Hybrid Data Visualization
Based On Depth Complexity Histogram Analysis

Abstract
In many cases, only the combination of geometric and volumetric data sets is able to describe a single phenomenon under observation when visualizing large and complex data. When semi-transparent geometry is present, correct rendering results require sorting of transparent structures. Additional complexity is introduced as the contributions from volumetric data have to be partitioned according to the geometric objects in the scene. The A-buffer, an enhanced framebuffer with additional per-pixel information, has previously been introduced to deal with the complexity caused by transparent objects. In this paper, we present an optimized rendering algorithm for hybrid volume-geometry data based on the A-buffer concept. We propose two novel components that adjust the memory utilization to the depth complexity of individual pixels on modern GPUs. The proposed components are compatible with commonly used A-buffer implementations and yield performance gains of up to eight times compared to existing approaches. We demonstrate the applicability of our approach and its performance with several examples from molecular biology, space weather, and medical visualization containing both, volumetric data and geometric structures.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism— I.3.8 [Computer Graphics]: Applications—

1. Introduction

With the widespread use of imaging technologies in data intensive research fields, visualization becomes an enabling technology that supports the exploration of acquired data sets. A recent trend is the multitude of data that arise not only in the form of multimodal volumetric data sets but also from geometric representations, which may be derived from the data or are acquired in different ways. Consequently, a challenge in visualization research is the efficiency and effectiveness at which this hybrid data can be rendered in a single setting. In many cases the integrated geometry is of similar or higher complexity than the volumetric data with which it should be combined. While modern volume rendering supports the inspection of otherwise dense data sets by facilitating semi-transparency, additional challenges arise when multiple volumes are combined with geometric data. Recent work suggests that geometric representations also benefit from semi-transparent properties [GRT13]. Nevertheless, when fusing these two types of data in a semi-transparent manner, current state-of-the-art algorithms do not support interactive exploration as complexity increases. Interactivity

is, however, an essential feature when exploring scientific data sets.

In this paper, we improve upon existing techniques to facilitate efficient visualization of hybrid data sets as they occur in today's imaging-dependent areas of science. The fundamental challenge, arising when blending multiple semi-transparent sources into a single image, is to ensure that the elements are blended in view-dependent front-to-back or back-to-front order. When fusing multiple volumes or including mesh geometry into the visualization pipeline, the sorting must be ensured along each viewing ray. This results in a performance bottleneck as well as in increased memory usage. As these shortcomings hinder interactive exploration of many hybrid data sets, we have analyzed several of these data sets from different real-world scenarios with the goal to develop an optimized rendering algorithm enabling interactive exploration. As a tool in the analysis we have used Depth Complexity Histograms (DCH) that correspond to the observed depth complexity across all pixels in the rendered image for a given scene and camera setting. The DCHs have been used to identify trends in the global distribution for scenes with hybrid data and to determine how these trends

relate to rendering complexity. Based on the similarities of the occurring distributions, we propose two novel components which we combine with existing techniques to form an optimized rendering algorithm for hybrid data. Both components target the utilization of memory at higher cache levels by minimizing the allocated arrays and by partitioning the depth sorting and recycling allocated memory. As a result, computational throughput is increased and the maximum supported depth complexity is less dependent on the amount of available local GPU memory. The components provide between three and eightfold performance increase compared to existing algorithms. These qualities enable the interactive visualization of large and complex hybrid data sets at interactive frame rates, which is one of the key ingredients for enabling scientific discoveries in data-intense sciences.

2. Related Work

Data fusion. One aspect of hybrid data rendering relates to fusion of multiple volumetric sources. Several publications have investigated how samples from multiple sources should be blended, including different levels of intermixing [CS99] as well as focus and context techniques [VG07]. A recent comparison of different intermixing schemes was presented by Schubert and Scholl [SS11].

Rendering multiple volumes with arbitrary placement, orientation, and resolution is often associated with computational costs for scene partitioning. To limit the per frame rendering cost, object space is often separated into convex regions homogeneously occupied by a fixed number of volumes. The convex regions are then sorted either using specialized data structures [GBKG04, LLHY09, LF09] or using depth peeling techniques [PHF07, RBE08]. Many of these approaches also utilize shader instantiation, similar to one of the components presented in this paper, but do not include solutions for inclusion of semi-transparent geometry.

Fusing opaque geometry with volumetric data is straightforward in most volume rendering pipelines and does not require a full scene partition [EHK*06]. Semi-transparent geometry, on the other hand, is more computationally demanding as the resulting object space partitioning is costly, particularly if the geometry is not closed or has concave features. Image space techniques have been used in such situations as these techniques are generally well adapted to inclusion of generic, semi-transparent geometry [BBPtHR08, KGB*09]. With these methods, entry and exit points for homogeneously occupied ray segments are extracted from the volume proxy geometry and scene partitioning is performed on a per-ray basis during rasterization. The advantages and disadvantages of the available image space approaches are directly related to which transparency technique they employ, i.e. depth peeling and A-buffers, which are discussed below.

Transparency rendering. To ensure correct blending order of multiple semi-transparent samples, Order Independent Transparency (OIT) algorithms have been an ongoing research topic for the last thirty years. Two of the most widespread approaches are depth peeling and A-buffers. Our algorithm follows the principles and global memory management of the A-buffer, which yields superior performance for scenes with high depth complexity [YHGT10, KKP*13].

The A-buffer concept was first introduced as an antialiasing method by Carpenter [Car84]. Later work [EP90, CICS05, BCL*07, MB07] included the first hardware adaptations of the A-buffer approach but limited the algorithms to a comparatively low number of samples or utilized presorting of primitives. Further improvements in graphics hardware led to A-buffer variants with support for higher depth complexities including paged variations of the algorithm [KGB*09, Cra10], Per-Pixel Linked Lists (PPLL) [YHGT10], and an alternative called Dynamic Fragment Buffers (DFB) [MCTB12]. We direct the interested reader to a survey provided by Maule *et al.* [MCTB11] for a comprehensive overview on raster-based transparency techniques. Both DFB and PPLL approaches will be described later in detail along with the straightforward Fixed Fragment Buffer (FFB) implementation [Cra10]. However, it should be noted here that the modern A-buffer literature focuses only on management of global GPU memory while employing mostly straightforward techniques for managing local GPU caches. Our work builds upon these methods by improving the management of local GPU caches and hence improves performance for a wide range of A-buffer implementations.

Depth peeling was first introduced by Mammen [Mam89] and has evolved to employ multi-directional and bucketapproximations [Eve01, WGER05, CMM08, BM08, LHLW09]. Although our main algorithm is not based on depth peeling, the technique arguably remains a close competitor to A-buffers among OIT solutions. In addition, one of the components proposed in this paper shares high resemblance with the original depth peeling approach.

3. Visualization of Scenes with Non-uniform Depth Complexity

In this paper, we have chosen four representative scenes of different fields including molecular science (*protein*), space weather simulations (*space*), neurosurgical treatment planning (*medical*), and computational fluid dynamics (*cfd*). Characteristic for the selected scenes is that the major computational costs arise from the task of ordering all contributions in terms of visibility. In modern OIT solutions, these costs are directly related to the observed depth complexity in the image, i.e., the number of overlapping contributions per pixel. At the same time, most scenes often feature nonuniform complexity distributions in image space with clusters of high depth complexity surrounded by regions of lower complexity. To analyze the complexity of such scenes we thus utilize depth complexity histograms.

(b) geometric data

(c) volume data

(a) hybrid visualization

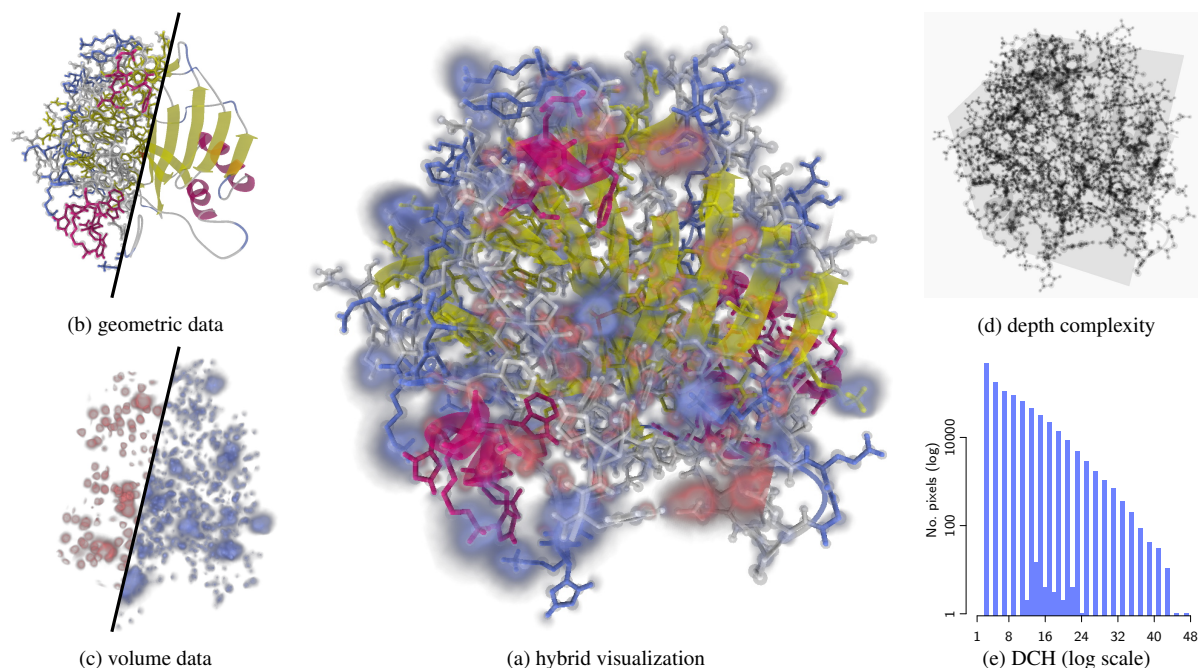(d) depth complexity

(e) DCH (log scale)

Figure 1: Visualization of a protein (*human carbonic anhydrase II*) combining hybrid data from different sources. (a) final rendering of hybrid visualization. (b) geometric protein representations. (c) volumetric data sources depicting electrostatic potentials. (d) image-space depth complexity (re-scaled for representational purposes). (e) depth complexity histogram (DCH, log scale). Our improved A-buffer algorithm is capable of rendering the entire scene at 32 fps—a performance increase of 5.9 times compared to prevalent techniques.

3.1. Depth Complexity Histograms

We use the term Depth Complexity Histogram (DCH) for a histogram that shows the distribution of depth complexities for all pixels given for a specific scene and camera setting. Each bin in a DCH is associated with a complexity range and holds the number of pixels whose depth complexities fall within that range (cf. Figure 1(e)). Computing a DCH is done by accumulating an integer "image" of the scene during a single rendering pass where each pixel value is treated as an atomic counter that is incremented once for each incoming fragment. The DCH can then be computed as a standard histogram over this integer image.

3.2. Visualization Challenges for Data Fusion

During the analysis of our four scenes, we made the observation that many pixels feature a low depth complexity and only a few pixels feature the highest depth complexities. This results in rapidly decreasing DCHs where the majority of pixels have a complexity that is only a fraction of the global maximum of the scene. In our experience, these findings are also applicable to many other scenes.

One example scene, the protein *human carbonic anhydrase II* (PDB-ID: 2CBA), is depicted in Figure 1. The scene

contains protein structures stored in polygonal meshes and volumetric data sources representing electron charge densities. Such protein visualizations are commonly used to examine possible docking positions between the protein and other molecules [SdG10].

From a depth complexity perspective, the protein scene is fairly well behaved. The maximum depth complexity is limited and the points with high depth complexity are evenly distributed across the entire scene. The DCH, however, decreases rapidly. Figure 1(e) shows a logarithmic plot of the DCH (the dominance of even numbered complexities is a result of closed geometry). The majority of all pixels (68%) have a depth complexity of 8 or less, whereas only very few pixels (6.6%) have a complexity larger than 17 with a maximum of 46 for this specific camera setting.

DCHs are not only dependent on scene content but also on the camera settings. Figure 2 illustrates how the DCH changes for the protein scene for an animated camera path. The time-dependent DCHs are shown as vertical bar-plots of screen coverage in percent (linear scale, right y-axis). Complexities have been binned according to the legend in the lower right and the DCH is plotted for every tenth time step. The camera path is illustrated with five snapshots taken at time points 50, 150, 250, 350, and 450 respectively (top row
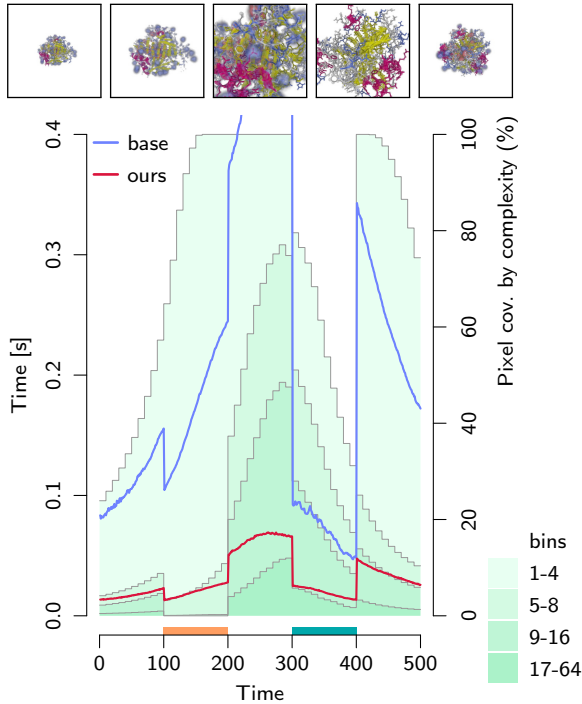
Figure 2: Performance and complexity over time for a pre-defined camera path around the protein data set of Figure 1. Rendering times are shown as lines (first y-axis) and DCHs as bar charts (second y-axis). The camera dollies in and out while the scene composition is changed every 100th step.
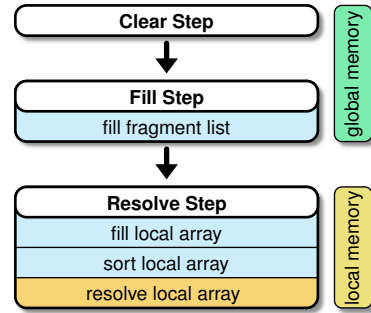

Figure 3: Overview of the A-buffer rendering approach on GPU hardware. Previous work has primarily focused on global memory management during the Fill Step. Our proposed algorithm is hence aimed at local GPU cache memory. Memory related steps are highlighted in blue.

of images). Additionally, the figure also shows performance (red and blue line plots, left y-axis) which will be discussed in later sections. In addition to camera movement, the scene composition is changed every 100th time-point. At the first marker (orange), the geometry of the stick representation is removed (cf. Figure 1(b)) reducing high-complexity pixels. At the second marker (green), both volume data sources are removed (cf. Figure 1(c)) reducing low-complexity pixels. From the plots, we can see that the DCH changes over time and that the rendering performance depend on these changes. We can also see that the DCH remains non-uniform in nature even with the camera very close to the protein. This opens up the possibility to save resources if the memory allocation could be adapted to the observed complexity on a per pixel basis. In the next section we provide a technical description of how current state-of-the-art OIT techniques adapt to non-uniform scene complexities.

4. Algorithms for Semi-Transparent Data Fusion

To construct an optimized algorithm we exploit the trend of non-uniform DCHs observed earlier. Three state-of-the-art implementations of the A-buffer concept serve as a reference. This section details the differences of those algorithms

in the dynamic management of global GPU memory. In addition, we show how to further optimize the algorithms with respect to their use of local caches on modern graphics hardware.

The principle of the A-buffer is to capture and store a list of rasterized fragments on a per-pixel basis (called *Fragment List*). This involves three sequential steps as illustrated in Figure 3: the Clear Step, the Fill Step, and the Resolve Step. Both the Fill Step and Resolve Step contain sub-steps associated with memory management (highlighted in blue). This includes global memory management mainly during the Fill Step and local cache management in the Resolve Step. Fragment data structure and further considerations on source types will be described in Section 6.

Managing Global GPU Memory

Three prevalent approaches for managing the global A-buffer memory during the Fill Step are used. These are Fixed Fragment Buffers (FFB) [Cra10], Dynamic Fragment Buffers (DFB) [MCTB12], and Per-Pixel Linked Lists (PPLL) [KGB*09, YHGT10, Cra10].

In Figure 4, the three approaches are illustrated side-by-side. In each approach, every pixel is associated with a fragment counter and an implicit or explicit pointer to the stored Fragment List. For the FFB, the global memory pool is statically partitioned and the pointer is implicitly given by the screen coordinates of the fragments. The global memory pool of the DFB is dynamically partitioned each frame and the explicit pointer holds a base offset into the pool. For the PPLL, the pointer denotes the list anchor for a linked list of fragments spread throughout the pool. Both, FFB and DFB ensure a contiguous memory layout for each individual pixel. The Fragment Lists are generally stored out-of-order in the comparatively slow but large global memory of the GPU.

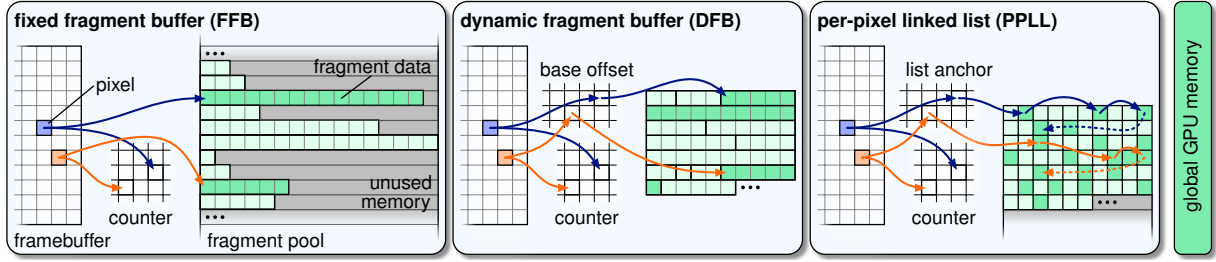DFB and PPLL are both dynamic algorithms and adapt

Figure 4: Comparison of three existing A-buffer implementations for managing global GPU memory. Both, the DFB and PPLL approaches adapt well to scenes with non-uniform depth complexities by allowing dynamic memory utilization.

the utilization of global memory well to non-uniform depth complexity distributions. Thus, consumption of global memory is almost optimal even for rapidly decreasing DCHs.

Managing Local GPU Caches

It is common practice to avoid the higher latency of global memory by copying the unsorted Fragment List to a local cache before sorting. An important aspect of such cache utilization is that the memory layout of modern GPUs typically exposes a limited shared cache to a group of cores. Additionally, modern GPUs hide global memory latency by hot swapping groups of threads in a manner similar to pipelining on a CPU [NVI11]. Under ideal circumstances, the number of active threads can eight times higher than the number of physical cores. Per thread allocations therefore need to be small enough such that all active threads designated to a group of cores can have their arrays allocated simultaneously. It is thus important to minimize the allocation of cache memory to maximize performance. Current trends also indicate that the number of cores increases faster than the available shared cache size, potentially escalating this problem in the future.

The prevalent approach described in the literature for managing local GPU caches is to allocate a fixed sized array, of size N per thread [Cra10, MCTB12]. This has two immediate consequences. First, the maximum depth complexity handled by the approach is limited to N, since all fragments beyond this number are discarded. Second, larger array sizes significantly reduce the number of active threads due to cache overflow and reduced hot swapping. A number often reported in the literature is $N = 64$. While the literature includes strategies, such as DFB and PPLL, for managing the global memory, there still remains room for improvement in managing local caches.

5. Improved Dynamic Depth Complexity Management

We propose two novel A-buffer components for A-buffer based algorithms based on the observed nature of the DCHs. Both components are designed to improve the management of local GPU caches. For clarity, we will use the term Local

Array to indicate the fast (local) memory allocated for the sorting of fragments.

The first component, illustrated in the center of Figure 5, is called per-pixel Array Optimization (ppAO) and ensures that all pixels are evaluated without allocating excessively large Local Arrays. The component segments image space into sections of similar depth complexity and separate shaders, compiled with different local arrays sizes, are triggered for each segment. The second component, illustrated to the right in Figure 5, is part of the Resolve Step and corresponds to a novel sorting procedure called per-pixel Depth Peeling (ppDP). The component breaks the sorting into smaller pieces by not loading all fragments at once and can thereby recycle memory.

A key factor of both components is that they are designed to be used in combination with the current state-of-the-art management of global memory described in the previous section. The full rendering algorithm, optimized for geometry intensive fusion scenes, is thus formed by combining the components proposed here with one of the existing solutions for global management, such as DFB or PPLL.

5.1. Dynamic Resource Management Using per-pixel Array Optimization

The objective of the proposed ppAO component is to limit the amount of unused memory in the local cache by performing the Resolve Step with Local Arrays that better correspond to the depth complexity of individual pixels. Since dynamic memory allocations are not possible at the highest cache levels of the GPU memory hierarchy, the optimization of per-pixel array sizes requires multiple shader programs to be instantiated with varying Local Array sizes. Rather than using a unique shader for every possible Local Array size we use a smaller set of pre-defined array sizes. This leads to a binning of the DCH, i.e., clustering of pixels with similar depth complexity into segments. All pixels in a segment are then resolved in a batch by a single shader. The full image is then processed in as many batches as there are segments. Note that the clustering is independent of the actual pixel locations and that each segment not necessarily corresponds to
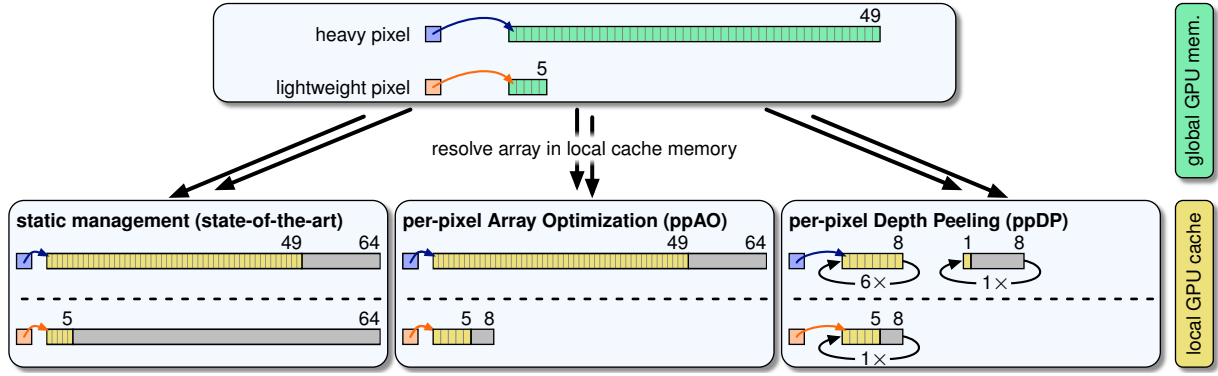
Figure 5: Comparison of approaches managing local GPU caches: static management (left) and our two proposed components ppAO (center) and ppDP (right). The unsorted input Fragment List (top) corresponds to the output of the Fill Step and resides in global memory. The array is sorted in local caches during the Resolve Step using predefined array sizes. Two pixels are exemplified, a heavy pixel (blue) and a light pixel (orange). The ppAO component significantly lowers the amount of unused memory (shown in gray), while the ppDP component lowers the amount of both unused and used memory by recycling it.

a focused region in the image. The ppAO is positioned after the Fill Step in the A-buffer algorithm (cf. Figure 3) making the ppAO responsible for managing and calling the Resolve Step on the respective complexity segments.

The instantiation procedure of the shader programs is straightforward as the individual instances only differ by a single number—the predetermined buffer size. It is also sufficient to instantiate only a small set of shaders due to the decreasing power curve observed in most DCHs. By analyzing the depth complexity of various scenes, we found that complexity segments of sizes 8, 16, 32, 64, ... give good results with respect to performance and memory consumption. The ppAO algorithm is outlined below.

```
// Fill Step
forall the pixels of the framebuffer do
    count depth complexity;
end
// Resolve Step, loop over all complexity segments
foreach i ∈ {8, 16, ..., max} do
    activate shader program for complexity segment i;
    foreach pixel p ∈ pixels do
        // mask pixel if not part of segment class i
        if depth complexity(p) = i then
            resolve segment with Local Array of size i;
        end
    end
end
```

To perform ppAO, we need to know the depth complexities of all pixels. For this purpose, we utilize a separate integer buffer the size of the framebuffer (one atomic counter per pixel) to count the number of fragments per pixel. The counting can be combined with the scene rasterization in the Fill Step and, hence, no additional rendering passes are required.

Once the depth complexities are known, the different complexity segments may be processed. This is done by looping over all pre-defined sizes of complexity segments and activating the shader program with the respective Local Array size. The clustering of pixels into complexity classes is handled implicitly during runtime by a masking step where all pixels that are not part of the current segment are discarded. Note that the graphics pipeline does not need to be flushed between triggering successive segments and that multiple segments may be evaluated in parallel (even if they are triggered sequentially). Figure 5, center, depicts the utilization of our approach for pixels exhibiting high and low depth complexity.

5.2. Preventing Over-sized Local Arrays Using per-pixel Depth Peeling

If the number of fragments inside a Fragment List exceeds the buffer size of the Local Array information is usually lost and the final rendering result might feature artifacts. One impractical solution is to increase the size of the Local Arrays. However, large parts of the pre-allocated memory will never be used since typically only a few pixels exhibit a high depth complexity (cf. Section 3.1 and Figure 5, left). To overcome this particular problem, we propose a variation of depth peeling on a per-pixel basis to provide a simple way to correctly deal with overflowing Local Arrays. Thus, the maximum supported depth complexity is no longer limited by the size of local memory but instead limited only by the amount of available global memory. In Figure 5, right, a Local Array size of 8 is used to illustrate our approach for two depth complexities.

The ppDP algorithm is outlined below and replaces the Resolve Step for individual pixels in Figure 3.

> **input** : list of fragment data per pixel
> $n \leftarrow$ sizeof(Local Array);
> create buffer with n elements;
> set min depth to 0;
> initialize current state;
> **repeat**
> > **foreach** *fragment $f \in$ fragment list* **do**
> > > **if** *depth(f) $\geq$ min depth* **then**
> > > > discard f in current pass;
> > >
> > > **else**
> > > > insert f into buffer enforcing sorting;
> > > > *// store at most n elements*
> > >
> > > **end**
> >
> > **end**
> > resolve buffer considering current state;
> > store min depth and state of last buffer element;
> > clear buffer;
>
> **until** *all fragments $f \in$ fragment list are resolved*;

The ppDP algorithm follows the approach of bucket depth peeling where a larger list is sorted and resolved as a set of smaller sequences but does so fully on the GPU between global and local memory without the need to re-render geometry. We split the Resolve Step into several subpasses which allows for the Local Array to be recycled and its size to be reduced. With a Local Array of size n, the process sorts $n-1$ elements in the Fragment List per subpass. While looping over the contents of the Fragment List, the n entries with the smallest depth values are chosen and stored in the buffer by using insertion sort. If a fragment was already processed it will be discarded in the current pass. After filling the buffer its contents are resolved. To ensure consistency between the individual resolve passes, we temporarily store the current state of the rendering, including volume occupancy and ray position. The current state and minimal depth are updated and the local buffer is emptied for the next pass. This approach does not require additional rendering passes but does require the entire Fragment List to be read multiple times from global GPU memory.

Sorting the full array thus takes a maximum of $\lceil D/(n-1) \rceil$ passes for a pixel with depth complexity D. The loop may be terminated sooner, e.g. in case of early-ray termination, and further read/write intensive sorting operations are avoided. Note that the size of the Local Array is constant. We observed an optimal Local Array size of 8 for our ppDP approach across all test setups with respect to performance.

Note that our ppDP algorithm may exhibit z-fighting issues similar to regular depth peeling. The issue arises when multiple fragments of a single pixel have identical depth and only a subset of these fragments gets included in one loop iteration. Since only the minimal depth is used to distinguish processed fragments from fragments yet to be resolved, the solution is not unique. This also complicates the decision in the subsequent iterations regarding which fragments should be discarded. So far in our work we have not experienced any artifacts so far from this kind of z-fighting. A potential solution to handle the problem is to introduce an additional flag per fragment entry whether it has been resolved. But this will also include an additional test thereby increasing the computational load.

6. Fused Rendering of Hybrid Data Sources

So far we have discussed the memory management of the A-buffer involved in the construction of the sorted Fragment List (Figure 3, blue highlights). In this section, we will focus on the A-buffer evaluation and the rendering of the scene (Figure 3, orange highlight) and implementation details including data structures.

6.1. Fragment List Creation and Evaluation

The following data structure is used for all fragments generated in the Fill Step as entries of the Fragment List: *z* (float32), *id* (int32), *color* (vec4 float16). The structure has a total size of 16 bytes and the same data structure is used for both geometric and volumetric contributions.

In the Fill Step, ABuffer_frag entries are computed and stored in the Fragment List. Geometric sources are shaded during this step and the computed color is stored explicitly in the *color* component. For volumetric sources, only the associated proxy geometry is rendered during the Fill Step and the *color* component is left uninitialized. The *z* and *id* components are treated identically for all source types and respectively hold the depth value in screen space coordinates and a unique source identifier. Bit operations are used on the *id* component to store both source type as well as a unique identifier. Sources may be rendered independently by different shaders but the produced fragments (instances of ABuffer_frag) are all inserted into the same global storage.

During the final stage of the Resolve Step, the scene content is evaluated and blended into the framebuffer. At this point, all entries in the Fragment List can be interpreted as intersection points along the view rays. Ray casting of the scene is performed in world space by sequentially looping over all entries in the Fragment List. Geometry fragments are blended directly to the buffer while volume rendering is performed for the interleaved ray segments. Volume occupancy is tracked through a bitmask which is updated every time a fragment of a volume proxy geometry is encountered. For more information on fragment list evaluation, particularly the use of bitmasks to track volume occupancy, we direct the reader to Brecheisen *et al.* [BBPtHR08]. Once the occupancy and entry and exit points are known, the problem is reduced to the problem of multi-volume rendering as discussed in Section 2.

6.2. Implementation Details

Our implementation supports global memory management using FFB, DFB, and PPLL as described in Section 4. The FFB and PPLL implementations are derived from the work by Crassin [Cra10]. The code for the DFB implementation was provided by Maule *et al.* [MCTB12] and was slightly changed to fit our needs. We use C++, OpenGL, and GLSL for our framework. The only exception is the scan step of the DFB implementation which is performed in CUDA using the CUDA Thrust library as explained by Maule *et al.*

For management of local cache memory we utilize our proposed components: ppAO and ppDP (cf. Section 5). The ppAO component is implemented using an 8bit stencil buffer for masking the different complexity segments to be triggered by separate shaders. The Fill Step is enclosed in a loop on the CPU, where different segments are triggered by modifying the OpenGL stencil function. Note that using an 8bit buffer does not limit the depth complexity to 255. Higher complexities are still possible but they will all be associated with the same segment. If larger segment limits are needed, the stencil buffer can easily be replaced with a texture of higher precision. The ppDP approach simply replaces the final steps of the Resolve Step and the loop is implemented directly in GLSL.

Local Arrays are always allocated as local buffers with predefined sizes inside the shaders and are, thus, not assigned explicitly to specific memory. Thread scheduling and memory assignment is therefore up to the discretion of the driver. The entire algorithm, with global and local optimizations, requires the implementation of three shaders namely clear, fill, resolve plus additional clones of the resolve shader containing different buffer sizes for the ppAO.

7. Results

We tested our proposed A-buffer components with four real-world cases from different fields. The performance was measured for a viewport size of 1024×768 on four different NVIDIA GeForce GTX GPUs: 560, 580, 670, and Titan (1 GiB, 1.5 GiB, 2 GiB, and 6 GiB VRAM respectively). Individual performance for the 580 and Titan are included in the paper (representing Fermi and Kepler architectures respectively). Performance for 560 and 670 are available as supplementary material. Reported averages were computed across all GPUs. Scene configurations will be described next in Section 7.1 followed by a presentation of performance in Section 7.2.

7.1. Scenario Descriptions

Figure 1, protein: The data corresponds to the protein *human carbonic anhydrase II*. The scene consists of four separate data sources; two volumetric data sets describing the electrostatic potential calculated at different resolutions and extents, a geometric stick representation of the protein, and a geometric ribbon model of the protein. Both geometric representations are colored according to the secondary structure type. The three-dimensional structure and potential fields are often visualized to examine potential docking positions between the protein and other molecules [SdG10].

Figure 8, space: The space data set depicts results of a multi-variate simulation of a time-dependent 3D magnetohydrodynamics system of the heliosphere during a coronal mass ejection event. This particular systems is currently used in space weather prediction [XOL04]. The scene consists of two volumes, showing the number of charged particles and the energy density, and three isosurfaces derived from the energy densities.

Figure 9, medical: The data, provided as the IEEE Visualization Contest in 2010, contains multiple medical imaging modalities as well as derived information sources for planning neurosurgical intervention [IEE10]. The scene consists of four data sources; two volumetric data sets depicting T1-weighted MR images of head and brain, two sources of geometric information in a surface extraction of a tumor segmentation and about 1678 fiber tracts obtained with diffusion tensor imaging. The combined information from all data sources is used to plan the safest possible access path for an intervention.
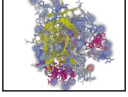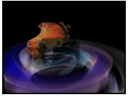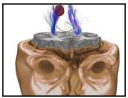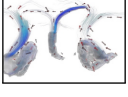
Figure 10, cfd: The data results from a computational fluid dynamics simulation of blood flow in the carotid artery of a human subject. The scene consists of a single computed tomography image and two sets of geometric primitives. A sparse glyph representation and more than 7500 individual streamlines.

More details on the different data sources of the individual scenes are listed in Table 1. Depth complexities are available as single view DCHs in Figures 1 (protein), 8 (space), 9 (medical), and 10 (cfd), respectively.

7.2. Performance Comparison

We have investigated how our proposed components improve the performance for each of the three A-buffer algorithms for global memory management. For each algorithm, we measured the performance for different combinations. In Figure 6, the results are shown from top to bottom for the four distinct scenes—protein, space, medical, and cfd. For each of FFB, DFB, and PPLL, we have measured the performance of four configurations for local memory management; baseline (static buffers), ppAO, ppDP, and ppAO+ppDP. The largest Local Array size allocated for the scenes was set to 64 (protein), 32 (space), and 128 (neuro and flow). Allocating a Local Array size to hold all depth layers of the medical scene resulted in driver crashes, thus, only the first 128 layers were resolved for baseline and ppAO configurations (ppDP is capable of resolving higher complexities with smaller Local Arrays). Likewise, using FFB for global management also limits maximum complexity, requiring ap-

Table 1: Data information for the four selected scenes, including the first frame of the sequence used for performance tests.

Scene	Data	Data size	1st Frame
protein	VOL	$127 \times 127 \times 127$	
	VOL	$71 \times 71 \times 71$	
	GEO	868k triangles	
	GEO	36k triangles	
space	VOL	$256 \times 256 \times 256$	
	VOL	$256 \times 256 \times 256$	
	GEO	162k triangles	
	GEO	100k triangles	
	GEO	67k triangles	
medical	VOL	$415 \times 487 \times 176$	
	VOL	$367 \times 395 \times 150$	
	GEO	$363k$ triangles	
	GEO	1678 fibers	
cfd	VOL	$76 \times 49 \times 45$	
	GEO	7.5k streamlines	
	GEO	≈ 100 arrows	

proximately 12.6 MiB per depth layer at 1024×768 resolution. Depth segments for ppAO were chosen as powers of two between 8 and the scene cap reported above. All performance results for ppDP were measured with a Local Array size set to 8. For each test, frame times were computed as the average over 30 seconds of rendering as the camera rotated around the scene at fixed distance. Early-ray termination was enabled for transparency values less than 0.02 during all benchmarks.

Due to a major performance penalty, we were unable to achieve results for the DFB on the same level as reported by Maule [MCTB12]. The penalty manifested as a 50 ms–200 ms delay associated with the context switch between CUDA Thrust and OpenGL that is required by the scan step. It was persistent across different GPUs and drivers and was also present in a minimal stand-alone DFB implementation. The results for the DFB approach reported in Figure 6 include this penalty.

Supported by the performance graphs, we can say that both ppAO and ppDP are capable of providing significant performance gains for a large variety of configurations. The performance gain is threefold for ppAO and about eightfold ppDP when averaged over all scenes and GPUs. For scenes with high depth complexities or homogeneous depth complexity distributions, the improvement obtained with ppAO is reduced since the processing of the heavy pixels becomes a bottleneck. In such scenarios, ppDP maintains a significant speedup, particularly when early-ray termination is enabled.
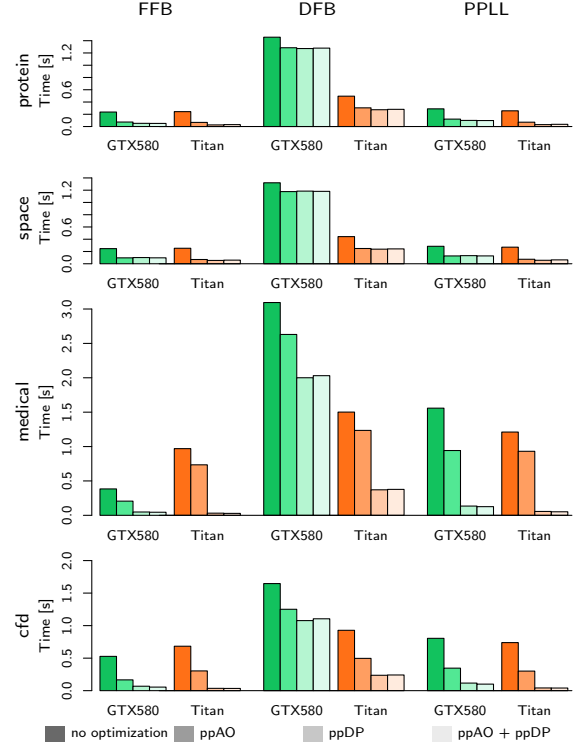
Figure 6: Performance comparison with our proposed ppAO and ppDP components. Columns represent the different A-buffer implementations in combination with our components (none, ppAO, ppDP, ppAO and ppDP combined).

In addition to the four real-world scenarios, we created a synthetic worst-case scene with screen-filling quads. The scene consists of 64 quads ($\alpha = 0.05$) aligned orthogonal to the viewing direction where their order is randomized in depth. The quads are scaled so that they all cover the entire viewport in the middle of the animation. Thus, all pixels feature a near-maximum depth complexity. Results are shown in Figure 7 and confirm that the performance of ppAO (as expected) approaches that of statically implemented buffers while ppDP maintains a 3.6 times performance increase.

8. Conclusions and Future Work

In this paper, we address the challenge of interactive exploration of complex hybrid data. Two adaptive data handling components are proposed to be combined with prevalent A-buffer techniques to form a final rendering algorithm. Both components are based on the observations we made when analyzing the histogram of the respective depth complexities of hybrid data sets as acquired in modern imaging-based sciences. Of the two, ppDP is better equipped to deal with very high complexities but does so at the cost of potential artifacts from z-fighting while ppAO performs well
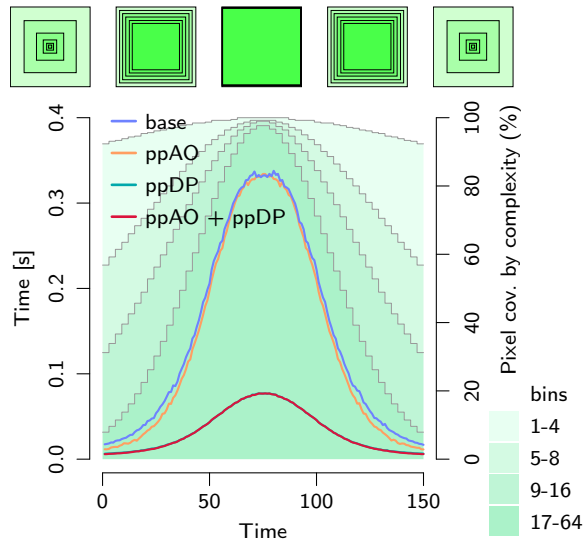
Figure 7: *Performance for a synthetic data set. Rendering times are shown as lines (first y-axis) and DCHs as bar charts (second y-axis).*

for moderate complexities with exact results. Optionally, the components can be combined to achieve support for high depth complexities while guaranteeing a minimum number of correctly blended samples.

An interesting consequence of our work is the potential alleviation of the traditional trade-off between performance and depth-support. We believe that this aspect makes our work valuable also in other areas of visualization or computer graphics that employ A-buffer based techniques.

In the future, we see several opportunities to improve the proposed methods. For instance, by combining our algorithm with space partitioning data structures, we might even get higher performance gains. However, perhaps an even more interesting avenue would be to analyze the trade-offs that can be made when blending multiple semi-transparent layers. Apart from this, the proposed technique also makes transparency more viable for a broader range of visualization techniques. For example, it will now be possible to encode fiber tracking uncertainties in the alpha channel of the visualized fibers without eliminating the possibility to also visualize a co-registered volume.

References

[BBPtHR08] Brecheisen R., Bartroli A. V., Platel B., ter Haar Romeny B. M.: Flexible GPU-based multi-volume ray-casting. In *Vision, Modelling and Visualization* (Konstanz, Germany, 2008), pp. 303–312. 2, 7

[BCL*07] Bavoil L., Callahan S. P., Lefohn A., Comba J. a. L. D., Silva C. T.: Multi-fragment effects on the GPU using the k-buffer. In *ACM SIGGRAPH Symposium on Inter-active 3D Graphics and Games (I3D)* (San Diego, USA, 2007), pp. 97–104. doi:10.1145/1230100.1230117. 2

[BM08] Bavoil L., Myers K.: *Order-Independent Transparency with Dual Depth Peeling*. Tech. rep., NVIDIA Developer SDK 10, 2008. URL: http://developer.download.nvidia.com. 2

[Car84] Carpenter L.: The A-buffer, an antialiased hidden surface method. In *ACM SIGGRAPH Computer Graphics and Interactive Techniques* (Minneapolis, USA, 1984), pp. 103–108. doi:10.1145/800031.808585. 2

[CICS05] Callahan S., Ikits M., Comba J., Silva C.: Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics 11*, 3 (2005), 285–295. doi:10.1109/TVCG.2005.46. 2

[CMM08] Carr N., Měch R., Miller G.: Coherent layer peeling for transparent high-depth-complexity scenes. In *ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware* (Sarajevo, Bosnia-Herzegovina, 2008), pp. 33–40. doi:10.2312/EGGH/EGGH08/033-040. 2

[Cra10] Crassin C.: OpenGL 4.0+ ABuffer v2.0: Linked lists of fragment pages. Personal Blog, July 2010. Open source OIT implementation. URL: http://blog.icare3d.org/2010/07/. 2, 4, 5, 8

[CS99] Cai W., Sakas G.: Data intermixing and multi-volume rendering. *Computer Graphics Forum 18*, 3 (1999), 359–368. doi:10.1111/1467-8659.00356. 2

[EHK*06] Engel K., Hadwiger M., Kniss J. M., Rezk-Salama C., Weiskopf D.: *Real-Time Volume Graphics*. A. K. Peters, Ltd., 2006. 2

[EP90] Ebert D. S., Parent R. E.: Rendering and animation of gaseous phenomena by combining fast volume and scanline a-buffer techniques. *Computer Graphics (Proceedings of SIGGRAPH 1990) 24*, 4 (1990), 357–366. doi:10.1145/97880.97918. 2

[Eve01] Everitt C.: *Interactive Order-Independent Transparency*. Tech. rep., NVIDIA, 2001. NVIDIA white paper. URL: https://developer.nvidia.com. 2

[GBKG04] Grimm S., Bruckner S., Kanitsar A., Gröller E.: Flexible direct multi-volume rendering in interactive scenes. In *Vision, Modelling and Visualization* (Stanford, USA, 2004), pp. 386–379. 2

[GRT13] Günther T., Rössl C., Theisel H.: Opacity optimization for 3d line fields. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2013) 32*, 4 (2013), 120:1–120:8. 1

[IEE10] IEEE: IEEE Visualization Contest 2010, Oct 2010. URL: http://sciviscontest.ieeevis.org/2010/. 8

[KGB*09] Kainz B., Grabner M., Bornik A., Hauswiesner S., Muehl J., Schmalstieg D.: Ray-casting of multiple volumetric datasets with polyhedral boundaries on many-core GPUs. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2009) 28*, 5 (2009), 152:1–152:9. doi:10.1145/1618452.1618498. 2, 4

[KKP*13] Kauker D., Krone M., Panagiotidis A., Reina G., Ertl T.: Rendering molecular surfaces using order-independent transparency. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)* (Girona, Spain, 2013), vol. 13, pp. 33–40. doi:10.2312/EGPGV/EGPGV13/033-040. 2

[LF09] Lux C., Fröhlich B.: GPU-based ray-casting

(b) geometric data

(d) depth complexity

(c) volume data

(a) hybrid visualization
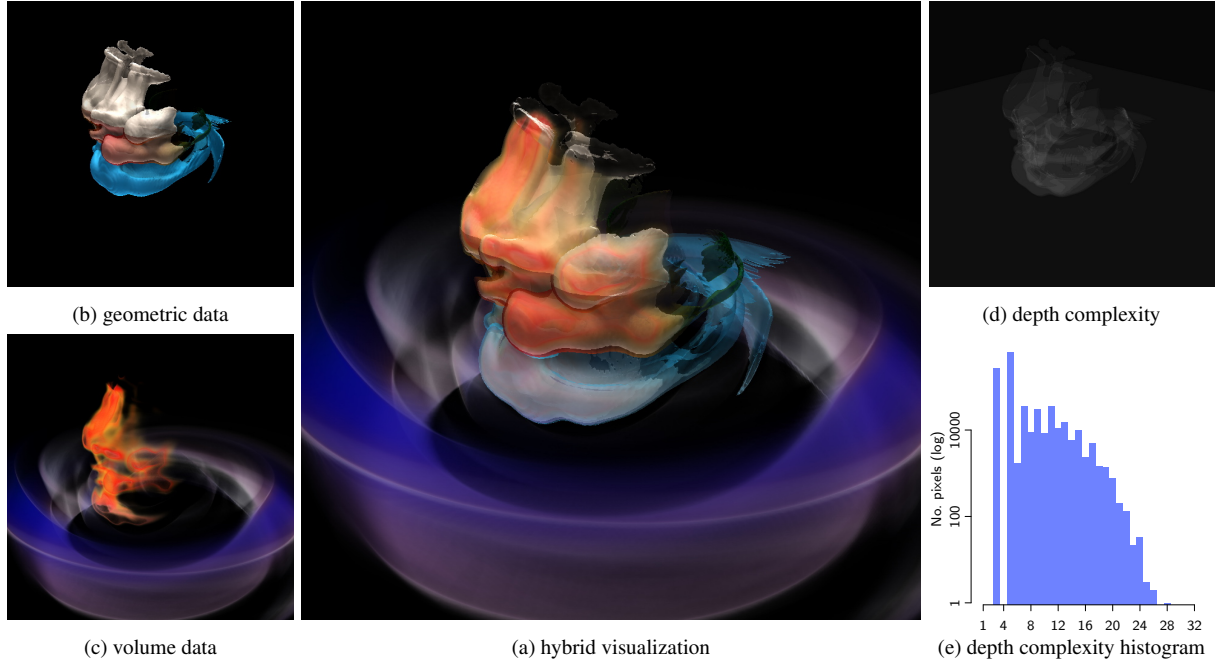
(e) depth complexity histogram

Figure 8: Visualization of solar mass ejection simulation data used for predicting space weather. (a) final rendering of hybrid visualization. (b) three geometric isosurfaces. (c) volumetric data sources. (d) image-space depth complexity (re-scaled for representational purposes). (e) depth complexity histogram (DCH, log scale). Our improved A-buffer algorithm achieves up to three times the performance compared to existing approaches.

(a) neurosurgical planning

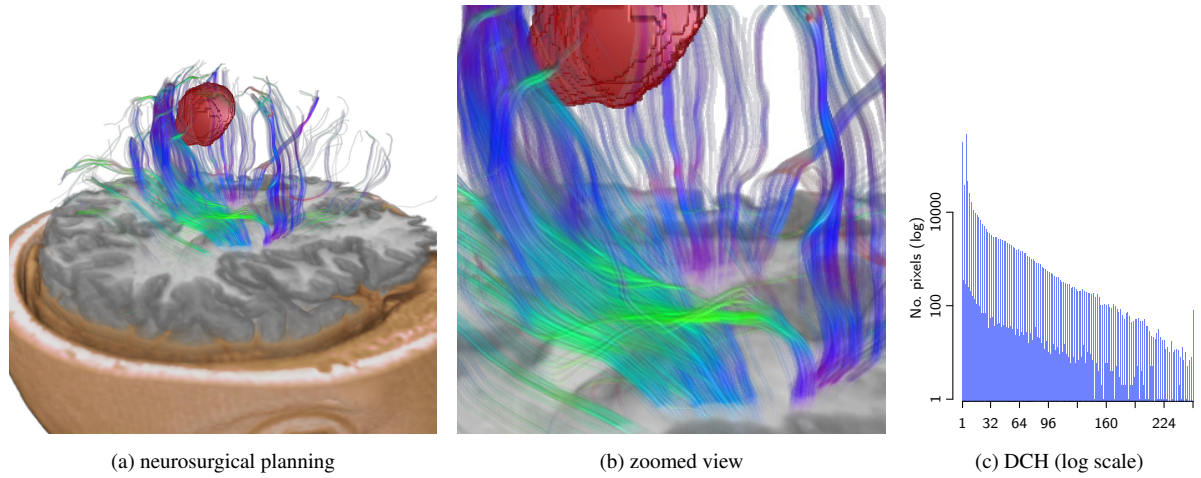(b) zoomed view

(c) DCH (log scale)

Figure 9: Fused visualization of hybrid data for the purpose of neurosurgical planning. The scene is particularly complex in areas where the DTI fibers converge towards two main bundles. (a) full scene, (b) zoomed view, and (c) the depth complexity histogram (DCH, log scale) of (a).
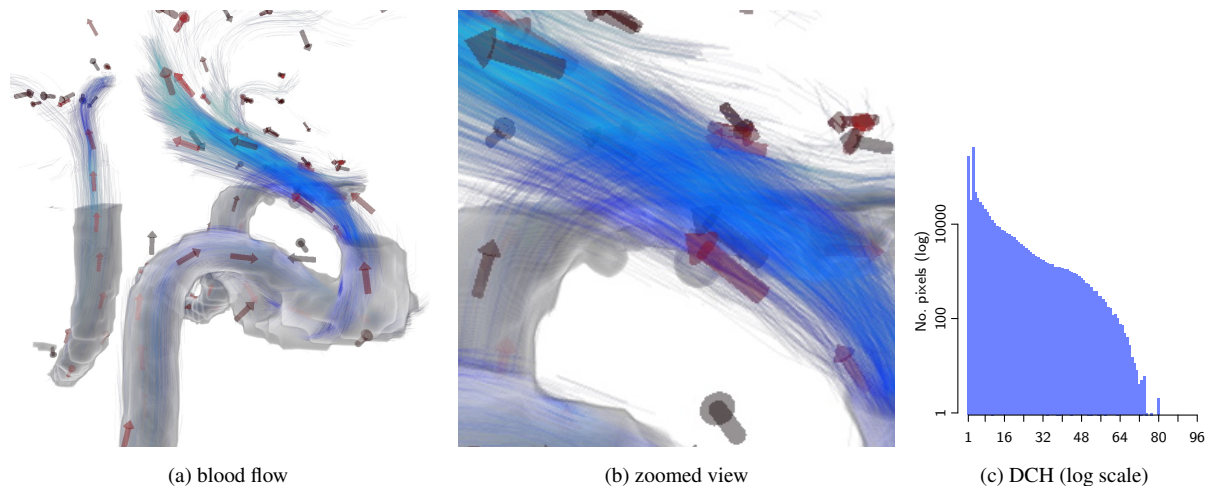
/ Hybrid Data Visualization

(a) blood flow   (b) zoomed view   (c) DCH (log scale)

Figure 10: *Fused visualization of hybrid data used to asses blood flow in the human carotid artery. The rendering uses transparency to reduce the focus on individual streamlines. (a) full scene, (b) zoomed view, and (c) the depth complexity histogram (DCH, log scale) of (a).*

of multiple multi-resolution volume datasets. In *International Symposium on Advances in Visual Computing: Part II* (2009), Springer-Verlag, pp. 104–116. doi:10.1007/978-3-642-10520-3_10. 2

[LHLW09] LIU F., HUANG M.-C., LIU X.-H., WU E.-H.: Efficient depth peeling via bucket sort. In *Conference on High Performance Graphics* (New Orleans, USA, 2009), pp. 51–57. doi:10.1145/1572769.1572779. 2

[LLHY09] LINDHOLM S., LJUNG P., HADWIGER M., YNNERMAN A.: Fused multi-volume DVR using binary space partitioning. *Computer Graphics Forum 28*, 3 (2009), 847–854. doi:10.1111/j.1467-8659.2009.01465.x. 2

[Mam89] MAMMEN A.: Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics and Applications 9*, 4 (1989), 43–55. doi:10.1109/38.31463. 2

[MB07] MYERS K., BAVOIL L.: *Stencil Routed K-Buffer*. Tech. rep., NVIDIA, 2007. URL: http://developer.download.nvidia.com. 2

[MCTB11] MAULE M., COMBA J. L., TORCHELSEN R. P., BASTOS R.: A survey of raster-based transparency techniques. *Computers And Graphics 35*, 6 (2011), 1023–1034. doi:10.1016/j.cag.2011.07.006. 2

[MCTB12] MAULE M., COMBA J., TORCHELSEN R., BASTOS R.: Memory-efficient order-independent transparency with dynamic fragment buffer. In *Conference on Graphics, Patterns and Images (SIBGRAPI)* (Ouro Preto, Brazil, 2012), pp. 134–141. doi:10.1109/SIBGRAPI.2012.27. 2, 4, 5, 8, 9

[NVI11] NVIDIA: Performance optimization. Tutorial at Super Computing Conference 2011, 2011. NVIDIA online document. URL: http://www.nvidia.com/object/sc11.html. 5

[PHF07] PLATE J., HOLTKAEMPER T., FROEHLICH B.: A flex-

ible multi-volume shader framework for arbitrarily intersecting multi-resolution datasets. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (2007), 1584–1591. doi:10.1109/TVCG.2007.70534. 2

[RBE08] RÖSSLER F., BOTCHEN R., ERTL T.: Dynamic shader generation for GPU-based multi-volume ray casting. *IEEE Computer Graphics and Applications 28*, 5 (2008), 66–77. doi:10.1109/MCG.2008.96. 2

[SdG10] SEELIGER D., DE GROOT B. L.: Ligand docking and binding site analysis with PyMOL and Autodock/Vina. *Journal of Computer-Aided Molecular Design 24* (2010), 417–422. doi:10.1007/s10822-010-9352-6. 3, 8

[SS11] SCHUBERT N., SCHOLL I.: Comparing GPU-based multi-volume ray-casting techniques. *Computer Science-Research and Development 26*, 1-2 (2011), 39–50. doi:10.1007/s00450-010-0141-1. 2

[VG07] VIOLA I., GRÖLLER E.: On the role of topology in focus+context visualization. In *TopoInVis 2005: Topology-based Methods in Visualization* (Budmerice, Slovakia, 2007), pp. 171–181. doi:10.1007/978-3-540-70823-0_12. 2

[WGER05] WEXLER D., GRITZ L., ENDERTON E., RICE J.: GPU-accelerated high-quality hidden surface removal. In *ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (Los Angeles, USA, 2005), pp. 7–14. doi:10.1145/1071866.1071868. 2

[XOL04] XIE H., OFMAN L., LAWRENCE G.: Cone model for halo CMEs: Application to space weather forecasting. *Journal of Geophysical Research: Space Physics 109*, A3 (2004). doi:10.1029/2003JA010226. 8

[YHGT10] YANG J. C., HENSLEY J., GRÜN H., THIBIEROZ N.: Real-time concurrent linked list construction on the GPU. *Computer Graphics Forum 29*, 4 (2010), 1297–1304. doi:10.1111/j.1467-8659.2010.01725.x. 2, 4