An Analysis of Scalable GPU-Based Ray-Guided Volume Rendering

Thomas Fogal*
HPC Group, Duisburg-Essen; SCI

Alexander Schiewe†
HPC Group, Duisburg-Essen; Intel VCI

Jens Krüger‡
HPC Group, Duisburg-Essen; SCI; Intel VCI

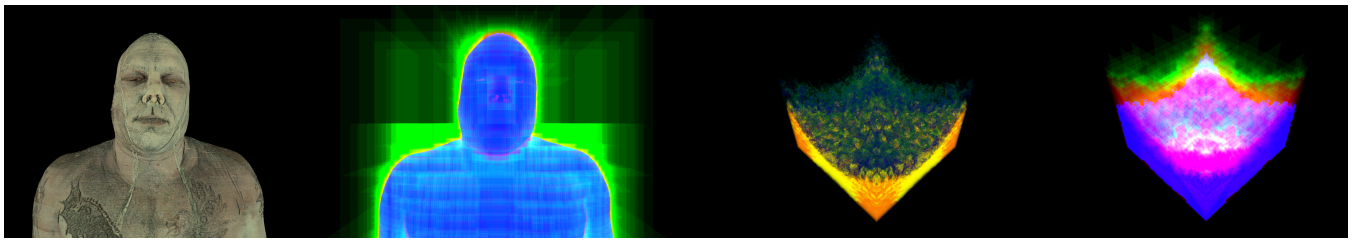

Figure 1: The Visible Human male full color (~12 GB) and a Richtmyer-Meshkov instability (~8 GB) render in 34 ms and 58 ms, respectively, using our ray-guided volume rendering implementation. On right are views which highlight the areas that take advantage of empty space leaping (green) and early ray termination (blue).

Abstract

Volume rendering continues to be a critical method for analyzing large-scale scalar fields, in disciplines as diverse as biomedical engineering and computational fluid dynamics. Commodity desktop hardware has struggled to keep pace with data size increases, challenging modern visualization software to deliver responsive interactions for $O(N^3)$ algorithms such as volume rendering. We target the data type common in these domains: regularly-structured data.

In this work, we demonstrate that the major limitation of most volume rendering approaches is their inability to switch the data sampling rate (and thus data size) quickly. Using a volume renderer inspired by recent work, we demonstrate that the actual amount of visualizable data for a scene is typically bound *considerably* lower than the memory available on a commodity GPU. Our instrumented renderer is used to investigate design decisions typically swept under the rug in volume rendering literature. The renderer is freely available, with binaries for all major platforms as well as full source code, to encourage reproduction and comparison with future research.

Index Terms: I.3.0 [Computing Methodologies]: COMPUTER GRAPHICS—General; I.3.m [Computing Methodologies]: COMPUTER GRAPHICS—Miscellaneous;

1 Introduction

Modern volume rendering is heavily focused on the concepts of empty space skipping and the fast detection of ray saturation. Both of these concepts have extensive effects on the amount of compute work required. However, even more relevant is their ability to reduce the working set of extremely large datasets down to a small kernel, which can significantly reduce the amount of data which must be loaded from a slow resource, such as the network or a local disk. This has enabled interactive volume rendering for very large data on commodity hardware [1, 2, 3].

There are a variety of trade-offs in the development of a modern volume renderer. The choice of brick size, for example, can significantly impact the effectiveness of empty space skipping. We note that the presentation of most volume rendering systems lacks detailed insight into these parameters. Further, these factors can interact in complex ways. As an example, empty space skipping works considerably better with smaller bricks sizes, but disk throughput drops sharply with small requests. Compression can further complicate the issue.

We seek to rectify this situation by performing a thorough study of the interaction of these parameters within the context of GPU-based ray driven volume rendering. We have surveyed recent volume rendering literature and implemented a renderer by piecing together the best ideas from a multitude of systems. These ideas were extended with notions required for our environment—for example, by removing the requirement that datasets fit in GPU memory. Along the way, we instrumented every corner of the renderer and utilized this instrumentation to exhaustively explore relevant options. The final result achieves better performance than previous work and provides a guided tour through the maze of design choices available in a modern volume renderer.

2 Related Work

Volume visualization on consumer graphics hardware has become widely utilized as a means to cope with the growing sizes of data. GPUs have proven useful in both ray-tracing and rasterization techniques [4, 5], rendering of diverse scenes [6], as well as considerably more general tasks [7].

Volume rendering accelerated by GPU hardware was established in the mid-90's [8, 9], initially based on hardware compositing of volume slices. The ability to do raycasting came later [10]. Since the time of the initial GPU-based volume renderers, researchers have been concerned with methods to work around the limited memory available on GPUs. The prominent technique for volume rendering large data on a GPU is to use a multiresolution representation [11, 12, 13]. This method hinges on the concepts of empty space leaping and early ray termination [14], two techniques developed early on which demonstrate that sampling can be significantly reduced in many instances of volume rendering.

There has been much work on accelerating ray-traced volume rendering in recent years. Voreen implements a more general architecture, including GPU-based raycasting [15]. Tuvok implements a flexible volume rendering system with support for very large datasets [16, 17]. Knoll et al. utilize a bounding volume hierarchy and optimized SSE to achieve very fast volume renderings [1]. Gobbetti et al. and Boada et al. detail methods for traversing tree structures on the GPU for the purpose of volume rendering [18, 19]. The Gigavoxels [3] system traverses N^3-trees on the GPU to choose an effective resolution. With the large gap between processing power

*e-mail: tfogal@sci.utah.edu
†e-mail:schiewe@intel-vci.uni-saarland.de
‡e-mail:jens.krueger@uni-due.edu

IEEE Symposium on Large Data Analysis and Visualization 2013
October 13 - 14, Atlanta, Georgia, USA
978-1-4799-1658-0/13/$31.00 ©2013 IEEE

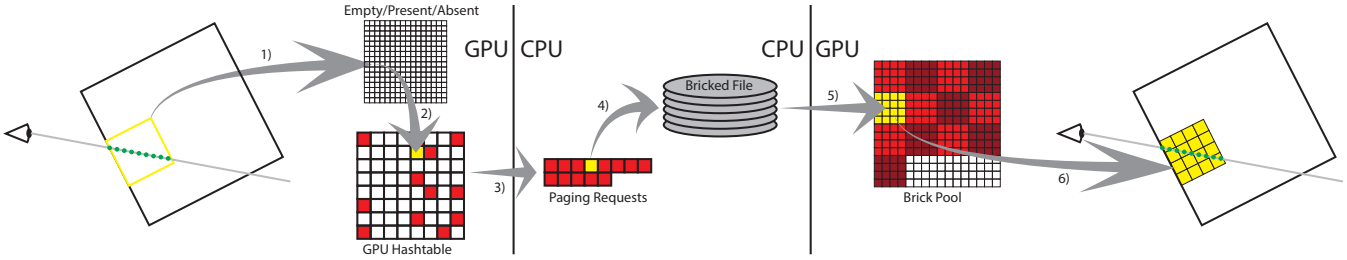

Figure 2: The missing brick reporting / paging subsystem of our volume rendering approach. Missing bricks are recorded into a hash table (1, 2), to be paged in (3, 4, 5) and rendered in subsequent frames (6).

and data sizes, some communities have turned to distributed memory systems for large-scale volume rendering [20, 21, 22, 23].

Our algorithm employs a lock-free data structure on the GPU for feedback information. Highly-concurrent Lock-free structures are ideal for the manycore GPU environment, however they have previously been challenged by the lack of concurrency primitives available for the OpenGL platform. We make use of a lock-free hash table very similar to that of Michael's [24], implemented in a manner similar to Lux and Fröhlich's implementation for terrain rendering [25].

Hadwiger et al. presented a volume renderer similar to ours [2]. Their system is aimed at volume rendering highly anisotropic data as it is streamed real-time from a high-resolution microscope. Our renderer improves upon theirs in a number of ways:

- We perform brick lookup each brick, instead of every sample, maintaining the simple and familiar ray-marching core that is well-documented in volume rendering literature.
- We expound on how to use modern GPU features to implement our lock-free feedback data structure, which enables the implementation to spend more time computing on the GPU and less time pushing data around.
- We utilize an out-of-core, progressive rendering methodology, breaking the GPU-memory-size barrier that limits data sizes from Hadwiger et al.'s work. This also allows us to gracefully scale down to consumer-level graphics cards.

While we believe these to be novel additions, we do not consider them to be this work's major contribution. Rather, we provide new depth to the discussions of a variety of parameters which are relevant in the development of a ray-guided direct volume renderer:

- The strategy to be used to load higher resolution data when a variety of intermediate choices are possible;
- an understanding of the miasma of issues surrounding bricking and brick sizes;
- empirical evidence demonstrating that the working set for direct volume rendering is indeed bound more by the screen resolution than the dataset;
- a novel method for ray-guidance storage and propagation to the input system's logic;
- how to effectively handle real-time updates to the transfer function; and
- the effect of brick layout strategies on large volume access times.

In contrast to previous renderers, ray-guided volume renderers couple the rendering process with the identification of which sub-volumes ('bricks') must be loaded. We describe the operation of ray-guided volume renderers, in Section 3. In Section 4 we detail a plethora of benchmarks which demonstrate the performance of the renderer.

In many prior volume renderer evaluations, results are generally limited to the raw performance of the renderer. However, we note that—for some reason—users of our volume renderer rarely ask how many milliseconds it takes to render the visual human. One thing users *do* ask is how large the data can get before the renderer becomes unusable. For this reason, we have engineered our renderer so that it does not require that the volume fit in core. Furthermore, users generally value a responsive system over a performant system. They are curious if money should be spent upgrading a video card or buying a solid state drive. Design elements are carefully expounded and conclusions are drawn in Section 5.

Finally, Section 6 gives our final remarks, and note both limitations and opportunities for future work.

3 Ray-Guided Grid Leaping

At the macro level, our algorithm is reminiscent of the recent work of Hadwiger et al. [2], as well as Engel's CERA-TVR [26] which in turn is based on the Gigavoxels system [3].

With Hadwiger et al., we share the requirement of a set of simple multiresolution Cartesian grids, along with an OpenGL-based table to report missing bricks. A multiresolution hierarchy is built as a preprocess for input data which exist at only one resolution (details are in Section 5). From the CERA-TVR system we inherit the idea to only recompute and request grid cells at boundaries.

3.1 Overview

We endeavor to create a volume renderer which can render massive datasets extremely fast on commodity GPU hardware. The major issues in such a renderer are:

1. Identifying regions which must be sampled densely.
2. Precisely locating the transition between these regions and regions which exhibit considerable homogeneity.
3. Terminating a ray as soon as possible.
4. Efficiently communicating regions to be rendered in the future to the IO layer.

Points (1) and (2) ensure we concentrate the computational effort on the areas which require it. Point (3) is critical because it means we do not have to load the data beyond the point of early termination, significantly reducing costly disk traffic. If point (4) is not sufficiently addressed, the renderer will load large amounts of data which are not needed for rendering, at severe costs in performance.

To the first point, we employ an efficient metadata structure which allows us to quickly identify these regions. Points (2) and (3) are handled through an educated choice of brick size, which is discussed more thoroughly in Section 5. A major component to modern volume renderers is how they address point (4), now by and large based on *ray guidance*. That is, the sampling characteristics of the ray determine which data to load. Stated differently, the future data

requirements are computed *in concert* with standard ray traversal and accumulation.

The entire operation is detailed in Figure 2. For each ray we compute the level of detail required to maintain a pixel error of less than one. With this level and the position in the volume we compute a brick index. This brick index is used to fetch information from a lookup table (Figure 2.1) to identify whether the brick is a) empty, b) non-empty and present on the GPU, or c) non-empty and absent. When it is empty, we skip the brick and repeat the process at the brick's exit point. When it is non-empty and present, we ray-cast that brick. When the brick is non-empty *and* not resident in GPU memory, the system returns the finest coarser level available and the missing entry is added to a GPU hash table (Figure 2.2). This table is read back to the host memory at the end of the frame (Figure 2.3), and used to page in bricks from disk or cache (Figure 2.4). A paged-in brick is then uploaded to a GPU texture pool (Figure 2.5), and a subsequent frame will use this portion of the brick pool for sampling (Figure 2.6).

The key component is that both ray-accumulation *as well as* identification of the bricks which are needed should occur on the GPU. The latter is natural to compute during standard ray-casting operations. Doing both operations on the GPU means brick identification comes very cheap, as it parallelizes very effectively. More importantly, performing this during ray-casting ensures that it is optimally accurate: the program never loads data which will not be used.

Algorithm 1 Ray-guided volume rendering. Each ray identifies the set of bricks which it needs for rendering independently, and reports this information for use in subsequent rendering passes.

```
1:  color = rayResumeColor
2:  terminated = true                          ▷ assume ray will finish
3:  repeat
4:      LoD = ComputeLOD(Depth(ray))
5:      brick, samplingRate = GetBrick(ray)
6:      offsets = PoolOffsets(brick)
7:      if samplingRate ≠ RequiredSamplingForLOD(LoD) then
8:          ReportMissingBrick(brick)
9:          if terminated then                 ▷ first missing brick?
10:             terminated = false
11:             rayResumePos = ray
12:         end if
13:     end if
14:     Raycast(ray, stepSize, offsets)
15: until ray ≥ exit ∨ Saturated(ray)
16: rayResumeColor = color
17: if done then
18:     rayResumePos = FINISHED
19: end if
```

The basic algorithm is given in Algorithm 1. Briefly, the appropriate sampling rate is identified and we look for the data at that resolution (lines 4, 5). `GetBrick` will always return some data, but the data may be at a lower resolution than request; this is communicated through the `stepSize` and the situation is handled on line 7. If our data are too coarse, we note that we are missing a brick (`ReportMissingBrick`) and where we are in the volume (`rayResumePos`) when this *first* occurred (`terminated`).

Every iteration through the outer loop, we perform this identification of the appropriate resolution. This satisfies our first goal as mentioned above: we identify the appropriate sampling resolution at every brick boundary. With small bricks, this means we will do few integration steps before early ray termination is recognized. Furthermore, we detect empty bricks at this stage as well.

3.2 Missing Data

As noted above, it is possible that data are undersampled while rendering. When this occurs, we display a coarser version of the data initially, but progressively refine those regions with finer resolution data until they are sampled at a rate of a single voxel per pixel, or the maximum data resolution available. This information is collected by the GPU as it renders, but must be communicated back to the CPU to coordinate disk access and update the appropriate area of the volume pool.

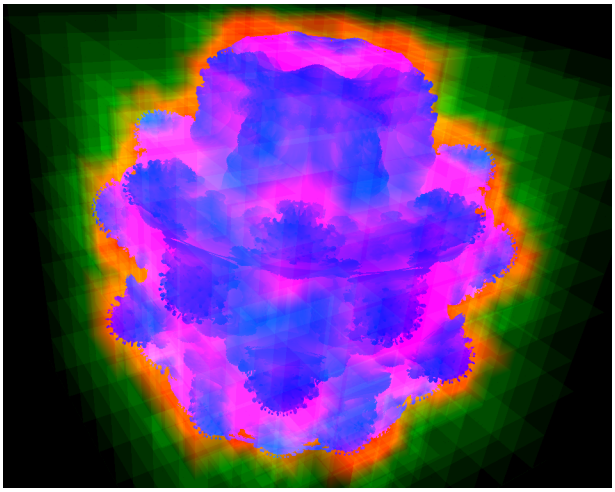

Figure 3: Volume rendering behavior for the Mandelbulb dataset. Green indicates bricks which were skipped via empty space skipping. Red indicates bricks which were sampled densely. Blue indicates bricks which were sampled but saturated quickly.

One solution for this would be to use multiple render targets to store information on which bricks are missing [3]. The limitation of this method is the limited mapping operation from the ray to the target buffer: there are only so many available render targets. Furthermore, this approach ignores the inherent spatial coherency between rays. Two neighboring rays are highly likely to request the same set of bricks, or at least have substantial overlap within the sets they require. With the multiple render targets approach, both pixels will encode the same value, and we will need to read back larger textures which consist of predominantly duplicate values.

Instead of utilizing extra render targets, we take advantage of an OpenGL extension which was promoted to core in version 4.2, `GL_ARB_shader_image_load_store`. This extension allows the creation of an image buffer which is independent of the current rendering buffer. Using the atomic load/store operations the extension provides, we implement a set based on a linearly-probed lock-free hash table stored in an `image_load_store` buffer. Since we are hashing based on the brick, multiple rays requesting the same brick hash to the same position. This allows us to keep the table—and therefore how much information we read back per-frame—quite small. We discuss sizing of the hash table in more detail in Section 5.3.1.

3.3 Brick Classification

Considering our target goals (1) through (3) given at the beginning of this section, one could classify a brick into one of three categories:

- skipped due to empty space skipping,
- early termination due to ray saturation, or
- sampled densely without saturating.

An important observation is that—in a very large number of cases—bricks fall into *either* the 'empty' or 'saturating' categories, and only *rarely* in the 'non-saturating' category. The factor which has the greatest effect on performance is how quickly a renderer can classify data into one of the first two categories, and therefore bypass a large set of the work.

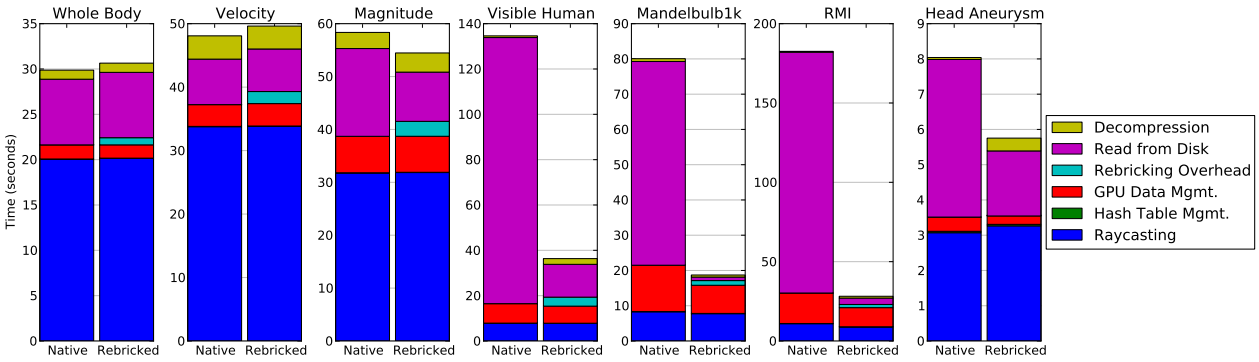

Figure 4: Time spent at various stages of our pipeline, aggregated over the generation of a rotation sequence. Comparisons are made between data stored with the ideal brick size for that dataset ('Native'), and data stored at a large brick size of 256^3 with the ideally-sized bricks created at run-time ('Rebricked'). 'Whole Body', 'Velocity', and 'Magnitude' suffer from a lack of ray saturation.

To make this identification effective, ray-guided volume renderers maintain the state of each brick, shared on both the GPU and host memories. During rendering, one uses the table to identify if a brick is empty. If so, the renderer leaps over that space instead. We store this as an array consisting of one 32-bit integer per brick of the dataset.

Figure 3 visualizes this classification for a large dataset under a typical view and transfer function. As shown there, the majority of the visualization falls into either the 'blue' (saturated quickly) or 'green' (skipped) sets. This also demonstrates how little data is actually required for a typical volume rendering. A similar rendering is given in the rightmost image of Figure 1, in which only the rays in the middle of the volume require high computation.

Of course, this classification depends largely on the transfer function and viewing parameters. In practice, however, transfer functions which produce *informative* visualizations tend to exhibit such ternary classifications.

When the transfer function is changed, this metadata information must be recomputed. For datasets with many bricks, this can induce a noticeable delay. Our current test platform can process about 7.5 million bricks per second, but even a 1 second delay between interactions is too much. Therefore, we offload this update to a background thread. Until the thread completes its work, the renderer considers all unprocessed bricks to be 'missing', causing it to request bricks which might be empty. Those bricks' metadata is directly updated and they are only loaded if they fail the empty check. The overall performance effects may be large, but the system remains responsive during this period.

4 Performance

In this section, we give an overview of the various stages of the renderer and how they perform. Unless otherwise noted, all timings were performed on a dual quad-core Xeon 2.2 GHz system using an NVIDIA GeForce GTX 680, with 24 GB of system memory and 4 GB of GPU memory. We mostly report results from commodity hard drives, explicitly noting some specific relevant uses of SSDs. In many cases, results were obtained from multiple screen resolutions, but we report results from an HD viewport (1920×1080) unless noted otherwise. Details of the data utilized and renderer timings are given in Appendix A.

4.1 Benchmarks

We have chosen a variety of benchmarks to evaluate the performance of our renderer, and we elucidate the logic behind those choices here. First, the choice of HD resolution is motivated by voxel-to-pixel error ratios. All modern high-performance volume renderers try to maintain a 1-to-1 ratio between projected voxels and pixels. Adaptive resolution selection is used to ensure this ratio. Without this feature, results will be aliased, too much information will be compressed to a single pixel, and performance will suffer. Adaptive resolution means that small viewports will not stress renderers: a 512×512 viewport can get along fine with a paltry few hundred megabytes of memory, irrespective of the input dataset size.

We utilize zoom-ins, as in the accompanying video and results such as those in Figure 5 and some in Table 1, to accentuate these high resolution issues. When the volume is far away, a very coarse resolution is utilized which maintains accurate voxel-to-pixel error ratios. As the camera comes closer, higher resolutions of the source data must be utilized. We terminate zoom-ins slightly after they fill the screen; beyond this point, frustum culling's effect dominates (see Figure 5). The most challenging cases for a volume renderer are when data are close enough to be seen at native resolution, but far enough away that no data can be culled by the frustum.

Rotations are used to demonstrate that the renderer does not rely solely on early ray termination. As described in Section 3.3 and depicted in Figure 3, most rays either skip large parts of the volume, or terminate very quickly. With a transfer function that produces a dense volume, bricks in the front will prevent bricks in the rear from ever being paged in, effectively meaning the volume renderer need only cope with the front *half* or even less of the volume. Barring pathological volumes and transfer function choices, rotations ensure all of the data has a chance to contribute to a sequence.

Transfer functions. Changing a transfer function is also an important benchmark in any volume rendering system. Doing so invalidates our brick metadata concerning which bricks are empty, causing some hash table entries in the next frame to make little sense (i.e. request bricks which are visible under the old transfer function but empty under the new one). Furthermore, the bricks in the GPU volume pool may be inappropriate for the new transfer function.

Renderer performance as measured by response time during such an interaction actually changes very little, and can even improve. However, quality suffers rather drastically. This is evident in the time to convergence after a change in the transfer function: in a typical case with the RMI data set (see Section A), time to convergence increased over 6x after changing the transfer function (from $\sim$380ms to $\sim$2300ms).

4.2 Results

To evaluate our renderer in different scenarios, we used a standard rotation scenario with a variety of datasets, measuring the length of each pipeline stage. Figure 4 has these results. As IO is the prime bottleneck in many cases, we implemented a 'rebricking' scheme to mitigate the amount of IO performed. Using large reads and caching, this significantly lowers the time spent doing IO. We used 'LZ4'

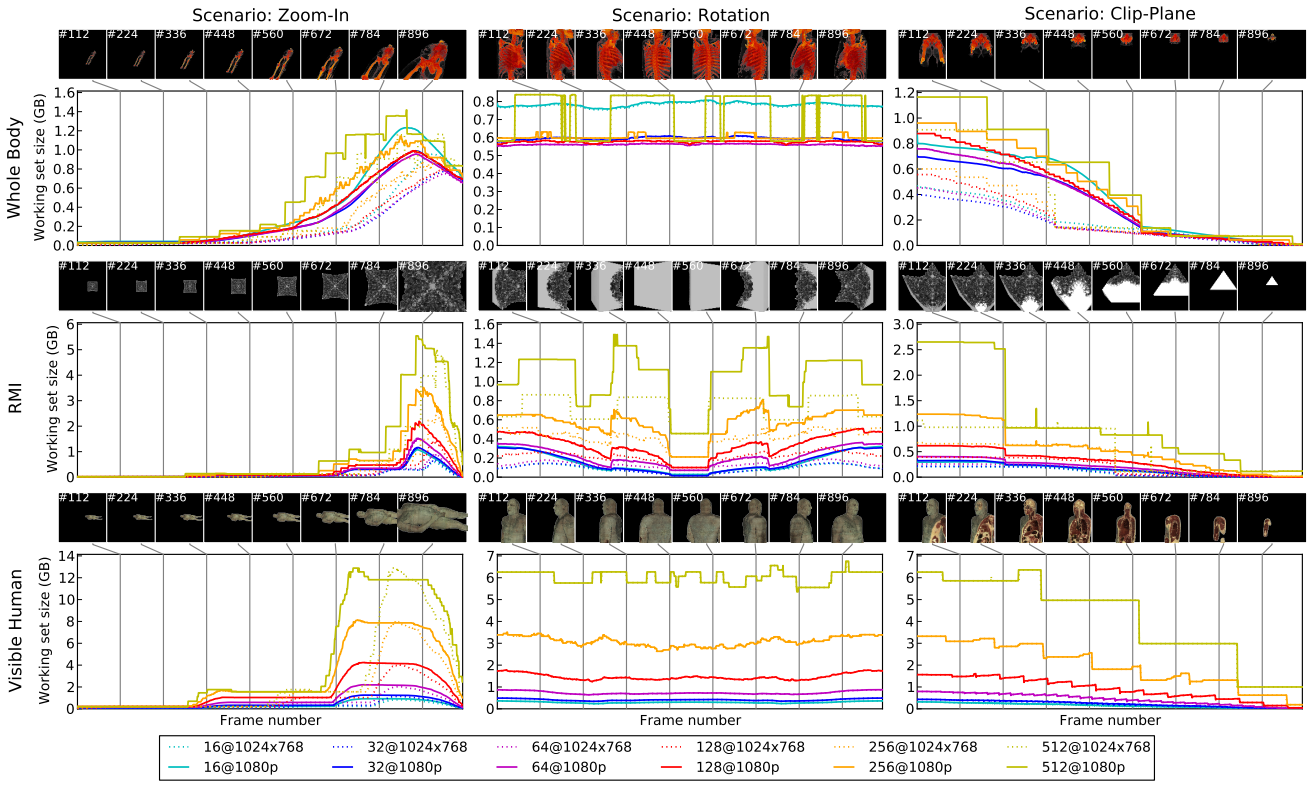

Figure 5: Working set sizes across three different scenarios for multiple datasets. Smaller brick sizes approximate the working set better.

compression when recording this performance data, which trades CPU time for IO time.

The majority of the time is spent ray-casting, pulling data from disk, and uploading the bricks to the pool. Our novel hash table approach keeps the table small, and so reading it is very cheap: even for large data, this component does not factor in to the overall performance. The other GPU data to manage is metadata information for our volume pool (i.e. which bricks are resident), but at a single machine word per brick it costs very little to push it down to the GPU, even for very large data.

Interestingly, the time spent managing GPU data is an increasing function of volume size *until* it peaks around the size of the RMI ($2048 \times 2048 \times 1920$). This reinforces our assertion that there is only so much data visible in a given frame—dependent only on the view frustum, and *not* the dataset size—and so at some point we saturate the set of visible data. Figure 5 and Section 5.1 include more discussion about working set sizes.

5 Design Tradeoffs

In this section, we try to explore aspects which have not been thoroughly addressed by previous literature. Details on trade-offs and the reasoning behind our final implementation choices are given.

5.1 Subdivision

How a system subdivides the volume into manageable pieces can have a large effect on the performance of the renderer. The primary considerations are in regard to early ray termination and empty space skipping: small bricks are much more likely to be composed of a small range or even uniform values, which will make it more likely that the brick can be skipped under a large set of transfer functions. Further, small bricks means one will detect ray saturation much more quickly, as this is checked only when exiting a brick.

Internal Overhead The primary drawback is reduced disk throughput due to utilizing many small requests. A further drawback is the data size overhead: each brick needs two voxels of ghost data in each dimension, for sampling and gradient computation purposes. This is negligible for large bricks, but grows sharply as the brick size approaches one, as shown in Figure 6. Figure 8 demonstrates that this is not strictly a theoretical result: a small brick size greatly increases not just size overhead, but also the time to reorganize the data on disk. From these Figures we can derive that for large datasets a brick size of less than 16^3 is impractical.

External Overhead We have performed a number of experiments to identify the working set size for multiple different brick sizes. Starting with the smallest practical size of 16^3, we increase the brick size up to 512^3.

As can be seen in Figure 5 the working set is bound not by just the data size, but the screen resolution as well. It can also be seen that the brick size heavily influences the working set size: larger bricks allow for less efficient utilization of empty regions. From the images we can derive that a brick size of less than 128^3 is desirable to reduce the working set to roughly the memory size of a GPU.

We note that the working set size is not a strict function of the brick size, however. Figure 5 and Table 1 also show that the choice of brick size is not clear-cut. The Visible Human male performs best with 16^3 bricks, for example, whereas the ideal brick size for the 'Magnitude' data is 64^3. For the 'Whole Body' dataset, using brick sizes of 16^3 actually resulted in *larger* working sets than 32^3. This occurs when the transfer function produces large regions of semi-transparency but never reaches saturation. Indeed, when datasets contain large swaths of semi-transparent regions, the conventional wisdom is reversed: large brick sizes are generally preferred, since they significantly improve disk throughput.

If we begin to consider secondary metrics, such as the response time of the system, the choice of brick size becomes even more

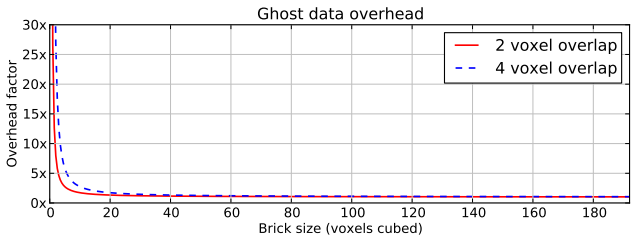

Figure 6: Brick size overhead. As bricks get smaller, the overhead for the additional ghost data grows significantly. At a larger brick size of 128^3, the overhead with 2 ghost voxels per dimension amounts to a few percent, whereas with 32^3 bricks this increases the dataset size by almost 50%.

complex. Since bricks are the atomic building blocks in a volume renderer, one cannot load less than a single brick from disk. Therefore a larger brick size imposes a larger response time on the system. These concerns would generally push a designer to choose smaller bricks.

However, disk performance falls very sharply with small requests [27]. It is nice for a system to respond within a few tens of milliseconds, but such concerns should not dictate the design to the point that end-to-end performance suffers drastically. Furthermore, small brick sizes are accompanied with significant overhead, as discussed in Figure 6, and do not compress as effectively as their larger counterparts.

Systems such as Reichl et al.'s hybrid surface rendering, CERA-TVR, and Gigavoxels utilize a static brick size of 32^3 [4, 26, 3]. This brick size exhibits few extremes of the performance issues mentioned above. However, it is certainly not the ideal choice for all circumstances.

5.2 Disk IO

5.2.1 Brick Layout

Figure 7 demonstrates how this changes with the brick size. Both disk IO times as well as decompression times are displayed there. As shown in the figure, reading data from disk becomes quite severe with small brick sizes. However, as brick sizes grow to 64^3 and beyond, decompression time becomes more important and overall time plummets. This effect is even more pronounced using a hard disk in place of the SSD used here. Intelligent layout strategies purport to minimize seek times; our results corroborate this, with the important caveat that seek times are not relevant with larger brick sizes.

5.2.2 Dynamic Rebricking

The renderer desires small bricks, as discussed in Section 5.1, as small bricks will help with early ray termination and empty space leaping. However Figures 7 and 6 clearly demonstrate that large brick sizes are preferable for disk performance and overhead reasons. To provide the best of both worlds, we implemented a 'dynamic' bricking scheme, whereby bricks are stored on disk in a rather large size (e.g. 256^3) but presented to the renderer as if they exist at some small resolution (32^3). The small bricks are dynamically generated from the large ones on request.

Since requesting a large brick for every small brick would only increase the disk traffic, we keep an additional brick cache in memory to source these copies from. Our cache uses a standard LRU strategy. This is advantageous when the working set of the data fits into the host memory, however when the working set exceeds the host memory we will evict entries before finishing a rendering. We stuck with this strategy since the working set often *does* fit into host memory, as established by Figure 5. If the renderer is to be used

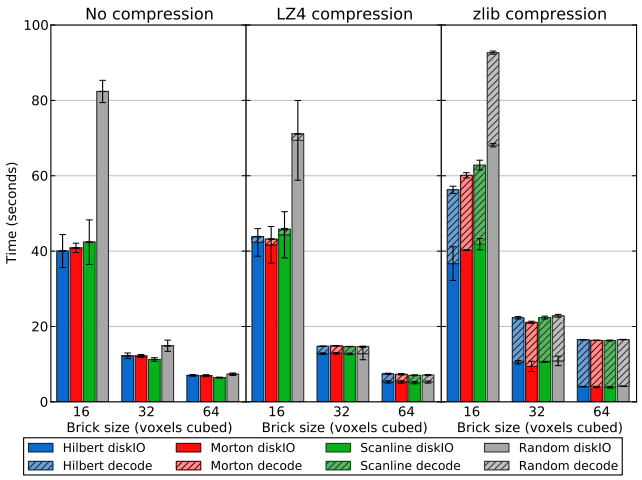

Figure 7: Time spent with IO-related tasks using an SSD for the RMI dataset's zoom-in scenario, sampled with 100 frames and a 1024×768 viewport. Layout strategies only see utility at small brick sizes.

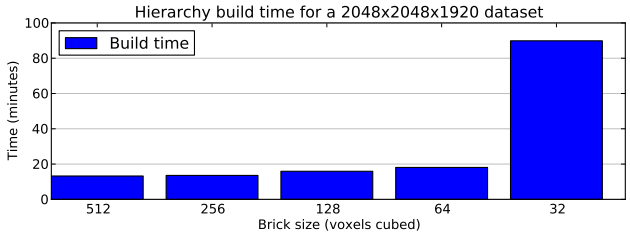

Figure 8: Time to build bricked representation for a medium-sized dataset, as a function of brick size. Renderers desire small bricks to perform efficiently, but generating such bricks takes significant preprocessing resources.

in an environment in which working sets are routinely larger than memory, an MRU strategy would be more appropriate.

Hierarchy Generation Reorganizing data into a set of bricks is mostly ignored in volume rendering literature, but becomes a significant bottleneck in real-world usage. Figure 8 shows the time our preprocess needs to generate this hierarchy, which increases sharply for small brick sizes. This time also increases with respect to dataset size. At the extreme scale, such data reorganization is completely infeasible: merely reading every datum might take months. We believe such reorganization will be feasible up to a few tens of terabytes. In practice, the authors and collaborators thereof tolerate this for up to 4 terabytes at present.

Rebricking the data at run time alleviates this problem. The data can be generated at very large brick sizes, enabling fast conversion and effective disk throughput, and then dynamically rebricked to very small sizes. Both disk and renderer deal with their ideal cases, then. The 'Rebricking' case of Figure 4 shows performance in this mode.

5.3 CPU/GPU Interface

Point (4) in our overview is the efficient communication of the ray guidance information from the location it is generated—the GPU—to the location it is utilized—the IO layer of a volume renderer. This section details how that communication happens.

We utilize a GPU-based hash table to store this data, though we note that we really only require a set. That is, our keys (brick IDs) are our values, and we only care about their *presence* in the table,

Algorithm 2 Greedy algorithm: request all bricks at all resolutions.

```
ReportMissingBrick(b)
repeat
    LoD++
    b = LookupBrick(ray, LoD)
    if Missing(b) then
        ReportMissingBrick(b)
    end if
until ¬Missing(b)
```

Algorithm 3 Global algorithm: only request bricks required to satisfy the final rendering request.

```
ReportMissingBrick(b)
repeat
    LoD++
    b = LookupBrick(ray, LoD)
until ¬Missing(b)
```

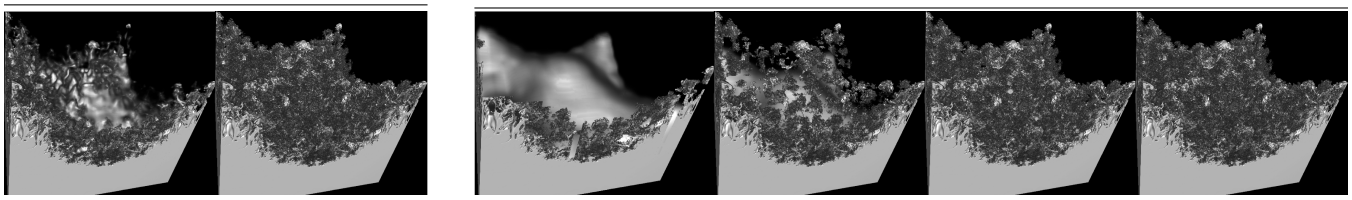

Figure 9: The effect of multiple brick replacement strategies. Renderings are select intermediate frames from the corresponding strategy. 'Greedy' strategies converge quicker and produce more densely-packed intermediate progress.

which we will read back and process as a list later. A list would work as well, but a hashing scheme allows concurrent inserts to proceed with less contention. During rendering, a ray may write into this table to indicate that it needs a non-resident brick to continue (see Figure 2, (c)). This small table will be read back from the GPU at the end of a frame and utilized to fill the volume pool with new data.

As locks do not exist in current GLSL versions (and potentially never will), lock-free structures are the only hazard-prone data structures which can be correctly implemented. Crassin et al.[3] workaround this by using multiple render targets: each pixel has its own unique set of memory to write into, and so there are no write hazards. Our scheme requires significantly less memory, but we must deal with these write hazards.

5.3.1 Hash Table Parameters

We map from the 4D index of the requested brick (spatial index + LoD) to a unique 1D index in the hash table. The mapping we utilize is simply converting the 4D index into its equivalent 1D form, as if it were stored in a 1D array. We increment the index by 1 so that we may use 0 to indicate that there is no entry at a location.

In a normal concurrent hash table, a lock is acquired for a table or bucket before an access. In lock-free data structures the primitives used to implement locks are instead used directly on the data values in question. Inserts into our table proceed mostly as described in previous work [24]. In the face of concurrent writes, this operation fails, and we attempt to probe a few times (presently: 10) before giving up.

The critical piece to note is: *it is not an error if a missing brick is not recorded*. As long as *some* missing bricks are recorded, the next frame *will* make progress. Each ray is either: finished, able to make progress, or unable to make progress due to a lack of bricks that it requires. Since our hash table only contains entries for bricks which were requested by a ray, then an invariant of our system is that: volume rendering is done, or there exists at least one ray which can make progress.

5.3.2 Strategies for Loading Coarser Bricks

When the resolution required is missing during ray-casting, a ray's brick requests can be what we call 'greedy' or 'global'. In the 'greedy' case, the ray requests intermediate levels of detail along the way, flooding the hash table with requests that *this* ray wants. In the 'global' case, each ray only requests what it absolutely needs, leaving space for other rays to request what they need. These cases are visually depicted and expounded in Figure 9.

The intuitive interpretation is that the 'greedy' approach will produce a more responsive, iteratively-refined image, whereas the 'global' approach will generate the final correct image quickest. However, the authors were surprised to find that the 'greedy' approach both produces more pleasing progress information *and* converges in the fewest number of frames. This is because it allows a ray to sample at its final resolution quickly, which can cause earlier ray termination.

6 Conclusions, Limitations, & Future Work

In this work, we have introduced an efficient, out-of-core, ray guided GPU volume renderer which scales to extremely large data. The system pulls inspiration from a patchwork of recent renderers, combining the advantages of many and reimplementing some ideas in light of modern GPU features. We have also contributed an evaluation and discussion of the tradeoffs inherent in the development of a modern ray-guided volume renderer.

Based on the data here, we conclude that a ray-guided volume renderer should work with bricks which are, on disk, 64^3 or larger. This minimizes time spent doing IO (Figure 7), and makes data layout irrelevant, obviating the need for a complicated component of the code. Since the required memory shrinks with the brick size, generating 32^3 or even 16^3 bricks on-the-fly is desirable, though which size is unfortunately too data-specific to answer generally. While 'bzlib' gives ideal compression ratios, it is very slow to decompress, and therefore most implementations will want to utilize 'LZ4' compression. A cache is a boon when data will not fit in GPU memory but will fit in the host's memory.

We have made a best-effort attempt to design both favorable and unfavorable conditions with which to test a volume renderer, but it is possible some considerations have been omitted. In particular, this renderer and many others rely heavily on the assumption that rays will saturate quickly. Subjectively, we have found this to be overwhelming valid for all our work in volume rendering, but this is not a rule and has not been thoroughly evaluated.

A second issue is the rendering modes evaluated. While our system supports 2D transfer functions as well, all performance results presented here utilized the 1D transfer function mode. Advanced rendering effects as well, such as those similar to ambient occlusion [28], are omitted. Such effects should have a variable impact,

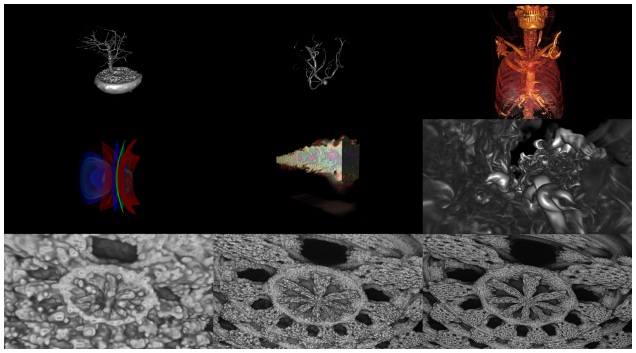

Figure 10: Selected frames from interactions used to record data for Table 1 or Figure 4

positively correlating to the proportion of rendering vs. IO times presented in Figure 4. Screen-space methods may provide acceptable quality without (comparatively) impacting performance.

Finally, reformatting the data into a bricked hierarchy continues to be the bane of high-performance volume rendering. This result is not expounded often enough in the literature. We hope this paper helps to reiterate to the community that the FLOPs may be free, but data movement will kill performance.

More importantly, we have contributed an evaluation and discussion of the issues inherent in the development of a ray guided volume renderer. As has been demonstrated, many of these choices are not as clear as previous reports may have inadvertently implied. The results presented in this work clearly depict the tradeoffs, to aid system designers in creating volume renderers which suit their particular environment.

We hope to extend this work to more diverse visualization scenarios. Ray-guidance-based isosurface generation is a natural candidate for these ideas. Furthermore, a common use case is combining an isosurface with volume rendering, which has the potential to significantly change such aspects as the working set size. The general idea that rendering should drive the visualization pipeline—as opposed to passively consuming the output of earlier operations—is one that is applicable in a much wider sense than that presented here.

A Data and Performance Details

We tested our renderer with a plethora of datasets, both real and artificially created. For space reasons, we discuss only a subset which proved to be a reasonable sampling of our available data. Renderer performance is depicted for a variety of datasets in Table 1. We discuss these in order of increasing size here.

Two small datasets are the Bonsai tree ("Bonsai") and "Aneurysm" datasets (Figure 10, top, left & middle). While small by today's standards, effective empty space leaping and early ray termination still double the performance (Table 1, note how performance doubles with smaller brick sizes).

The "WholeBody" dataset (Figure 10, top, right) is a contrast-enhanced CT scan of a human body. As sometimes happens in the biomedical domain, these data have limited slice resolution but a plethora of slices. Coarser resolutions must be careful to downsample anisotropically, else the in-plane resolution washes out too quickly.

"Velocity" (center, left) comes from the simulation of an exploding star; we chose this dataset because our ideal transfer function for it is quite transparent, preventing the renderer from taking advantage of early ray termination. Highly transparent transfer functions which still produce informative results are a rarity but still occur. For these data, the additional overhead of small bricks can have a fairly drastic effect on performance. This dataset is one of the rare datasets for which lighting actually makes the visualization more *difficult* to interpret, and so we always render this dataset with lighting off.

Table 1: Per-frame rendering time at 6 different brick sizes, for a variety of datasets depicted in Figures 10 and 1. **Optimal brick sizes** are dataset dependent.

	Rendering Time (ms)				
Dataset	16^3	32^3	64^3	128^3	256^3
Bonsai	**16**	20	26	31	28
Head Aneurysm	**27**	34	40	55	85
Whole Body	140	94	82	77	**67**
Velocity	376	208	146	118	**110**
Magnitude	132	93	**80**	82	85
RMI	**60**	64	61	67	67
Visible Human	**34**	37	47	67	123
Mandelbulb1k	**21**	**21**	**21**	22	25
Mandelbulb4k	**27**	30	37	47	47
Mandelbulb8k	**33**	37	45	60	78

Table 2: Dataset properties for test datasets.

Dataset	**Resolution**		**Size**
Bonsai	256 × 256 × 256	8 bpp	16 MB
Head Aneurysm	512 × 512 × 512	16 bpp	256 MB
Whole Body	512 × 512 × 3172	16 bpp	1.5 GB
Velocity	1000 × 1000 × 1000	16 bpp	1.9 GB
Magnitude	2025 × 1600 × 400	16 bpp	2.4 GB
RMI	2048 × 2048 × 1920	8 bpp	7.5 GB
Visible Human	1728 × 1008 × 1878	32 bpp	12.2 GB
Mandelbulb1k	1024 × 1024 × 1024	8 bpp	1 GB
Mandelbulb4k	4096 × 4096 × 4096	8 bpp	64 GB
Mandelbulb8k	8192 × 8192 × 8192	8 bpp	512 GB

The "magnitude" dataset (center, middle) comes from a combustion simulation and represents another intermediate step towards larger data. The lower half of this dataset actually has a very faint trace of data, which causes the renderer to sample densely. The expense of computing lighting information for fragments which ultimately contribute very little has a notable effect on performance.

The Richtmyer-Meshkov Instability ("RMI", Figure 1 right and Figure 10, center, right) and the Visible Human (Figure 1 left) are popular datasets in the volume rendering literature; details can be found in previous work.

We created a series of "Mandelbulbs" at various resolutions ($1k^3$, $4k^3$, $8k^3$). These are an extension of the mandelbrot fractal into 3 dimensions. This has many of the same properties of the data used in Crassin et al. [3], which took a large bone scan and added Perlin noise to increase the sampling requirements. We create the high-resolution features *a priori*, so no GPU features were used to accelerate this process. At equivalent resolutions to that work, we see double to an order of magnitude improved performance, but for this work we report results at 1080p HD resolution. A descriptive view of the Mandelbulb is given in Figure 3 and there are close-ups visible in Figure 10 (bottom row; center, right).

B Source Code

The renderer used in this work is freely available, as part of the ImageVis3D [16] package, which can be obtained through a simple web search. We encourage others to reproduce and build upon our results.

Acknowledgments

This research was made possible in part by the Intel Visual Computing Institute; the NIH/NCRR Center for Integrative Biomedical Computing, P41-RR12553-10; and by Award Number R01EB007688 from the National Institute Of Biomedical Imaging And Bioengineering. The content is under sole responsibility of the authors.

REFERENCES

[1] Aaron Knoll, Sebastian Thelen, Ingo Wald, Charles D. Hansen, Hans Hagen, and Michael E. Papka. Full-resolution interactive cpu volume rendering with coherent bvh traversal. In *Proceedings of the 2011 IEEE Pacific Visualization Symposium*, pages 3–10, 2011. http://dl.acm.org/citation.cfm?id=2015627.

[2] Markus Hadwiger, Johanna Beyer, Won-Ki Jeong, and Hanspeter Pfister. Interactive volume exploration of petascale microscopy data streams usign a visualization-driven virtual memory approach. In *Proceedings of IEEE Visualization 2012*, 2012.

[3] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, Boston, MA, Etats-Unis, feb 2009. ACM, ACM Press. http://maverick.inria.fr/Publications/2009/CNLE09.

[4] F. Reichl, M. G. Chajdas, and R. Westermann. Hybrid sample-based surface rendering. In *Proceedings of the workshop on Vision, Modeling, and Visualization (VMV 2012)*, 2012.

[5] Christian Dick, Jens Krüger, and Rüdiger Westermann. GPU ray-casting for scalable terrain rendering. In *Proceedings of Eurographics 2009 - Areas Papers*, pages 43–50, 2009.

[6] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: A general purpose ray tracing engine. *ACM Transactions on Graphics*, August 2010.

[7] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[8] T.J. Cullip and U. Neumann. Accelerating volume reconstruction with 3D texture hardware. Technical Report TR93-027, University of North Carolina, Chapel Hill N.C., 1993.

[9] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings ACM Symposium on Volume Vis*, pages 91–98, 1994.

[10] Jens Krüger and Rüdiger Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings IEEE Visualization 2003*, 2003. http://wwwcg.in.tum.de/Research/data/vis03-rc.pdf.

[11] Imma Boada, Isabel Navazo, and Roberto Scopigno. Multiresolution volume visualization with a texture-based octree. *The Visual Computer*, 17(3):185–197, 2001. http://vcg.isti.cnr.it/publications/papers/vc17_3.pdf.

[12] Eric C. LaMar, Mark A. Duchaineau, Bernd Hamann, and Ken Joy. Multiresolution techniques for interactive texture-based volume visualization. In *Visual Data Exploration and Analysis VII*, volume 3960, pages 365–374, Bellingham, Washington, 2000. SPIE, The International Society for Optical Engineering. http://dl.acm.org/citation.cfm?id=834140.

[13] Manfred Weiler, Rüdiger Westermann, Chuck Hansen, Kurt Zimmerman, and Thomas Ertl. Level-of-detail volume rendering via 3d textures. In *Proceedings of the 2000 IEEE symposium on Volume visualization*, VVS '00, pages 7–13, New York, NY, USA, 2000. ACM. http://doi.acm.org/10.1145/353888.353889.

[14] Marc Levoy. Efficient Ray Tracing of Volume Data. *ACM Trans. Graph.*, 9(3):245–261, 1990. http://doi.acm.org/10.1145/78964.78965.

[15] Jennis Meyer-Spradow, Timo Ropinski, Jörg Mensmann, and Klaus Hinrichs. Voreen: A Rapid-Prototyping Environment for Ray-Casting-Based Volume Visualizations. *IEEE Computer Graphics and Applications*, 29(6):6–13, 2009. http://dx.doi.org/10.1109/MCG.2009.130.

[16] Thomas Fogal and Jens Krüger. Tuvok, an Architecture for Large Scale Volume Rendering. In *Proceedings of the 15th International Workshop on Vision, Modeling, and Visualization*, November 2010. http://www.sci.utah.edu/~tfogal/academic/tuvok/Fogal-Tuvok.pdf.

[17] Thomas Fogal and Jens Krüger. Size Matters - Revealing Small Scale Structures in Large Datasets. In Wolfgang Schlegel Olaf Dössel, editor, *Proceedings of World Congress on Medical Physics and Biomedical Engineering*, pages 41–44. Springer, 2009.

[18] Enrico Gobbetti, Fabio Marton, and José Antonio Iglesias Guitián. A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer*, 24(7):797–806, July 2008. http://dx.doi.org/10.1007/s00371-008-0261-9.

[19] Imma Boada, Isabel Navazo, and Roberto Scopigno. Multiresolution volume visualization with a texture-based octree. *The Visual Computer*, 17(3):185–197, May 2001.

[20] Hank Childs, Mark Duchaineau, and Kwan-Liu Ma. A scalable, hybrid scheme for volume rendering massive data sets. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization*, pages 153–162, May 2006. http://www.idav.ucdavis.edu/publications/print_pub?pub_id=892.

[21] Mark Howison, E. Wes Bethel, and Hank Childs. MPI-hybrid Parallelism for Volume Rendering on Large, Multi-core Systems. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, Norrköping, Sweden, May 2010. LBNL-3297E.

[22] Thomas Fogal, Hank Childs, Siddharth Shankar, Jens Krüger, R. Daniel Bergeron, and Philip Hatcher. Large Data Visualization on Distributed Memory Multi-GPU Clusters. In *Proceedings of High Performance Graphics 2010*, 2010.

[23] J. Beyer, M. Hadwiger, J. Schneider, W.-K. Jeong, and H. Pfister. Distributed terascale volume visualization using distributed shared virtual memory. Poster at IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV), 2012.

[24] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the ACM symposium on Parallel algorithms and architectures*, pages 73–82, New York, 2002. ACM. http://dl.acm.org/citation.cfm?id=564870.564881.

[25] Christopher Lux and Bernd Fröhlich. GPU-based ray casting of stacked out-of-core height fields. In *ISVC'11: Proceedings of the 7th international conference on Advances in visual computing*. Springer-Verlag, September 2011.

[26] Klaus Engel. CERA-TVR: A framework for interactive high-quality teravoxel volume visualization on standard pcs. Poster at IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV), 2012.

[27] Thomas Fogal and J. Krüger. Efficient I/O for Parallel Visualization. In *Eurographics Symposium on Parallel Graphics and Visualization, Llandudno, Wales, UK*, pages 81–90, April 2011.

[28] Matthias Schott, Vincent Pegoraro, Charles Hansen, Kevin Boulanger, and Kadi Bouatouch. A Directional Occlusion Model for Interactive Direct Volume Rendering. In *EG Symposium on Visualization (IEEE-VGTC '09)*, volume 28, 2009.