EUROGRAPHICS 2014 / B. Lévy and J. Kautz
(Guest Editors)

*Volume 33 (2014), Number 2*

# Hybrid Data Visualization Based On Depth Complexity Histogram Analysis

paper1257

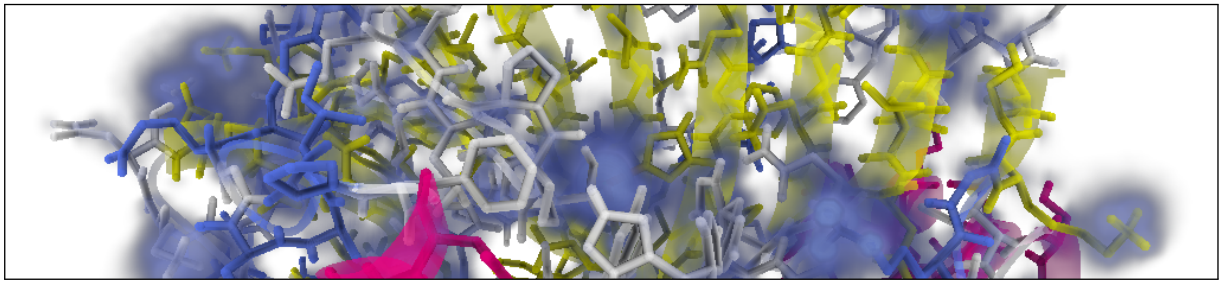


Figure 1: Application of the proposed visualization algorithm to a protein molecule data set. The contributing structures given by the bonds, electron density fields and the global protein structure can be visualized in an integrative manner, whereby semi-transparent structures of both geometries and volumes are correctly fused at a frame rate of 92.85 fps.

**Abstract**

*Visualization plays a central role when analyzing large and complex data in different scientific disciplines. In many cases, only geometric and volumetric data sets together describe a single phenomenon under observation. This heterogeneity demands specialized hybrid rendering algorithms, capable of rendering large quantities of different types of data in real-time. To better understand the demands of these scenarios, we have analyzed the overdraw complexity of heterogeneous data sets frequently occurring in different scientific disciplines. We have analyzed their view-dependent* depth complexity histograms *to investigate the rendering complexity of scenes that include both high quantities of semi-transparent geometry and volumetric data sources. Based on the findings we have made during these investigations, we propose an optimized hybrid volume/geometry rendering algorithm. The algorithm is adaptive with respect to the depth complexity of the rendered scene and maximizes the benefits of modern GPUs and results in significant performance gains compared to existing techniques. We will describe the conducted analysis and its findings, explain how these findings enable optimized multi volume/geometry fusion, and discuss the achieved results.*

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms 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 the acquired data sets. Consequently, a challenge in visualization research is the growing size of modern imaging data sets, as well as the number of captured and derived attributes. This multitude of data does not only arise from multimodal volumetric data sets, but also from geometric representations which may be derived from the data or be acquired in different ways. In many cases the integrated geometry can be of similar or even higher complexity than the volumetric data set with which it should be fused. While modern volume rendering supports the inspection of otherwise dense data sets by facilitating semi-transparency, transparency results in additional challenges when multiple volumes, or even volumes fused with

geometric data are present. While the incorporation of transparency effects in volumetric rendering can be considered as an integral part, recent work again highlights that geometric representations also benefit from semi-transparent depictions [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– an essential approach when analyzing scientific data sets.

In this paper, we propose a visualization algorithm that has been developed to support efficient rendering of hybrid data sets as they occur in today's imaging dependent areas of science. To enable an intuitive exploration of this data, we have designed our algorithm such that it enables the integration of transparency in a similar fashion as it is known from conventional volume rendering. However, similar as in volume rendering, supporting transparency leads to additional rendering challenges. The fundamental challenge arising when blending multiple semi-transparent sources into a single image, is to ensure that the elements are view-dependently blended in front-to-back or back-to-front order. When fusing multiple volumes, or including mesh geometry into the visualization pipeline, this sorting must also be ensured along each viewing ray, which results in a performance bottleneck as well as in increased memory usage. As these shortcomings forbid an interactive exploration of most hybrid data sets, we have analyzed several of these data sets from different real world scenarios with the goal to develop a hybrid visualization algorithm enabling interactive exploration. Therefore, to quantify the degree of overdraw of these data sets, we have analyzed their *depth complexity histograms* which enabled us to better understand their nature with respect to the rendering complexity. Through this analysis, we were able to classify the depth complexity of single rays and consider a global depth complexity distribution with respect to an entire data set. Based on the similarities of the occurring distributions, we were able to develop an adaptive visualization algorithm for hybrid data, which enables us to annul the memory limitations of modern GPUs, while at the same time support a performance which is doubled to quadrupled in comparison to previous algorithms. The combination of these two qualities, enables the interactive visualization of large and complex hybrid data sets at real-time frame rates, which is one of the key ingredients for enabling scientific discoveries in data intense sciences. To be able to support these two qualities, our algorithm has an adaptive nature which is realized through a two-stage processing, that maintains optimal buffer lengths with regard to GPU throughput, and thus enables significantly higher depth complexity in semi-transparent visualizations as compared to previous approaches.

In the remainder of the paper, we will describe our depth complexity analysis, and discuss its implications for the proposed hybrid visualization algorithm. We will further describe the introduced algorithm, and demonstrate its application to real world data sets as they occur in different areas of science.

## 2. Related Work

**Data fusion.** Hybrid data visualization is a growing field, providing simultaneous exploration of multiple data sources. One such data source is volumes acquired from different modalities or from multiple simulations. Typical scenes may consist of volumes with arbitrary orientations and rendering is often performed with varying sample densities for different volumes. A common approach to mitigate the limit the inevitable performance hit is to separate object space into convex regions, occupied by a fix number of volumes. Grimm *et al.* [GBKG04] introduced a structure called V-objects for this purpose. Later, Plate *et al.* [PHF07], Rössler *et al.* [RBE08], Lindholm *et al.* [LLHY09] and Lux *et al.* [LF09] introduced alternative subdivision or space-filling schemes to achieve the goal of identifying homogeneous ray segments. Many of the above mentioned works also proposed automatic shader initiation or instantiation as a flexible means to render the different volume combinations. Brecheisen *et al.* [BBPtHR08] incorporated depth peeling techniques to extract the entry and exit points for each ray segment. All of the works above are relevant literature for volumetric fusion, but employ solutions that are either based on expensive CPU side geometry operations, which effectively prohibits complex scenes, or utilize GPU transparency solutions that are less optimal for complex scenes with high amounts of geometry.

A related branch of volume fusion research deal with how the samples should be blended. Cai and Sakas [CS99] initially introduced three levels of intermixing in; image, accumulation and illumination. A recent comparison of different intermixing schemes was presented by Schubert and Scholl [SS11]. Arguably, the most frequently applied approach is to sample and compute the color contribution for all volumes separately before blending. That is, each volume has a unique transfer function. A commonly occurring simplification is to sample all volumes within a certain segment according to the highest individual step length of any volume. Both these approaches are used also in this work.

**Transparency rendering.** To ensure correct blending order or multiple semi-transparent samples, Order Independent Transparency (OIT) algorithms have been an ongoing research topic for the last thirty years. Two of the more well known approaches are depth peeling [Mam89, BM08, LHLW09] and A-buffers [Car84]. In this work we focus exclusively on the A-buffer approach which avoids the multiple rendering passes required by depth peeling. This is particularly important for scenes with a high maximum depth complexity. In such environments, the A-buffer techniques have shown superior performance [YHGT10, KKP*13] over depth peeling which requires all geometry to be re-rendered a prohibitively high number of times.

The A-buffer concept was introduced by Carpenter [Car84] as an anti-aliasing solution. Since then, the concept has spawned multiple similar techniques and variations, most of which implement the variations of the original approach on GPU hardware. Everitt [Eve01] present one of the first hardware adaption, later followed by a renewed interest as recent developments in hardware made the A-buffer approach viable for a wider set of scenes. Notable contributions include Myers and Bavoil [MB07] with stencil routed A-buffers, Yang *et al.* [YHGT10] with per-pixel linked lists (PPLL), and Crassin [Cra10] with a paged variation of per-pixel linked lists (pPPLL). A comprehensive overview was presented by Maule *et al.* [MCTB11] who later introduced an A-buffer implementation called dynamic fragment buffers (DFB) [MCTB11].

In short, a naive implementation of the A-buffer approach corresponds to a pre-allocation of fixed sized array for all pixels, causing high memory consumption. We call this a fixed fragment buffer (FFB). The main goal works cited above is to minimize this memory consumption. The idea behind the PPLL [YHGT10, Cra10] approaches is to distribute memory as needed, storing the fragments for each pixel as linked lists in a globally allocated memory pool. The DFB [MCTB11] solution was presented with a similar goal, but utilized an additional geometry pass to achieve continuous memory for individual lists, creating a better environment for successive read operations. Our work build upon the methods described in the above mentioned literature, and should improve performance for a wide range if particular A-buffer implementations.

## 3. The Challenge of Visualizing Scenes With Potentially High Depth Complexity

The major source of computational overhead introduced when fusing multiple data sources is to maintain a correct blending order among all contributions. In modern IOT solutions, this cost is directly related to the observed depth complexity in the image. That is, the number of overlapping contributions per pixel. To analyze this depth complexity, we introduce the depth complexity histogram.

### 3.1. Depth Complexity Histograms (DCH)

We define the term Depth Complexity Histogram (DCH) to a histogram that plots the distribution of depth complexities across all pixels in an image. Each bin in a DCH thus holds the number of pixels which depth complexity falls within a certain range. See Figure 3 for an example of a DCH. The computation of DCH is straightforward. First, a gray scale image is rendered where the pixel color is defined solely by the number of fragments for each pixel. The DCH is then computed as a standard histogram for this 'depth complexity image'.

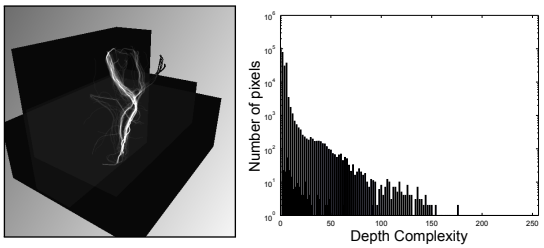


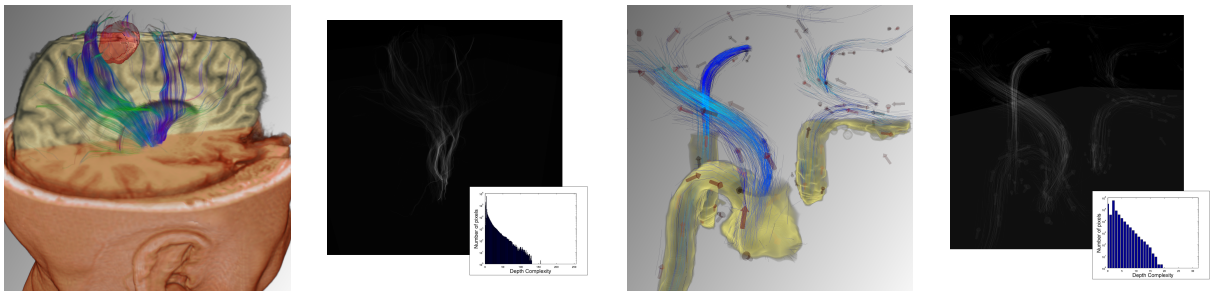

Figure 3: *Left:* Visualization of the depth complexity for a scene used to render Figure 2.a. *Right:* The Depth Complexity Histogram (DCH) of the same scene. We use the DCH to analyze the distribution of varying degrees of complexity in the image (in terms of number of pixels). An observed trend is that scenes exhibit a rapidly decreasing complexity curve (many pixels with low complexity and fewer pixels with higher complexity). The left image is normalized for presentational purposes.

### 3.2. Identified Visualization Challenges for Data Fusion On Modern Architectures

When analyzing the DCHs of commonly occurring scenes in visualization, it is apparent that the number of pixels decreases drastically for higher depth complexities. The DCH behaves like a decreasing power function. In other words, we have many pixels with low depth complexity and few pixels with high depth complexity. One example of such a scene is presented in Figure 2.a. It is constructed from the 2010 IEEE Visualization Contest data and consists of three volumetric data sets as well as a high number of geometric objects, most of which are individually modeled fibers tracts. The image space depth complexity and the corresponding DCH for the scene are presented in Figure 3.

Rendering the 2010 IEEE Visualization Contest data with transparency activated for all objects is challenging due to the potentially high number of layers that need sorted and blended in correct depth order. Even current state-of-the-art techniques are typically not sufficient to render such scenes as they are better suited to scenes with more homogeneous DCHs, or DCH's with significantly lower upper limits. The reason for this that their adaption to modern architectures have made them dependent on a maximum number of supported depth layers and that this number greatly effects performance. However, as can be seen in Figure 3 the vast majority, 91%, of the pixels have a depth complexity of 8 or less, whereas only very few, 0.2%, have a complexity of 128 or higher. This means that there are significant savings to make if dynamic management of memory could be used to avoid redundant array sizes for a majority of the pixels. As a comparison, in the molecule data set shown in Figure 6 82% of the pixels have complexity level of 8 or less and here are no pixels above a complexity level of 64.

It can be concluded that for scenes with rapidly decreas-



(a) Neurosurgical planning. (b) Depth Complexity. (c) Blood flow visualization. (d) Depth Complexity.

Figure 2: Fused visualization of hybrid data is becoming an important tool in many fields. Figure a-b shows a visualization used for neurosurgical planning, while Figure c-d shows a visualization used to asses the blood flow in the carotid artery. Both visualizations contain multiple sources of semi-transparent geometry as well as volumetric data. Performing these rendering in real time is essential in order for a user to explore the data, but challenging due to the high depth complexities of the scenes.

ing DCHs, a handful of bad pixels will have a large negative impact of the computational cost of the entire image. The challenge therefore remains to create an approach that fully utilize advanced parallel architectures and fast, but less flexible, memory hierarchies even for dynamic scenes with rapidly decreasing DCHs.

## 4. Order Independent Transparency For Data Fusion

To exploit the knowledge gained from a depth complexity analysis, as described above, we take as our starting point the the well-known concept of the A-buffer and add the capability of dynamic resource management and a per pixel depth peeling to enable handling of complex scenes and gain significant performance increase.

First we will briefly review the classical concept of the A-buffer as introduced in 1984 by L. Carpenter [Car84] before reviewing a key set of adaptations that has been proposed for its deployment on concurrent architectures. Here we will only briefly cover the functionality of the A-buffer and its associated memory usage. We will then, in the next section, propose a novel set of adaptations as well as a combined algorithm that better utilizes the parallelism and memory hierarchies of modern hardware.

For a comprehensive introduction to raster-based transparency techniques, and the A-buffer in particular, we direct the reader to the survey provided by Maule *et al.* [MCTB11]. Furthermore, as our contributions are not tied to any particular A-buffer implementation, we limit ourselves to a generalized introduction of A-buffer on modern architectures and refer the reader to relevant literature [Cra10, KGB*09, MCTB12, YHGT10] for comprehensive descriptions of particular A-buffer implementations.

### 4.1. The A-buffer concept and its adaption to modern architectures

The principle behind the A-buffer is to store a sorted list of fragments per pixel. Two relevant structures for the A-buffer approach are thus the fragment list and its associated data structure: Fragment List, and ABuffer_s. Each Fragment List is is responsible for storing information from all rasterized fragments belonging to a certain pixel. The content of the ABuffer_s structure varies with the application but minimally contains the depth value for the fragment and an object id (not uncommonly accompanied by a color and additional meta data).

To construct and resolve a Fragment List for each pixel in the image, a minimal A-buffer setup on modern architectures includes the following buffers:

*Render Target:* A render target ($W \times H \times$ vec4) for accumulating the final color.

*List Anchor:* A per-pixel anchor ($W \times H \times$ uint32) for the Fragment Lists.

*Fragment Pool:* A memory pool (size varies with implementation) for elements to all Fragment Lists.

*Atomic:* A single atomic counter (uint64) to control the distribution of memory from the Fragment Pool to multiple threads.

*Semaphore:* A per-pixel semaphore ($W \times H \times$ 1bit) to prevent race conditions when appending fragments from multiple threads to the Fragment List of a single pixel.

*Local Array:* A per-thread local array ($N_t \times L$) of ABuffer_s used for the sorting, where $N_t$ in the number of threads that share the same pool of memory in the hierarchy and $L$ is the length of the local array.

The Atomic and Semaphore buffers are standard requirements to avoid synchronization issues and race conditions in parallel environments. The Local Array is a product of the memory hierarchies that are accessible in modern hardware. It is used to speed up the sorting of the fragment arrays

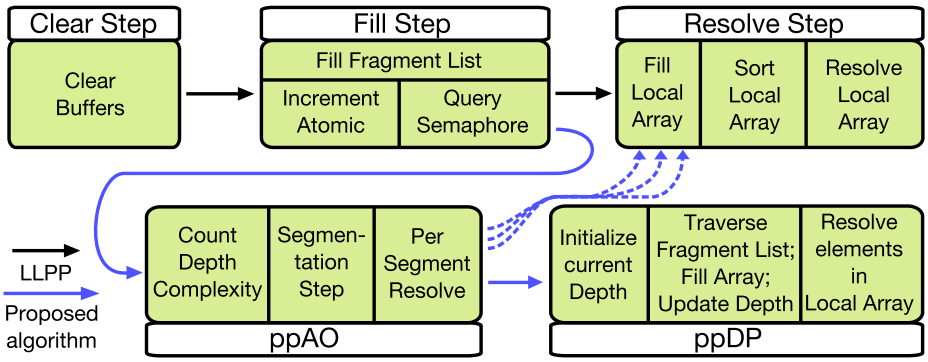



Figure 4: Flowchart!

by executing sorting algorithm, which are often read/write heavy, on faster memory higher up in the hierarchy.

The step-by-step procedure to evaluate the A-buffer is illustrated in Figure 4 and performed as follows:

*Clear Step: (before rasterization)* Clear all buffers and pointers and release any dynamically allocated memory

*Fill Step: (after rasterization)* Create Fragment Lists by allocating memory from the Fragment Pool for each rasterized fragment and append each entry to the list of the corresponding pixel (using the List Anchor). Increment Atomic to acquire memory for fragment and query Semaphore to avoid race conditions when appending fragment at List Anchor.

*Resolve Step:* Sort and traverse each Fragment List on a per-pixel basis while resolving color and transparency. Fill Local Array (from Fragment Pool), sort Local Array (in place), resolve Local Array.

Note that the Resolve Step requires the Fragment List list to be sorted in order to ensure correct blending. This is naturally one of the more computationally expensive aspects of the A-buffer approach. The sorting may be performed either during the creation of the list or after the list is completed, but must be performed prior to resolving the list. Transparency, blending and anti-aliasing are all performed while traversing the (sorted) Fragment List which at this point is stored in the Local Array.

Concurrent GPU implementation regulates storage of the Fragment Pool to the relatively large but slow VRAM memory, while the Local Array (and thus the sorting) will be located on faster memory, such as the L1 cache or registers (this memory will be re-used across multiple, subsequently executed, pixel threads). The Local Array can in some cases be explicitly assigned to a certain hierarchy level, such as in [KGB*09] where it is declared in the CUDA shared memory. If the hierarchy level of the Local Array is not set explicitly, it is in most cases left to the graphics driver discretion to declare the array on the 'fastest possible' memory level.

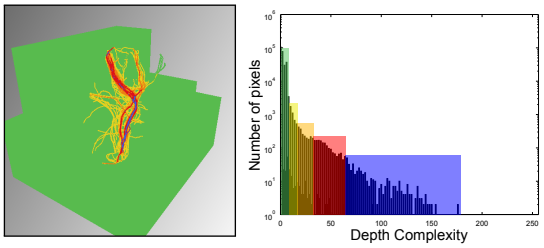



Figure 5: *Left:* The depth complexity image from Figure 3 is divided into segments with similar depth complexity. *Right:* The corresponding segmentation of the DCH with thresholds at 8, 16, 32 and 64, corresponding to 127k, 10k, 1k and .2k pixels respectively. Optimized A-buffer procedures are triggered for each segment that better leverages the available resources with regards to the segments complexity (see Section 5.2). A further specialized version of the procedure is triggered for the the segment of pixels (colored blue) which depth complexity would overflow the local array (see Section 5.3).

## 5. Introducing Methods for Dynamic Depth Complexity Management

To overcome the limitations of concurrent hardware we introduce a rendering algorithm with two novel optimizations. We begin by outlining the full algorithm before providing additional detail to each part.

### 5.1. Depth Adaptive A-buffer Rendering

The algorithm we propose adapts to the observed nature of the DCH's n two ways. First we add an additional step between fragment creation, the Fill Step step, and evaluation, the Resolve Step step. This step segments image space into sections of similar depth complexity and triggers optimal buffer sizes for each such segment. It is hence denoted per-pixel Array Optimization (ppAO).

In order to further adapt our algorithm to the nature of the

DCHs, we propose a new sorting procedure, called per-pixel Depth Peeling (ppDP), that breaks the sorting into smaller pieces by not loading all fragments at once. The procedure shares similarities to depth peeling as it iteratively sorts and resolves part of the full list, but does so fully, between global and local memory, without need to re-render geometry.

The two described improvements, the ppAO and the ppDP, can be merged with most current state-of-the-art A-buffer implementations to form an improved algorithm, as illustrated in Figure 4. The algorithm is particularly well suited for such scenes where the DCH is a decreasing power curve (many low complex pixel, fewer high complex pixels). We will now describe each part in more detail.

### 5.2. Dynamic Resource Management Using Per-pixel Array Optimization (ppAO)

The objective of ppAO is to perform the Resolve Step with an as small as possible Local Array for all pixels. This can be illustrated using the DCH, as each bin (or a group of bins) in a DCH corresponds to a *complexity segment* in the image (sets of pixels with similar depth complexity). The relation between DCH binning and complexity segments is illustrated in Figure 5.

To utilize a memory hierarchy that does not allow dynamic allocation, per-pixel array size optimization requires multiple shader programs to be instantiated with different Local Array sizes. The instantiation procedure is relatively straightforward as the individual instances only differ by a single, typically pre-process defined, number. It is also more than sufficient to instantiate a relatively low set of sizes due to the drastic decrease observed in most DCHs. We typically use segments at 8, 32 and 128 depending on the scene.

To ensure that an optimal Local Array size is selected per pixel during the Resolve Step, we need to know the depth complexities of all pixels. For this purpose we use an integer buffer:

*Complexity Counter* An integer buffer ($W \times H \times$ uint32) to count the number of fragments per pixel

The optimization takes place between the Fill and Resolve Step of the A-buffer procedure presented in Section 4.1, as illustrated in Figure 4, and is realized as follows:

A. *Per-Pixel Array Optimization* (before the Resolve Step)

- A.a *Count Step:* Count the depth complexity of each pixel
- A.b *Segmentation Step:* Divide the image into *complexity segments* (sets of pixels with similar depth complexity) using the Complexity Counter
- A.c *Per-Segmented Resolve:* Trigger the Resolve Step of the A-buffer procedure with an, per complexity segment, optimally short Local Array

The Count Step can be performed during the Fill Step, while the Fragment Lists are created, and therefore does not require an additional rendering pass. The Segmentation Step(although it sounds intimidating) is easily implemented as a masking operation based on the Complexity Counter. It is most easily performed as a part of the Per-Segmented Resolve, where thresholding the Complexity Counter is used to mimic a stencil buffer with the result that all pixels get triggered with an optimally sized Local Array. Note that the graphics pipeline typically does not need to be flushed between triggering successive segments.

### 5.3. Preventing Local Array Overflow Using Per-pixel Depth Peeling (ppDP)

The objective of ppDP is to provide a simple way to support larger Fragment Lists than what is possible to fit within the maximum size of a Local Array. This would make the highest supported depth complexity to be limited by the much larger global memory rather than by the smaller local memory.

For this, we propose a form of depth peeling between the slower Fragment Pool storage and the faster Local Array during the filling of the Local Array. In short, the Fragment Pool is read from beginning to end multiple times, with one page (size of Local Array minus one) getting sorted and resolved per iteration. Sort-on-insert and a 'current minimum depth' condition are used to ensure that the Local Array pages are peeled in depth order. The process is carried out individually for each pixel, does not require additional rendering passes, and only needs to be activated for pixels with very high depth complexity.

The optimization should be activated for all pixels that would overflow the Local Array. For these pixels, the Resolve Step of the A-buffer procedure in Section 4.1 is modified to perform the following steps (executed in a loop), as illustrated in Figure 4:

B. *Per-Pixel Depth Peeling:*
A temporary variable, *tempFrag*, is used to store information between successive loops

- B.a Copy the *furthest* used element (with the highest z-value) from the previous iteration to a temporary variable *tempFrag*. (If first run, clear tempFrag.)
- B.b Clear all elements in Local Array.
- B.c Traverse the full Fragment List and fill the Local Array. Use tempFrag to discard fragments that have already been processed in previous iterations (keep if tempFrag.z > frag.z). Fragment that are discarded due to being further away than the last element in Local Array will be processed in later iterations.
- B.d Resolve all active elements in Local Array.
- B.e Store any volume bitmasks (or other similar meta data) and start over at step B.a. Repeat until all fragments have been processed. The number of iterations is approximately $\frac{pixel\ DC}{array\ size}$ rounded up (e.g, 10 iterations

to resolve 150 depth layers with a Local Array size of 16).

It should be noted that the ppDP algorithm as described above may exhibit z-fighting issues (similar to regular depth peeling). The issue arises when multiple fragments for a single pixel have identical depth values and only a subset of these fragments get included in a particular loop iteration. This complicates the decision in the following iteration regarding which fragments should be discarded or kept (the problematic fragments will at this point share the same depth as *tempFrag*). We have so far not experienced any artifacts from this form of z-fighting. A potential way to alleviate the problem is to make sure that the *tempFrag* is either unique or the first-of-its-kind in terms of depth. All fragments sharing this depth are then known to be yet unresolved.

## 6. Implementation

The methods presented are compatible with most concurrent A-buffer implementations, including; fixed fragment buffers (FFB), dynamic fragment buffers (DFB) and linked list approaches (PPLL). Our implementation is based an openly available implementation of the A-buffer, available here [Cra10], which implements the FFB and PPLL variants. To use the ppAO part of the algorithm, a Complexity Counter needs to be added if it does not already exist within the chosen A-buffer implementation. A Complexity Counter buffer can alternatively be implemented as a stencil buffer or as an integer texture.

For our implementation we used the following A-buffer structure

*ABuffer_s: float32 z:* holds the camera space depth, *int32 id:* holds source id, *vec4 float16 pos:* holds the spatial position of the fragment in global coordinates, *vec4 float16 col:* holds the color and alpha of the of the shaded fragment.

For a total of 24 bytes per fragment. The spatial position is predominantly used to compute entry and exit points for volume rendering. Explicit storage of a per-fragment color allows for external renderers to contribute to the system at the cost of storage. For example, the geometry in most of our examples were rendered using separate modules (separate shaders and execution) for greater flexibility.

During execution, the fragments are resolved in order. If the current front fragment is part of a geometry, its color is unpacked and blended to the framebuffer. If the current fragment is part of a volume bounding box, the bitmask integer that tracks volume occupancy is updated as described in [BBPtHR08]. Between each set of consecutive fragments, a ray segment is computed in global coordinates and subsequently transformed to the local coordinate systems of the volumes, and so all volumes are sampled at each sample point along the ray segment.

## 7. Results

The presented algorithm have been tested on four real-world cases from different fields:

*Figure 6, 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 extent, a geometric "stick" representation of the protein, and a geometric "ribbon" model of the protein, both colored by secondary structure type. The three-dimensional structure and potential fields are often visualized to examine possible docking positions between the protein and other molecules [SdG10].

*Figure 7, Space:* This dataset depicts a multi-variate simulation of a time-dependent 3D MHD simulation of the heliosphere during a coronal mass ejection event as it is currently used in space weather prediction [XOL04]. The scene consists of two volumes, showing the number of charged particles and the energy density respectively, and three iso-surfaces derived from the energy density.

*Figure 2.a-b, Neuro:* The data corresponds to the IEEE Visualization Contest data from 2010, containing multiple medical imaging modalities as well as derived information sources for planning neurosurgical intervention [IEE10]. The scene consists of four data sources; two volumetric datasets depicting T1 weighted MRI images of the head and brain, and two sources of geometric information in a surface extraction of a tumor segmentation as well as a set of DTI fiber tracts. The combined information from all data sources is used to plan the safest possible access for intervention.

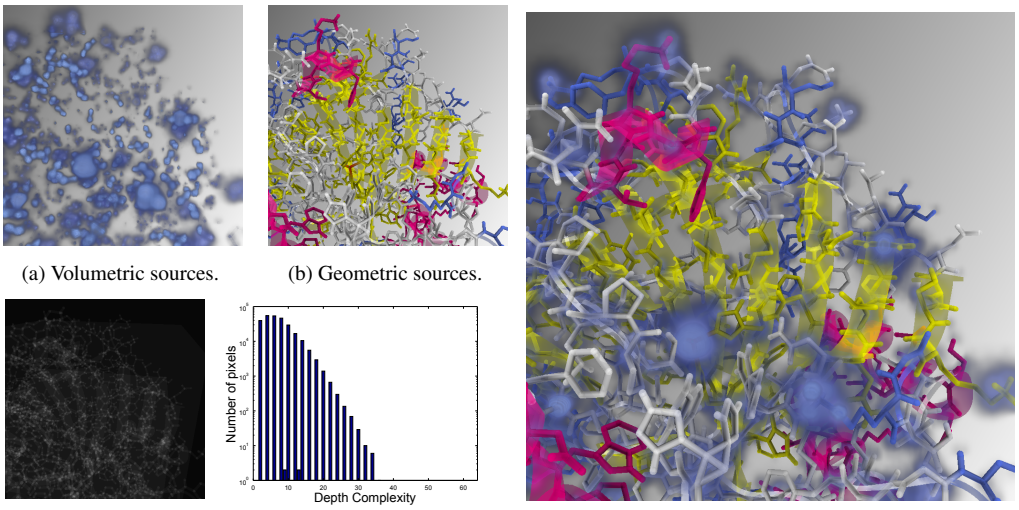
*Figure 2.c-d, Flow:* The data depicts the blood flow in the carotid artery of a human subject. The scene consists of a single Computed Tomography image, a two sets of geometric primitives in the form of individual particle streamlines and as well as a more sparse glyph representation.

### 7.1. Performance comparisons

The full algorithm and its two proposed parts, the ppAO and the ppDP, were tested on a Nvidia GeForce 560 Ti and a Nvidia GeForce 580. We begin by examining the parts separately.

For the ppAO, replacing a single shader that allocates a 64 element Local Array with a combination of four shaders that allocates Local Array sizes of 64-32-16-8 elements respectively provides an average performance gain of 3 times. The same gain when replacing a 32 element Local Array with a 32-16-8 combination lies at an average of 1.8 times. The trend holds for both for larger as well as smaller array sizes. The overhead for setting up the buffers and triggering the separate patches was negligible in out tests.

For the ppDP, we observed notable results for the Fermi




(a) Volumetric sources. (b) Geometric sources.

(c) Depth complexity. (d) Depth Complexity Histogram. (e) Final protein visualization.

Figure 6: Visualization of protein 2CBA, including both geometric representations or the protein structure and computational volumetric data. Images show; a) two volumetric data sources, both depicting the electrostatic potential at different resolutions and extents, b) three semi-transparent iso-surfaces derived from energy density, c) the image space depth complexity with a white points of 64, and d) the corresponding Depth Complexity Histogram (DCH), in log scale. Our improved A-buffer algorithm is capable of rendering the entire scene at 36 fps.

based GPUs from Nvidia that suggest looping a smaller array a higher number or iterations is significantly faster than looping larger arrays fewer times. For example, the average framerate when replacing a 64 element Local Array (without ppDP) with a 32 element Local Array looped twice (using ppDP), the framerate increases from 6 to 17 FPS. When running an 8 element Local Array looped 8 times, the performance increases further to around 40 FPS. This notable increase in performance for shorter array sizes is most likely a product of driver optimization, and its source being the limited size L1 cache. When scheduling a set of rays where each thread allocates a Local Array of 64 elements, the GPU can barely maintain a single thread per core before running out of L1 space. On the other hand, if each thread only allocates a Local Array of 8 elements, the GPU can schedule multiple rays per core and hide global memory latency by hot-swapping multiple threads and thus maximize throughput [Nvi11].

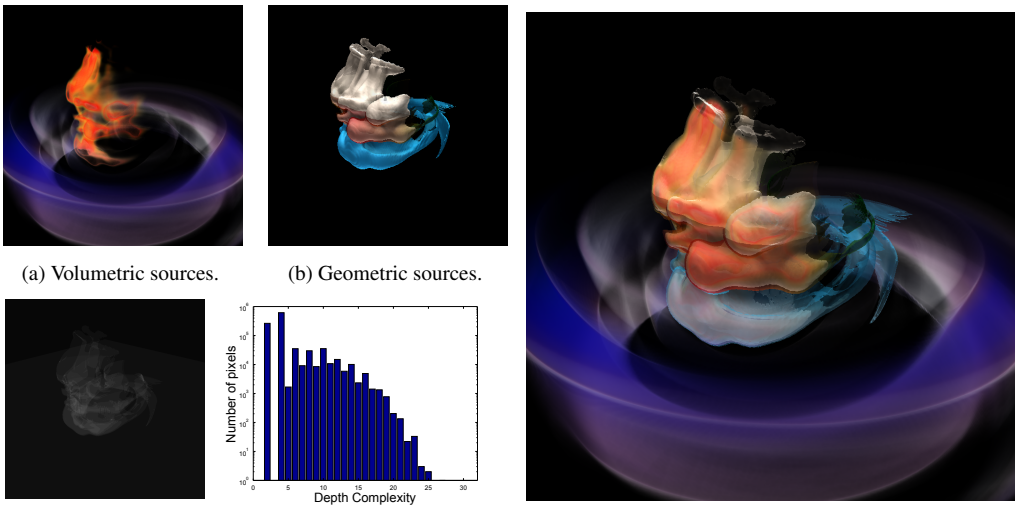
The best possible performance was achieved when running the full algorithm, activating both ppAO and ppDP. The configuration included two shaders; the first utilizing an 8 element array without looping (for executing all pixels between 1–8 depth complexity), the second utilizing an 8 element array with looping activated (for executing all pixels with depth complexities greater than 8). With this configuration, our algorithm rendered all 160+ depth layers of the IEEE Visualization Contest data with a performance gain of up to 8 times over a reference implementation that only rendered the first 64 layers.

Performance naturally varied between GPU selection and choice of underlying A-buffer implementation. Table 1 contains the performance measures as performed on the GeForce 580 using PPLL as the underlying A-buffer implementation. The trends observed for the 580 also held up for the 560 Ti (with overall lower framerates), and also held up when running FFB as the underlying A-buffer implementation.

## 8. Conclusions and Future Work

In this paper, we have introduced a novel visualization technique which supports the interactive exploration of complex heterogeneous data. By applying our approach, it becomes for the first time possible, to fuse volumetric data with semi-transparent geometry which has a high depth complexity. The proposed algorithm has been developed based on the observations made, when analyzing the depth complexity of heterogeneous data sets as acquired in modern imaging-based sciences. Therefore, we have facilitated the analysis of depth complexity histograms, which lead to the development of the proposed adaptive data handling scheme. By performing a delayed depth sorting, we are thus able to exploit the full potential of the fast local GPU memory, as we can keep subsets of the set of samples to be sorted in this memory. We have shown the performance gain achieved by our method, and discussed its application to real-world data sets.

In the future, we see several opportunities to improve the proposed method. For instance, by combining it with space




(a) Volumetric sources. (b) Geometric sources.

(c) Depth complexity. (d) Depth Complexity Histogram. (e) Final space weather visualization.

Figure 7: Visualization of a solar mass ejection simulation, used for prediction of space weather. Images show: a) two volumetric data sources, depicting the inner and outer extents of the simulation at different resolution, b) two geometric models; a "stick" representation of the atomic structure and a "ribbon" representation of secondary structures, c) the image space depth complexity with a white point of 64, and d) the corresponding Depth Complexity Histogram (DCH), in log scale. Our improved A-buffer algorithm achieves up to four times the performance by better leveraging the available fast local memory on the GPU.

partitioning data structures, we might even get a higher performance gain. However, we would also be interested in analyzing the tradeoffs that can be made when blending multiple semi-transparent layers. Apart from this, the proposed technique also paves the way for new visualization techniques. For instance, it will now be possible to encode fiber tracking uncertainty in the alpha channel of the visualized fibers, without eliminating the possibility to also visualize a coregistered volume. In the future, we would like to explore this and other interesting possibilities.

## References

[BBPtHR08] BRECHEISEN R., BARTROLI A. V., PLATEL B., TER HAAR ROMENY B. M.: Flexible GPU-based Multi-Volume Ray-Casting. In *VMV* (2008), pp. 303–312. doi:10.1109/MCG.2008.96. 2, 7

[BM08] BAVOIL L., MYERS K.: *Order-Independent Transparency with Dual Depth Peeling*. Tech. rep., NVIDIA Developer SDK 10, Feb. 2008. URL: http://developer.download.nvidia.com/SDK/10\/opengl/src/dual_depth_peeling/doc/DualDepthPeeling.pdf. 2

[Car84] CARPENTER L.: The A-buffer, an antialiased hidden surface method. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1984), SIGGRAPH '84, ACM, pp. 103–108. doi:10.1145/800031.808585. 2, 3, 4

[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/opengl-40-abuffer-v20-linked-lists-of.html. 3, 4, 7

[CS99] CAI W., SAKAS G.: Data Intermixing and Multi-volume Rendering. *Computer Graphics Forum 18*, 3 (1999), 359–368. 2

[Eve01] EVERITT C.: Interactive Order-Independent Transparency, 2001. 3

[GBKG04] GRIMM S., BRUCKNER S., KANITSAR A., GRÖLLER E.: Flexible Direct Multi-Volume Rendering in Interactive Scenes. In *Vision, Modeling, and Visualization* (2004), pp. 386–379. 2

[GRT13] GÜNTHER T., RÖSSL C., THEISEL H.: Opacity optimization for 3d line fields. *ACM Transactions on Graphics (Proc. ACM SIGGRAPH)* (2013), (to appear). 2

[IEE10] IEEE: IEEE Visualization Contest 2010, Oct 2010. URL: http://sciviscontest.ieeevis.org/2010/. 7

[KGB*09] KAINZ B., GRABNER M., BORNIK A., HAUSWIESNER S., MUEHL J., SCHMALSTIEG D.: Ray-Casting of Multiple Volumetric Datasets with Polyhedral Boundaries on Manycore GPUs. In *ACM Transactions on Graphics (TOG)* (2009), vol. 28, ACM, p. 152. doi:10.1145/1618452.1618498. 4, 5

[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)* (2013), Association E., (Ed.), vol. 13, pp. 33–40. doi:10.2312/EGPGV/EGPGV13/033-040. 2

[LF09] LUX C., FRÖHLICH B.: GPU-Based Ray-Casting of Multiple Multi-Resolution Volume Datasets. In *Proceedings of the 5th International Symposium on Advances in Visual Computing: Part II* (Berlin, Heidelberg, 2009), ISVC '09, Springer-Verlag, pp. 104–116. doi:10.1007/978-3-642-10520-3_10. 2

| | | | Performance Configurations (timings in FPS) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | PPLL | | | PPLL +ppAO | PPLL +ppDP | | PPLL +ppAO +ppDP | |
| Setup | Data | Data size | 32 | 64 | 128 | 128—8 | 8* | 16* | 8*,8 | % |
| Protein | VOL | $127 \times 127 \times 127$ | 53.08 | 25.34 | 10.30 | 66.71 | 87.03 | 74.68 | 92.85 | **3.6X** |
| | VOL | $71 \times 71 \times 71$ | (NC) | | | | | | | |
| | GEO | 45k triangles | | | | | | | | |
| | GEO | 36k triangles | | | | | | | | |
| Space Weather | VOL | $256 \times 256 \times 256$ | 9.86 | 4.21 | 3.22 | 28.56 | 34.86 | 24.80 | 36.15 | **8.5X** |
| | VOL | $256 \times 256 \times 256$ | (NC) | | | | | | | |
| | GEO | 162k triangles | | | | | | | | |
| | GEO | 100k triangles | | | | | | | | |
| | GEO | 67k triangles | | | | | | | | |
| Neuro | VOL | $415 \times 487 \times 176$ | 21.19 | 9.25 | 3.29 | 7.94 | 39.37 | 32.04 | 40.73 | — |
| | VOL | $367 \times 395 \times 150$ | (NC) | (NC) | (NC) | (NC) | | | | |
| | GEO | $54 \times 52 \times 29$ | | | | | | | | |
| | GEO | 1678 streamlines | | | | | | | | |
| Flow | VOL | $76 \times 49 \times 45$ | 19.25 | 9.35 | 3.93 | 29.43 | 32.95 | 27.05 | 34.31 | **3.6X** |
| | GEO | 1986 streamlines | (NC) | | | | | | | |
| | GEO | $\approx 100$ arrows | | | | | | | | |

Table 1: Interactive performance measurements of our approach. NC = Not correct, configuration can't handle current depth complexity. For the Neuro scene, no existing algorithms could render the scene without discarding unresolved fragments.


[LHLW09] LIU F., HUANG M.-C., LIU X.-H., WU E.-H.: Efficient Depth Peeling via Bucket Sort. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), HPG '09, ACM, 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. In *Computer Graphics Forum* (2009), vol. 28, pp. 847–854. 2

[Mam89] MAMMEN A.: Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique. *IEEE Comput. Graph. Appl. 9*, 4 (July 1989), 43–55. doi:10.1109/38.31463. 2

[MB07] MYERS K., BAVOIL L.: Stencil routed A-Buffer. In *ACM SIGGRAPH 2007 sketches* (New York, NY, USA, 2007), SIGGRAPH '07, ACM. doi:10.1145/1278780.1278806. 3

[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. 3, 4

[MCTB12] MAULE M., COMBA J., TORCHELSEN R., BASTOS R.: Memory-Efficient Order-Independent Transparency with Dynamic Fragment Buffer. In *Graphics, Patterns and Images (SIBGRAPI), 2012 25th SIBGRAPI Conference on* (2012), pp. 134–141. doi:10.1109/SIBGRAPI.2012.27. 4

[Nvi11] NVIDIA: Performance Optimization. Tutorial at Super Computing Conference 2011, 2011. Nvidia online document. URL: http://www.nvidia.com/object/sc11.html. 8

[PHF07] PLATE J., HOLTKAEMPER T., FROEHLICH B.: A Flexible Multi-Volume Shader Framework for Arbitrarily Intersecting Multi-Resolution Datasets. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (Nov. 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. *Computer Graphics and Applications, IEEE 28*, 5 (2008), 66–77. 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. 7

[SS11] SCHUBERT N., SCHOLL I.: Comparing GPU-based Multi-Volume Ray-Casting Techniques. *Comput. Sci. 26*, 1-2 (Feb. 2011), 39–50. doi:10.1007/s00450-010-0141-1. 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). 7

[YHGT10] YANG J. C., HENSLEY J., GRÜN H., THIBIEROZ N.: Real-time Concurrent Linked List Construction on the GPU. In *Proceedings of the 21st Eurographics conference on Rendering* (Aire-la-Ville, Switzerland, Switzerland, 2010), EGSR'10, Eurographics Association, pp. 1297–1304. doi:10.1111/j.1467-8659.2010.01725.x. 2, 3, 4