections, we perform in-depth comparisons of our system with octree traversal approaches, for which we have implemented both *kd-restart* and *kd-shortstack* [15]. This shows that our system scales much better to deep resolution hierarchies.

8.1 Scalability Analysis and Comparison

This section discusses and compares the major scalability aspects of our system: scalability of the representation itself, scalability of traversal during rendering, and scalability of cache updates and usage.

8.1.1 Scalability of Volume Representation

Table 1 illustrates the scalability of our multi-resolution virtual memory scheme. We first consider the *resolution hierarchy*. For a volume with (r_x, r_y, r_z) voxels, and relative voxel dimensions (s_x, s_y, s_z), for the following computations we define $r_v = \max_i(r_i s_i)$, to correctly take into account arbitrary voxel anisotropy. For a volume resulting from a microscope pixel resolution of 5nm and a slice thickness of 40nm, $s_x = s_y = 1$ and $s_z = 8$. We choose a voxel block size of b_{vox}, and compute $\ell_{max} = \lceil \log_2(r_v / b_{vox}) \rceil$. The *page table hierarchy* is determined by choosing a small number of hierarchy levels t, and a page table block size b_{pt}. The corresponding page directory size of resolution level ℓ then is: $r_{pd}(\ell) = \left\lceil \lceil r_v(\ell)/b_{vox} \rceil / {b_{pt}}^{t-1} \right\rceil$, where $r_v(\ell)$ is the size of resolution level ℓ. A choice of $t = 2$ or at most $t = 3$ is sufficient for extremely large volumes, due to the extreme logarithmic scaling (essentially base b_{pt}). This leads to easily manageable page directory sizes. Table 1 illustrates this scalability for example volume sizes and different choices of b_{vox}, b_{pt}, and t. Following the same argument, and the fact that the maximum number of voxels in the view frustum is independent of the full volume resolution, small *page table cache* sizes are sufficient as well. Our implementation can reference a visible frustum of $2,048^2 \times 16,384$ with a 64MB texture.

8.1.2 Scalability of Volume Traversal

In an octree-based scheme, locating a voxel in resolution level $\ell = 0$ requires $\mathcal{O}(\ell_{max})$ traversal time from the root, which is logarithmic in r_v. In contrast, in our scheme any resolution level ℓ can be accessed by going directly ($\mathcal{O}(1)$ time) to the page directory of ℓ, and traversing the page table hierarchy in $\mathcal{O}(t)$ time. Although this can also be considered to be logarithmic in r_v, since the choice of t depends logarithmically on the volume resolution, $t \ll \ell_{max}$. Moreover, since very small t are sufficient for extremely large volumes, we consider t to be a system constant that is chosen for the largest possible volume size. Page table hierarchy traversal time then becomes $\mathcal{O}(1)$. In practice, we keep the constant factor small by exploiting spatial coherence.

These differences in scalability also show up clearly in practice, see the view depicted in Fig. 6. Fig. 7(a) compares the behavior of octree traversal vs. our approach in practice for increasing visible resolution. For octree traversal, we compare against *kd-restart* [9], as well as *kd-shortstack* [15]. The former is used in the systems of [6, 7].

volume resolution	vol size	$\ell_{max}+1$	$t=2$		$t=3$	
			16	32	16	32
$32,768^2 \times 4,096$	4 TB	11	64	**32**	4	1
$120,000^2 \times 15,000$	196 TB	13	235	**118**	15	4
$512,000^2 \times 64,000$	15 PB	15	1,000	500	63	**16**
$2,000,000^2 \times 250,000$	888 PB	17	3,907	1,954	245	**62**

Table 1. **Scalability of the page table hierarchy.** Page directory resolutions resulting from $b_{vox} = 32$, voxel anisotropy 1:8, and choices of $b_{pt} = 16$ or 32, respectively. We suggest the highlighted configurations.

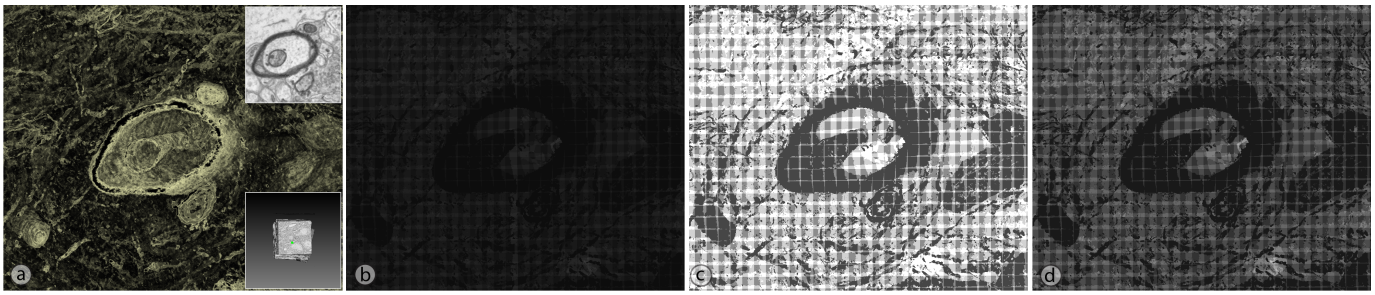

Fig. 6. **Hierarchy traversal complexity.** For the view ($\ell = 0$) in (a), image (b) encodes the number of page table entries accessed by each ray in the page table hierarchy ($t = 2$): *avg* $= 2.9$, *min* $= 2$, *max* $= 8$. Image (c) encodes the number of visited octree nodes for the same view using *kd-restart* [9] as used by [6, 7]: *avg* $= 21.9$, *min* $= 11$, *max* $= 77$. Image (d) uses *kd-shortstack* [15] with a stack depth of 4: *avg* $= 7.6$, *min* $= 4$, *max* $= 24$.

8.1.3 Scalability of Cache Updates and Usage

Figs. 7(b,c,d) illustrate and compare the number of required cache updates and the cache usage of our system vs. octree approaches. When the cache needs to be filled for a completely new view, cache usage and updates are the same. Our system requires *page table block updates* in the page table cache, *voxel block updates* in the block cache, and *3D block construction requests* issued to the volume construction stage (Section 5). Octree-based systems have to update the nodes in the *node pool*, and the voxel blocks (bricks) in the *brick pool* [6, 7]. Our system needs to perform a much smaller number of updates for each of these caches. For resolution level $\ell = 0$ and the view in Fig. 6, our approach updates 2146 voxel blocks, whereas the octree requires 2768 updates. This difference becomes much more critical when 3D blocks have to be constructed dynamically. For disk access efficiency, instead of requesting individual 32^3 voxel blocks, we request blocks of $512 \times 512 \times 32$ from the volume construction stage. The view in Fig. 6 requested 18 such 3D blocks with our approach, but 36 with an octree. Considering the time it takes to construct each block (Section 8.4.1), this constitutes a significant reduction in the latency perceived by the user. In the page table cache, our system requires only a few page table blocks (32^3) for this zoom-in. For level $\ell = 0$, we have to update between 1 and 4 blocks, whereas the octree approach must update around $5,000$ nodes. Although each octree node requires less storage [6, 7] than a page table block, updating the octree node pool with thousands of individual texture downloads constitutes a significant bottleneck.

8.2 Virtual Memory vs. Octree Traversal

Our approach and octree traversal differ significantly in the kind of volume data for which either of the two approaches is efficient. Here, we discuss these differences in theory. Section 8.3 and Table 2 then illustrate the corresponding frame rate differences for real data.

Octree traversal is efficient when: (1) Large subvolumes can be treated identically, i.e., they are either empty, homogeneous, or are rendered at a lower resolution. (2) The hierarchy is not too deep, i.e., ℓ_{max} is not too large. The main reason for this is that tree traversal always starts at the root ($\ell = \ell_{max}$) and goes down the tree to smaller ℓ. This means that rendering low resolutions is much more efficient than rendering high resolutions. This problem increases linearly with ℓ_{max}. For empty space skipping, this means that detecting large empty subvolumes is efficient, because they are detected with a few traversal steps. Conversely, detecting small empty subvolumes is inefficient.

Virtual memory access is most efficient when: (1) Only small subvolumes can be treated identically, i.e., data are either dense, or empty space forms only small clusters. (2) Data should be rendered at high resolution. (3) The hierarchy can be very deep (ℓ_{max} is large). This is true because any desired resolution level ℓ can be accessed directly in $\mathcal{O}(1)$ time, and ray-casting directly accesses the ℓ corresponding to the screen resolution. The efficiency of virtual memory access decreases when dynamic traversal of the resolution hierarchy is required (Section 4.2.3). Nevertheless, even the worst case of going all the way to ℓ_{max} is a $\mathcal{O}(\ell_{max})$ operation. However, octree traversal can be more efficient in this case due to a lower constant overhead per traversal step. In practice, however, reducing the resolution by too many levels is not useful for scientists. When the reduction in resolution is limited and ℓ_{max} is large, virtual memory access is more efficient than octree traversal even when dynamic resolution traversal is required.

8.3 Ray-Casting Performance

Our ray-casting performance primarily depends on the amount of visible data, which is mainly determined by the transfer function and the view. It does not depend significantly on the full volume resolution. Table 2 (column *'our'*) gives typical average frame rates for rendering $\ell = 0$ of microscopy volumes (D1-D3), as well as an industrial CT volume for comparison (D4), to a 1024×768 viewport. D1-D3 are very dense and do not allow or require a lot of empty space skipping. D4 was used in [7] and contains a lot of empty space. A typical view of D4 that we measured contained $22,058$ 32^3 voxel blocks, of which 61% were classified as empty. Skipping this empty space as described in Section 4.2.3 increased performance from 6fps to the listed 30fps (TF #2). Fig. 7(c) plots frame rates over an animation sequence of volume D1.

For comparison, column *'tree'* in Table 2 gives frame rates for rendering using *kd-restart* traversal as in [6, 7]. Our frame rates for D1-D3 are consistently better, as expected. For D4, tree traversal in principle has the empty space skipping advantages described above. However, in our comparisons our approach (Section 4.2.3) was still always faster. At least for this test volume, the empty areas are still not large enough for tree traversal to leverage its theoretical advantages.

8.4 Data Processing

This section evaluates the visualization-driven construction of 3D volume blocks, and the EM-driven processing of raw image tiles.

8.4.1 3D Block Construction

This stage comprises the following steps: (1) retrieve required subtiles from the visualization archive, (2) optional image decoding, (3) data download to the GPU for stitching and resampling, and (4) readback of the stitched data to the CPU. The optional JPEG encoder at a compression of 4:1 reads (and decodes) more than 300 Mpixels/sec from a standard hard disk. To measure the performance of this stage, we have issued a large number of requests for 512^2 cross-sections of the volume, varying the slice number requested to evaluate the 3D construction. We measured the minimum, average, and maximum latency in two scenarios. A worst-case scenario, for which we forced our implementation to execute steps 1–4 single-threaded and serialized to give an upper bound on the latency to be expected. The second scenario runs steps 1–3 interleaved. Our findings are summarized in Table 3. For the minimum latency, data has been cached by the OS

data set	resolution	size [GB]	TF	our	tree
D1: mouse cortex	$21,494 \times 25,790 \times 1,850$	955	#1	75	61
			#2	12	9
D2: hippocampus 1	$18,000 \times 18,000 \times 304$	92	#1	77	63
			#2	19	15
D3: hippocampus 2	$14,176 \times 10,592 \times 308$	43	#1	72	58
			#2	22	13
D4: rotation sensor	$2,048 \times 2,048 \times 2,048$	8	#1	55	44
			#2	30	25

Table 2. **Test volumes and frame rates [fps].** Frame rates are almost independent of resolution, depending more on the transfer function. TF #1: linear ramp. TF #2: semi-transparent (see Fig. 1). D4 only for reference.

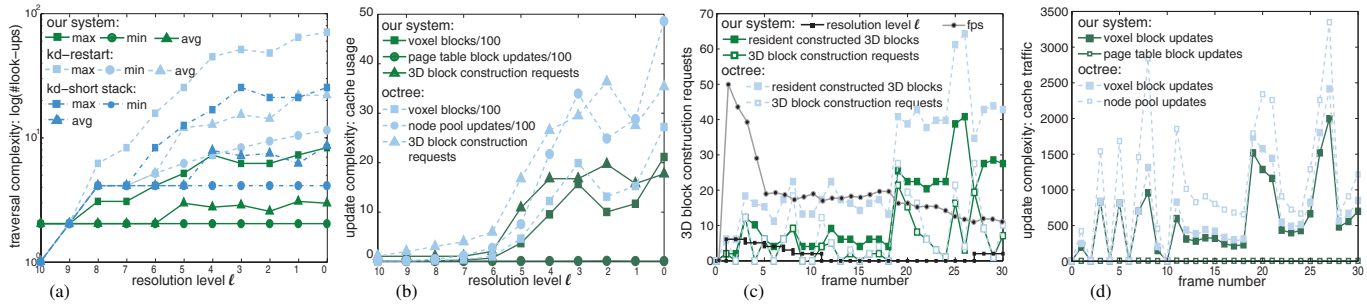

Fig. 7. **Hierarchy traversal complexity and cache update/usage evaluation and comparison.** (a) Log-scale traversal statistics (*max*, *min*, *avg*) over all rays in a 1024×768 image for a zoom-in from $\ell = 10$ to $\ell = 0$ (see Fig. 6): Page table hierarchy entries accessed ($t = 2$), vs. octree nodes visited (*kd-restart* compares with [6, 7]). (b) Cache update/usage for *voxel blocks*, *page table blocks*, and 3D blocks *requested for construction*, vs. the equivalent requests for an octree. Each resolution level was measured individually. (c,d): Statistics over a sequence of 30 animation frames with cache re-use from frame to frame: *Frame* 1–11: zoom-in; *frame* 12–27: pan. *Frame* 18: the transfer function changes to more transparency. Total for each frame vs. new to previous frame. (c) 3D block construction requests. (d) Voxel block and page table updates vs. node pool updates [6, 7].

and resides in CPU memory, while for the maximum latency data has to be retrieved first. With increasing level ℓ, data locality improves and therefore also the latency. Since the number of 128^2 sub-tiles required to form the construction varies due to the overlapping input data, so does the latency. We thus measure numbers normalized per sub-tile as follows: Optional JPEG-decoding takes 0.12ms, while transferring data to the GPU and rendering takes around 0.02ms per sub-tile. The GPU-to-CPU read-back is independent of the amount of sub-tiles rendered and takes around 0.68ms. The worst case latency (interleaved) to retrieve a fully constructed $512^2 \times 32$ block was around 1.7s (6.8ms / 32^3 voxels), while a block is typically constructed in less than 0.3s (1.2ms / 32^3 voxels). A fresh start from empty caches takes up to 15s.

8.4.2 Raw Image Tile Processing

The data set depicted in Figs. 1 and 6 (Table 2: D1) comprises 11,689 image tiles of size $12,000 \times 12,000$ in the acquisition archive. We used the EM simulator to generate these from $1,850$ slices of size $21,494 \times 25,790$ from the original data set. After raw image tile processing, this data occupies 210GB for voxel data and 13.3MB for index structures in our visualization archive. It is thus perfectly feasible to keep the index data structure resident in CPU memory even for data sets in the petabyte range. Further, our raw image tile processor achieves a sustained performance of 85 Mpixels/sec. It is thus significantly faster than the current measurement apparatus (yielding 10 Mpixels/sec [4]) and is able to keep pace with the future plans of our neuroscience collaborators (40 Mpixels/sec), as well as our current implementation of the EM Simulator, which achieves a sustained throughput of 45 Mpixels/sec on our test machine. All disk transfer is included in the performance reported. The start-up latency for both the simulator and the raw image tile processor is on the order of a few seconds.

8.5 Discussion and Limitations

Our system strives to enable neuroscientists to navigate and explore petascale EM volumes. Instead of thinking about the system as a "typical" volume renderer, we view it as being more analogous to a Google maps approach, albeit for dense 3D volumes instead of for 2D maps.

Limitations. Our system targets always rendering data corresponding to the full screen resolution. This approach incurs two main drawbacks. The first one is the *latency* until a complete image of the desired data resolution is generated. The rendering frame rate, as given in Table 2, is completely decoupled from the time it takes until missing data have been constructed and downloaded into the GPU cache textures. When all visible data have arrived, a correct image is rendered at high frame rates. However, this does not consider how long it takes until a correct image is visible. Overall latency varies significantly, and ultimately depends on the number of new 3D blocks that must be constructed for a new frame in addition to already cached data. Our system design strives to minimize this number and thus the latency, as illustrated in Fig. 7(c), and often requires fewer—and never more—blocks to be constructed than when tree traversal is used. Nevertheless, the worst case latency (Section 8.4.1) multiplied by the number of required 3D blocks (Fig. 7(c)) can be several seconds until a fully correct image is rendered when all caches are empty. Missing blocks or blocks rendered with lower resolution can optionally be color-coded for visual feedback on the current correctness of the rendered image.

ℓ	serial performance			interleaved performance		
	min	avg	max	min	avg	max
0	5.50ms	9.69ms	59.82ms	4.41ms	9.42ms	54.01ms
1	5.80ms	9.70ms	45.56ms	4.57ms	8.88ms	40.01ms
2	6.49ms	9.32ms	30.59ms	5.13ms	6.83ms	25.73ms
3	5.63ms	7.12ms	14.25ms	4.58ms	4.58ms	9.72ms

Table 3. **Times for stitching and resampling** to construct a 512×512 cross-section from a given resolution level ℓ of volume D1 (Table 2).

The second drawback is that our system assumes that the current working set, i.e., all visible data of the desired resolution, always fits into the block cache. If this is not the case, a correct image cannot be rendered. This problem can be handled by using distributed rendering on multiple GPUs [1], or by performing multiple rendering passes [7].

Both of these drawbacks can be mitigated by using a global *level of detail bias* to force lower-resolution rendering. This is also supported by our system, but has not been used for the results reported here. Our design goal was to avoid reducing rendering quality for interactivity.

Volume rendering of EM data. Our data are very dense and noisy, as well as highly anisotropic. They thus do not easily facilitate high-quality volume rendering, hindering the perception of connected structures in 3D. Tackling this problem is outside the scope of this paper. However, neuroscientists at minimum require rendering arbitrarily oriented slicing planes whose position and orientation can be changed interactively, and the infrastructure required for this is almost the same as for volume rendering, using a fully opaque "transfer function".

9 Conclusions and Future Work

We have illustrated that the two major design choices of our system for the first time enable the interactive exploration of petascale electron microscopy data streams via: (1) *visualization-driven* 3D data construction, and (2) a novel virtual memory scheme. The former *decouples* the multi-resolution hierarchy required for visualization from data acquisition. This decoupling is crucial to achieving scalability to the petascale for our target data streams. The latter *decouples* the resolution hierarchy from the hierarchy for volume sampling during ray-casting. This enables scaling to dense, anisotropic petascale volumes that cannot be handled by previous systems. The latter have to traverse the resolution hierarchy in logarithmic time, which becomes a significant bottleneck for deep hierarchies. We essentially have different goals than previous systems: instead of favoring lower fall-back resolutions and large clusters of empty space, we directly access any target resolution in $\mathcal{O}(1)$ time. Data of (limited) lower resolution are only used as a fall-back when the desired resolution is not yet available, but searching from high to low resolutions instead of vice versa.

In the future, we want to fully integrate our pipeline with the microscopy setup of our collaborators at the Harvard Center for Brain Science. We are also working on integrating distributed rendering into our system [1], which we consider to be orthogonal to the approaches presented here. Finally, upcoming GPU hardware features, such as *partially resident textures* [2] might allow tighter integration of our system with GPU memory management.

ACKNOWLEDGMENTS

We would like to thank Jens Schneider and Thomas Theussl for their contributions, Klaus Engel and Florian Link for their valuable input, and our collaborators at the Harvard Center for Brain Science. The project was partially supported by the Vienna Science and Technology Fund (WWTF) through project ICT08-040, the Intel Science and Technology Center in Visual Computing, Google, and Nvidia. Data set D4 courtesy of Siemens Healthcare, Components and Vacuum Technology, Imaging Solutions. Data was reconstructed by the Siemens OEM reconstruction API CERA TXR (Theoretically Exact Reconstruction).

REFERENCES

[1] J. Beyer, M. Hadwiger, J. Schneider, W.-K. Jeong, and H. Pfister. Distributed terascale volume visualization using distributed shared virtual memory. In *Posters at Large-Data Analysis and Visualization 2011*, 2011.

[2] B. Bilodeau, G. Sellers, and K. Hillesland. AMD GPU technical publications: Partially resident textures (PRT) in the Graphics Core Next. `http://developer.amd.com/documentation/presentations/GPUTechnicalPublications/`, 2012. Accessed on 31/03/2012.

[3] I. Boada, I. Navazo, and R. Scopigno. Multiresolution Volume Visualization with a Texture-Based Octree. *The Visual Computer*, 17:185–197, 2001.

[4] D. Bock, W.-C. Lee, A. Kerlin, M. Andermann, G. Hood, A. Wetzel, S. Yurgenson, E. Soucy, H. S. Kim, and R. C. Reid. Network anatomy and in vivo physiology of visual cortical neurons. *Nature*, 471(7337):177–182, 2011.

[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 153–162, 2006.

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

[7] K. Engel. CERA-TVR: A framework for interactive high-quality teravoxel volume visualization on standard PCs. In *Posters at Large-Data Analysis and Visualization 2011*, 2011.

[8] T. Fogal, H. Childs, S. Shankar, J. Krüger, R. D. Bergeron, and P. Hatcher. Large data visualization on distributed memory multi-GPU clusters. In *Proceedings of High Performance Graphics 2010*, pages 57–66, 2010.

[9] T. Foley and J. Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *Proceedings of Graphics Hardware 2005*, pages 15–22, 2005.

[10] E. Gobbetti, F. Marton, and J. Guitan. A single-pass gpu ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer*, 24(7):797–806, 2008.

[11] S. Guthe and W. Strasser. Advanced Techniques for High-Quality Multi-Resolution Volume Rendering. *Computers & Graphics*, 28:51–58, 2004.

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

[13] J. L. Hennessey and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, fifth edition, 2011.

[14] K. E. Hoff, J. Keyser, M. Lin, D. Manocha, and T. Culver. Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proceedings of SIGGRAPH*, pages 277–286, 1999.

[15] D. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive k-d tree gpu raytracing. In *Proceedings of Interactive 3D Graphics and Games 2007*, 2007.

[16] M. Howison, E. W. Bethel, and H. Childs. MPI-hybrid Parallelism for Volume Rendering on Large, Multi-core Systems. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, pages 1–10, 2010.

[17] D. M. Hughes and I. S. Lim. Kd-jump: a path-preserving stackless traversal for faster isosurface raytracing on gpus. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1555–1562, 2009.

[18] M. Kraus and T. Ertl. Adaptive texture maps. In *Proceedings of Graphics Hardware 2002*, 2002.

[19] J. Krüger and T. Fogal. Tuvok - an architecture for large scale volume rendering. In *Proceedings of the 15th Vision, Modeling and Visualization Workshop 2010*, 2010.

[20] J. Krüger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proc. of IEEE Visualization*, pages 287–292, 2003.

[21] E. LaMar, B. Hamann, and K. Joy. Multiresolution Techniques for Interactive Texture-Based Volume Visualization. In *Proc. of IEEE Visualization*, pages 355–362, 1999.

[22] A. Lefohn, S. Sengupta, and J. Owens. Resolution-matched shadow maps. *ACM Transactions on Graphics*, 26(4):1–23, 2007.

[23] J. W. Lichtman and W. Denk. The big and the small: Challenges of imaging the brain's circuits. *Science*, 334(6056):618–623, 2011.

[24] T. Lindeberg and L. Bretzner. Real-time scale selection in hybrid multi-scale representations. Technical report, KTH (Royal Institute of Technology), 2003.

[25] G. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd., 1966.

[26] T. Peterka, H. Yu, R. Ross, and K.-L. Ma. Parallel volume rendering on the IBM Blue Gene/P. In *Eurographics/ACM SIGGRAPH Symposium on Parallel Graphics and Visualization*, 2008.

[27] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek. Stackless kd-tree traversal for high performance gpu ray tracing. *Computer Graphics Forum*, 26(3), 2007.

[28] B. Summa, G. Scorzelli, M. Jiang, P.-T. Bremer, and V. Pascucci. Interactive editing of massive imagery made simple: Turning Atlanta into Atlantis. *ACM Transactions on Graphics*, 30(2):7:1–7:13, 2010.

[29] C. C. Tanner, C. J. Migdal, and M. T. Jones. The clipmap: a virtual mipmap. In *Proceedings of SIGGRAPH '98*, pages 151–158. ACM, 1998.

[30] J. M. P. van Waveren. id tech 5 challenges: From texture virtualization to massive parallelization. Talk in Beyond Programmable Shading course, SIGGRAPH 2009, 2009.

[31] M. Weiler, R. Westermann, C. Hansen, K. Zimmerman, and T. Ertl. Level-Of-Detail Volume Rendering via 3D Textures. In *Proc. of IEEE Symposium on Volume Visualization*, pages 7–13, 2000.