Eurographics Symposium on Parallel Graphics and Visualization (2022)
R. Bujack, J. Tierny, F. Sadlo (Editors)

A Flexible Data Streaming Design for Interactive Visualization of Large-Scale Volume Data

Qi Wu*[1], Michael J. Doyle†[2], and Kwan-Liu Ma‡[1]

[1]University of California - Davis, USA
[2]Intel Corporation

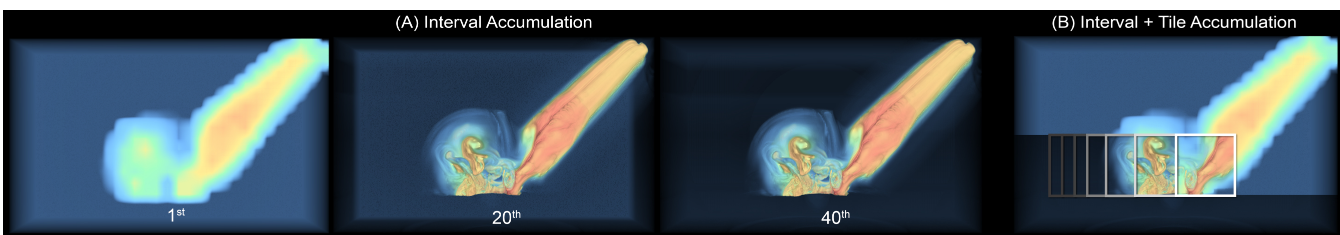

Figure 1: *Large-scale progressive volume rendering of the deep ocean water asteroid impact dataset [PG17]. A) In our system, the progressive rendering is done by breaking the volume interval into smaller segments, and only compute one segment per frame. B) Additionally, our system can also break the framebuffer into smaller tiles, and only render one tile at a time. Both method allows our rendering system to significantly reduce memory footprints.*

Abstract

Modern simulations and experiments can produce massive amounts of high-fidelity data that are challenging to transport and visualize interactively. We have designed a data streaming system to support interactive visualization of large volume data. Our streaming system design is unique in its flexibility to support diverse data organizations and its coupling with a highly efficient CPU-based ray-tracing renderer. In this paper, we present our streaming and rendering design and demonstrate the efficacy of our system with progressive rendering of streaming tree-based AMR (TAMR) volume data and radial basis function (RBF) particle volume data. With our system, interactive visualization can be achieved using only a mid-range workstation with a single CPU and a modest quantity of RAM.

1. Introduction

Modern simulations and experiments can produce massive amounts of high-fidelity data given continuous advancements in computation and data acquisition technologies. These data can provide critical details for studying complex physical phenomena and chemical processes, but they can also be challenging to visualize interactively without using purpose-built hardware such as a PC cluster. The possibility of visualizing such large volume data in more resource-limited environments such as a single workstation or a laptop is always desirable, as it promises to lessen or eliminate the need for expensive and specialized equipment and makes visualization vastly more accessible. However, achieving this ability requires solving two fundamental issues: 1) Large datasets can make it challenging to maintain sufficient bandwidth to utilize available compute resources adequately; 2) Even if sufficient bandwidth to the compute resources can be established, the parallel design of most modern processors necessitates careful workload distribution to fully utilize the available compute power.

The first issue can manifest in several ways. A high precision dataset can easily exceed the RAM capacity, causing problems with both loading and rendering the data. Even if the data can fit in the RAM, the large working set can degrade the system performance (e.g., spending a lot of time waiting for I/O) or create difficulties in a multi-user environment where multiple applications compete for resources. To alleviate these problems, an effective solution is out-of-core processing by streaming data on-demand from disk in real-time. Data streaming occurs when independent subsets of larger data are loaded and processed at different times. This approach is useful when we cannot fit the entire dataset in RAM, or if we otherwise wish to limit RAM usage. However, to maintain interactivity

*e-mail: qadwu@ucdavis.edu
†e-mail: michael1.doyle@intel.com
‡e-mail: ma@cs.ucdavis.edu

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

DOI: 10.2312/pgv.20221064

with data streaming methods, it is often necessary to leverage additional techniques such as level-of-detail and progressive rendering.

For the second issue regarding parallelization, two approaches are often used. Task parallelism arranges independent visualization subtasks that operate on the same data using multiple processors. This technique is popular for rendering acceleration (e.g., multi-threaded rendering). A pipelined approach splits a visualization task into several parallelizable stages working on different data subsets. This approach allows better utilization of hardware resources and can overlap the execution time of different stages (e.g., rendering and the data loading). However, to prevent a stage from becoming a bottleneck, the execution times of different stages need be balanced carefully.

In this work, we contribute a flexible data streaming design, which addresses the challenges outlined above. Our goal is to enable interactive visualization of large-scale volume data on mid-range personal computing devices with limited memory and computing budgets. Whereas previous streaming systems were optimized for one particular type of data structure and it is unclear how to adapt them to new data structures, our design can be applied to different types of data structures (e.g., octree, BVH) with minimal implementation overhead (Section 4). In particular, to handle large-scale volume data, we introduce a generalized tree-based multiresolution representation of the data, on top of which a well-engineered CPU-based interactive and progressive ray tracing system is built to make possible utilization of any computer systems. Our design is also different from the delayed loading approach used by others such as BrickTree [WWJ19] and OpenVDB [Mus13], because voxels accessed by the visualization are continuously loaded from secondary storage and cached in system RAM appropriately. The resulting system produces a fixed memory footprint and simultaneously takes advantage of data streaming, task parallelism, and pipeline parallelism in a unified fashion. We demonstrate our design by applying it to rendering tree-based AMR (TAMR) data as well as radial basis function (RBF) particle data, and we release our implementation at https://github.com/wilsonCernWq/openvkl-streaming.

2. Related Work

In this section, we review existing methods for rendering large-scale volumes. We first go over the history of large-scale volume rendering using data streaming, followed by a review of related work using non-data-streaming methods with a focus on overcoming the I/O challenges of large-scale data. Finally, we detail existing solutions introduced particularly for TAMR and RBF particle data.

2.1. Data Streaming for Large Scale Volumes

Volume rendering with data streaming is not a new topic. A straightforward implementation would divide the original dataset into pieces and process one piece at a time [LSMT99, BHP15]. This method is also used by several general-purpose visualization frameworks such as VTK [SM05], probably due to its simplicity and generalizability. However, the main drawback of this approach is that more rendering passes are needed for each frame, making interactivity more difficult to maintain. LaMar et al. [LHJ00] and Weiler et al. [ZWE*00] were among the first to combine data streaming and LoD for rendering large scale volume data. In particular, their approaches convert the volume into a set of multi-resolution textures that fit entirely in GPU texture memory, and then leverage LoD for adaptive resampling to minimize data lookups. Gobbetti et al. [GMG08] later offered a similar method for rendering out-of-core volumes on GPUs using hierarchical representations. However, their hierarchical volume representations are built on the CPU using octree-like structures. This design was further improved by GigaVoxels [CNLE09] using ray-guided streaming for rendering voxel surfaces. This improvement can optimize GPU to CPU communication and thus enhance interactivity. The CERA-TVR [Eng11] system later generalized GigaVoxels to volume visualization and supported both LoD and progressive rendering. It offered progressive rendering through image-space refinement, allowing high-quality pixels to be computed by successive frames. The *ImageVis3D/Tuvok* system [FK10, FSK13] also combined data streaming with both LoD and progressive rendering for large-scale interactive volume visualization. In this system, LoD is enabled during interactive explorations, while progressive rendering is used when high-quality images are needed. However, their progressive rendering algorithm is coupled with data streaming; thus, the size of the data subset loaded and rendered in each frame is controlled by the RAM capacity and affected by the rendering budget. They also introduced a more efficient data caching mechanism for regular volumes that combines the Most Recently Used (MRU) strategy and the Least Recently Used (LRU) strategy at the same time. The work done by Hadwiger et al. [HBJP12] leveraged previous work and presented a new approach to address and stream petascale imaging volumes from GPU. Sarton et al. [SCRL19] later generalized this approach to all regular volumes. In their approach, a new multi-resolution texture-based volume representation was introduced, analogous to virtual memory and hardware page caches. Data chunks are loaded and cached if the corresponding virtual address is visited by rays, the resolution of the data chunk is controlled adaptively using LoD. This allows their approach to scale better for extremely large uniform grids compared with the GigaVoxels and CERA-TVR systems.

2.2. Non Data Streaming Methods

For high-end platforms with enormous RAM capacity, data streaming is not strictly needed. Therefore, volume renderers just focusing on asynchronous data loading can also be beneficial. Due to the scope of this work, we only review methods leveraging on-demand data loading, meaning that data I/O and the volume rendering processes are largely overlapped. OpenVDB [Mus13] is a hierarchical data structure heavily used in the industry for representing sparse dynamic volumes. It uses a B+tree to exploit spatial coherence and efficiently encodes data values and topologies. It also implements pipeline parallelism through deferred loading, which means that data values are only loaded as needed during rendering. However, their implementation of deferred loading can only work on voxel buffers (i.e., leaf nodes) and this functionality is not implemented in its successor, NanoVDB [Mus21]. The Bricktree method presented by Wang et al. [WWJ19] provides another general algorithm to overlap rendering and I/O. Their work adopted an octree structure with branching factor N^3 and can also use ray-guided stream-

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

ing to avoid loading invisible data chucks. Wang et al. also provided an efficient implementation to render their data structures utilizing multithreading and SIMD vectorization in OSPRay [WJA*16]. However, their method does not support data eviction. Thus, the machine RAM capacity must be sufficient to hold the entire dataset to achieve good performance.

2.3. TAMR and RBF Volume Rendering

Adaptive Mesh Refinement (AMR) data uses multi-resolution grids to represent computational domains for more efficient allocation of compute and memory resources. Particularly, an AMR data can be block-structured (SAMR) [BC89] or tree-based (TAMR) [BO84]. SAMR grids are stored as a hierarchy of overlapping and successively finer uniform grids. In contrast, TAMR grids are represented as sparse octrees and can allow for more economical use of memory resources for large-scale datasets [DL19]. For rendering TAMR data specifically, Labadens et al. [LCPT12] used tree structures to generate volume splats for direct volume ray tracing. However, their approach only allows nearest-neighbor interpolation. Leaf et al. [LVI*13] improved this by leveraging the multiresolution interblock interpolation of Ljung et al. [LLY06]. However, their method requires a distributed computing cluster to handle massive TAMR data. For CPU-based ray tracing, Wang et al. [WMU*20] developed high-quality reconstruction filters within OSPRay [WJA*16] to directly operate on TAMR grids.

Smoothed particle hydrodynamics (SPH) [Mon92] is a mesh-free method for simulating motion of fluids in space. The resulting particle data can be rendered by constructing a scalar field composed of additively weighted SPH kernels [KJM21]. Because this technique allows particles to be directly rendered without simplifications, it was also generalized to render other particle data, often referred as RBF volume rendering [KWN*14]. For large scale RBF volume rendering, Jang et al. [JFSP10] contributed a KD-tree based method. Fraedrich et al. [FAW10] used an octree hierarchy to dynamically resample particles into perspective space uniform grids of predetermined size for interactive visualization. Reda et al. [RKN*13] developed a uniform grid based acceleration structure for volume ray casting in GPU. Knoll et al. [KWN*14] employed coherent bounding volume hierarchy (BVH) traversal to efficiently evaluate the RBF field, which eliminates the need for costly per-ray neighbor search, and repeated queries of the same basis functions at different samples.

3. Overall Design

Our design (Figure 2) consists of three primary parts: The *renderer*, which is responsible for determining the location of samples and implementing real-time LoD and progressive rendering (Section 3.3); the *hierarchical abstraction*, which implements a generalized structure for representing the volume and algorithms for traversing the volume (Section 3.1); and the *data management system*, which maintains I/O requests, address translation and data caching, and generates streaming requests. This design structure was chosen to support our goal of a flexible data streaming system; when applying our design to a new data structure, only the *hierarchical abstraction* needs to be slightly extended in order to support the new structure.

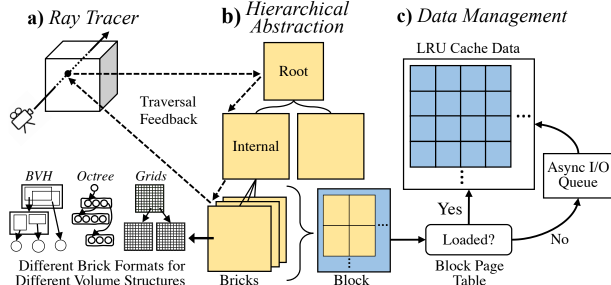

Figure 2: *The overall system. a) A progressive renderer, which communicates with the hierarchical abstraction via traversal feedback. b) A hierarchical data abstraction to represent volumes. The fundamental unit of this abstraction is referred to as a* ***brick****, and a group of bricks loaded simultaneously are referred to as a* ***block****. c) A data management system for SIMD vectorized LRU caching, page table lookup, and asynchronous streaming.*

3.1. Hierarchical Abstraction

The hierarchical abstraction provides a flexible structure for representing volume data in our streaming-based visualization system. We refer to the fundamental unit of data in this representation as a *brick*, which represents a set of contiguous voxels of the same resolution (shown as yellow boxes in Figure 2b). The meaning of a brick will vary depending on the underlying volume data structure. For example, for octree-based volumes, a brick can represent an octant, with the root being handled differently. For binary-tree-based volumes, a brick can represent a tree node.

A key feature of our abstraction is that multiple such bricks can be nested to form a tree hierarchy, where deeper levels of the hierarchy represent successively higher resolution representations of the underlying data. The number of bytes used by each brick should be upper-bounded. This upper bound should be small to provide finer granularity for the abstraction. Then we group hundreds of bricks together and stream them simultaneously. Such brick groups are referred as *blocks* (shown as blue boxes in Figure 2). These blocks are always aligned with a fixed data size (referred to as the block size). If a block is not 100% filled, zeros are padded.

To preserve the generalizability of our design, we do not enforce a universal brick format. Instead, a *brick format* and the corresponding *brick accessor* should be defined for each volume data structure. There are two assumptions for how brick formats should be designed. First, it must be possible to compute approximated data values for samples taken within the spatial region of the brick using only information stored in the brick (i.e. it may not refer to data stored in child bricks for this operation). This assumption enables us to implement LoD and progressive rendering. Second, multiple classes of bricks with different data sizes can be defined for a brick format(e.g., internal bricks and leaf bricks), but bricks that belong to the same class should contain the same number of bytes. This allows us to better optimize data streaming. In Section 4, we describe more details on how to create brick formats for a selection of distinct volume data structures. Having bricks to follow cache line alignment can be beneficial as well, but it is controlled by the underlying brick format and it does not affect the correctness of our design.

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

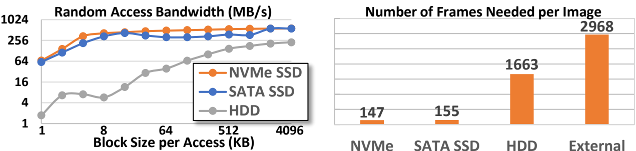

Figure 3: *Left: Comparison of various hard drives'* ***random access performance****. Even for NVMe SSDs (Intel 660p 1TB), random streaming requests smaller than 4KB lead to significant I/O overhead. Right: comparison of the number of frames required by our progressive rendering system to complete an image with the highest accuracy when the data is located on different devices ("External" means an external USB 3.0 HDD); Results obtained by rendering the Disney Cloud using our progressive treelet method (Section 5).*

3.1.1. Block Structure for Streaming

By placing the majority of our data on disk and streaming only data accessed by the current frame, the overall memory footprint of the volume renderer can be significantly reduced. This unlocks better scalability for memory-constrained environments. As a consequence, in order to achieve high performance interactive visualization using our system, a storage device with high random access bandwidth is needed. Luckily, modern solid state drives (SSD) can provide this capability. Therefore, our design can perform much better on SSDs compared to mechanical hard drives. Notably, the sequential I/O bandwidth of the storage device is not particularly relevant to our design. Therefore, similar performance is expected between NVMe and SATA SSDs, because their performance differences on random data access is not as drastic as those on sequential access. As emerging Peer-to-Peer Direct Memory Access (P2P) technology begins to enable GPUs to directly access SSDs via PCIe lanes [BBCS19], the use of NVMe SSDs might be justified. However, we leave P2P technology as an interesting area of future work for now.

Ideally, we would like the disk to support efficient access of arbitrary bricks on disk, since predicting data access patterns can be challenging. However, as each separate I/O operation will inevitably create some overhead, streaming only a single brick (which can be as small as several bytes) per request would be inefficient. We therefore make streaming requests at a granularity of the aforementioned *blocks* (Figure 2). Figure 3 shows the relationship between the block size and random access bandwidth for different storage devices. Moreover, in our streaming design, the entire volume domain is virtualized using a page table and an associated LRU cache (Section 3.2). As the chosen block size is decreased. more blocks are required to represent the dataset, and thus more memory would be needed to maintain the page table and cache.

In our design, the block size should also be aligned with the disk's physical sector size. This would allow data to be accessed directly through unbuffered file I/O (i.e., the `CreateFile` and `ReadFile` API with the `FILE_FLAG_NO_BUFFERING` flag on Windows), resulting in better streaming performance. Additionally, the block size used in the data file can also be different from the block size used for streaming (e.g., stream multiple blocks per I/O request). This is possible because blocks are uniformly aligned in both memory and the disk. This feature allows one to try multiple different block sizes for minimum zero-padding, while streaming with a different block size for better I/O performance. In our implementation, we use blocks of roughly 64KB because it yields a good balance between streaming performance and data scalability.

3.1.2. Treelet Optimization

The block structure enables more efficient streaming and guarantees that all the data associated with a brick can be obtained in at most one stream operation, even for arbitrarily ordered bricks. However, just as the performance of a hardware cache can vary drastically under different access patterns, the ordering in which bricks are stored can also affect rendering performance. In particular, a suitable brick ordering can significantly reduce the average number of streaming requests needed for traversal, resulting in a better rendering performance and lower cache miss rate.

One solution is to group bricks following the hierarchy order and create *treelets* (small sub-trees as shown in Figure 4). Different treelet designs suit different scenarios. For example, depth-first treelets may be good for visualizing voxelized surfaces because the ray tracing algorithm typically traverses the hierarchy in depth-first-order. However, breadth-first treelets would allow for quick filtering over multiple child nodes and thus can be advantageous for visualizing volumes with lots of empty spaces. In our design, we developed a treelet-based optimization to combine both orderings.

In this scheme, we create treelets by dividing the brick hierarchy once every N levels using a breath-first-traversal (Figure 4a, where N is fixed). In our TAMR implementation, we choose $N = 4$ as each TAMR brick occupies 72B, and so a full TAMR octree of more than five levels can potentially occupy more than 64KB space (our chosen block size), whereas a 4-level-full-octree can only occupy to $(8^4 - 1)/7 \times 72\text{B} \approx 42\text{KB}$. Similarly, in our RBF particle volume implementation, we choose $N = 9$ as an RBF particle BVH brick can occupy up to 112B, a 9-level-full-binary-tree can only take $(2^9 - 1) \times 112\text{B} \approx 57\text{KB}$. Obviously, we cannot always construct full treelets, so we use a depth-first traversal to collect multiple treelets for each block (Figure 4b). For different datasets, we tried slightly different block sizes to minimize zero-paddings.

3.2. Data Management System

Although modern SSDs are fast, the bandwidth provided is still not enough by itself to fully utilize the processing power of a modern CPU. Therefore, in addition to supporting streaming, we also implemented a page table and an LRU cache that can efficiently support concurrent and SIMD-vectorized accesses. Thanks to the brick abstraction, changing low-level details such as the scalar type will not affect the behavior of the data management system, allowing us to use the same data management system to support multiple volume structures.

3.2.1. Block Page Table

The block page table is a 64-bit integer array supporting random access, with each table entry associated with one block in the data file. The primary purpose of the block page table is to translate a block index in the file space to a block index in the RAM buffer space. This is necessary as we store only a small fraction of the file in

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

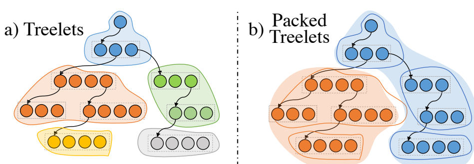

Figure 4: *a) A bigger tree can be decomposed into multiple smaller treelets, distinguished by different colors. Having multiple incoming links is not allowed for a treelet. Each treelet is created following a breath-first-traversal order. b) Because filling a block with exactly one treelet can result in inefficient use of space, multiple treelets are packed before forming blocks. Treelets are packed using a depth-first-traversal order.*

RAM at any one time. This operation is heavily used in our traversal implementation, as streaming often causes blocks to be shuffled. Such a shuffling breaks connections between a parent block and its children. Address translation automatically fixes these connections without the need to explicitly edit any brick values.

Within each block page table entry, we reserve two bits as flags as shown in Figure 5a, and the remaining bits are used to store the translated block index (i.e., the location of the block in the RAM LRU cache). The first flag (i.e., the status flag in Figure 5a) encodes whether the corresponding block in file has been cached. If the block is not loaded, we use the second flag (i.e., the request flag in Figure 5a) to encode whether a streaming request has been sent for this block. If the block is indeed loaded and cached, the request flag encodes whether the cache age data corresponding to this block has been updated. Essentially, the design purpose of the request flag is to guarantee that only one streaming request will be sent if the block is missing, and only one cache update will be made per frame if the block is presented in our cache. However, it is possible that multiple rays can be updating the same flag concurrently, which causes race conditions, repeatedly streaming the same block and repeatedly updating the same cache entry. Therefore atomic operations are used to update this flag.

We use this atomic method in all of our performance benchmarks and evaluation in this paper. However, having one atomic operation for each data access can reduce rendering performance due to synchronization overhead. For this reason, we also implemented an alternative method that does not utilize any atomic operations. On Windows, simple reads and writes to properly-aligned 32-bit variables are atomic operations [WSC*]. We utilize this feature and align our block page table and the LRU cache to 32-bit, and only use simple reads and writes to update the flag. Because threads all write the same value, values will not become corrupted. Although this method cannot eliminate all duplicate requests, our measurements indicate that it can eliminate most duplicated streaming requests and cache updates per frame. While the remaining small number of duplicated cache updates will not significantly affect rendering performance, repeatedly streaming the same data block can still be bad. So we also implemented a filtering algorithm in our asynchronous I/O queue implementation to further prevent this. In our preliminary experiments, we saw some small performance advantages brought by this method (up to 40% speedup in terms of minimum FPS for RBF particle volumes, and up to 5% for TAMR volumes). However, we stick with the atomic method for our primary results.

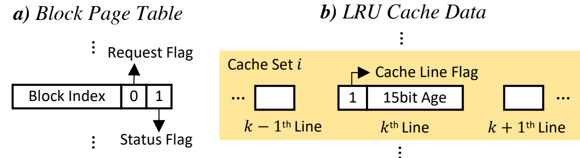

Figure 5: *Our page table and LRU cache design. In particular, we use a 64-bit integer to encode each cache line and each block page table entry for address translation.*

3.2.2. LRU Cache

We use the least recently used (LRU) algorithm to manage cached blocks. In particular, our LRU cache maintains two data structures: A cache table containing $N \times M$ entries and a pre-allocated data buffer for holding maximum $N \times M$ blocks. Each row within the table represents an associative set containing M cache lines, thus there are N sets in the cache. Each cache line is a 16-bit integer and is associated with a block in the pre-allocated data buffer (as shown in Figure 5b). The first bit of the cache line is used as a flag to indicate whether the current cache line is in use. If it is in use, the remaining bits are used to encode the corresponding block's age (i.e., the number of frames since the the block has been accessed). Notably, our cache table does not store tag bits, this is because the same information has already been stored by the block page table. Although this approach leads to a larger memory footprint, it can significantly simplify the data access process if a cache hit is found, thus improving the rendering performance.

In our design, the LRU algorithm is executed in between frames. First, when a streaming request is finished, the least recently used cache line will be searched. We copy over the loaded data into the cache line's corresponding buffer space. Finally, once all the finished requests are resolved, we increment all the cache line's ages by one. Although the entire operation is processed synchronously with respect to rendering, we did not observe a significant performance overhead caused by it in our CPU-based implementation. For a GPU-based implementation, a different LRU implementation should probably be used.

3.2.3. Data Streaming with Asynchronous I/O

All streaming requests are handled using a concurrent queue and a fixed-sized data streaming pool, operating in a dedicated I/O thread. The thread concurrently receives raw requests sent out by rays using the mechanism described in previous subsections. If the streaming pool is not full, these requests are consumed and asynchronously executed by the I/O thread using OS-dependent system calls. Data streamed from disk are temporarily staged in additional RAM buffers, one allocated for each I/O request. These data will not participate in rendering until they are committed to our LRU cache used by the renderer. For simplicity, we only commit these data changes between frames, and in our evaluation, we did not see this simplification becoming a performance bottleneck. However, we believe that an entirely concurrent commit mechanism should be possible. During human interaction, the rendering parameters are changed, and thus streaming requests made by previous frames should be canceled. In our implementation, we found that canceling a streaming request can create a small OS overhead. When too many requests are executed during interaction, the accumulated time to remove all invalid streaming requests can sig-

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

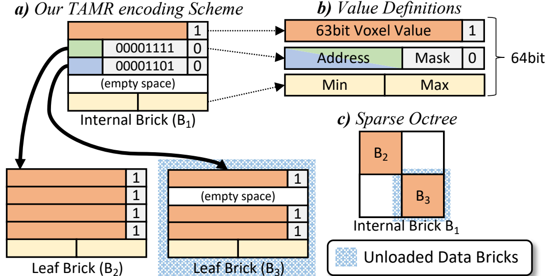

Figure 6: *a) Our brick design for TAMR and c) the corresponding octree structure. b) Within an internal brick, there can be internal nodes linking to other bricks. These links can be either a RAM space index (green) or a file space index (blue). For a leaf node, a 63 bit value is encoded (orange). Zero-value empty slots will be inserted for empty children. Each brick also contains the value range of itself (yellow).*

nificantly impact system interactivity. In our implementation, up to 20k streaming requests can be executed simultaneously.

3.3. Rendering

We couple our streaming system with an an efficient ray-tracing renderer supporting LoD control and progressive rendering to achieve interactive visualization of large volumetric datasets.

LoD control in our renderer is achieved as follows. A tree traversal with a certain LoD requirement will be terminated under three conditions: (a) a leaf brick is reached, (b) an internal brick with LoD level smaller than the the requested one is reached (in our context, a "smaller" LoD = higher resolution), or (c) an unavailable brick is reached. For (a), the most accurate sampling result can be returned. For (b), an approximated value for the sample can be returned. For (c), not only will the approximated value be calculated, the traversal algorithm will also return to the renderer the minimum LoD level found during the traversal. We refer to this process as the *traversal feedback loop* as shown in Figure 2. In our implementation, these feedback values are returned using additional ray payloads, and we leverage this traversal feedback mechanism to further achieve interactive progressive rendering. In particular, two progressive rendering methods are implemented.

The first method is called interval accumulation, inspired by several previous works [FK10, WBUK17], which conducts progressive volume rendering by dividing the entire volume integral into K smaller segment integrals. By allowing each ray to calculate only one segment per frame, interactivity can be improved. For each segment, samples will be evaluated and integrated as normal, and the results for each segment are stored in an array of size K. The result from the Kth segment should be stored in the Kth array entry. Such an array is allocated for every pixel at initialization, essentially creating a K-buffer. The final pixel color to display at each frame will be calculated by compositing all the currently cached colors from the K-buffer. However, since a sampling request posted by the renderer may not be fulfilled (due to data streaming), simply caching segment colors is insufficient. To compensate for this, for each segment, we also store the maximum of minimum LoD levels (as returned by the traversal feedback loop) of all samples taken within the segment. The K-buffer is reinitialized when the viewport or any

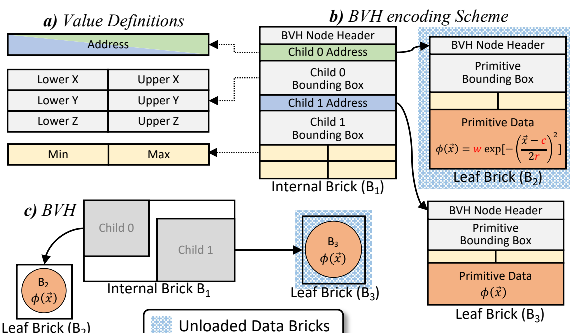

Figure 7: *a) Our brick design for RBF particle volumes. c) In particular, we pack the entire BVH into the streaming structure, and allow the BVH to be also progrssively streamed. b) We have two basic bricks: the internal brick which describes an internal BVH node containing two children, and the leaf brick which contains a BVH leaf node and the corresponding particle data.*

scene parameters are changed, and a lower quality rendering will be displayed (by increasing LoD) to improve interactivity during this period. This method allows our implementation to significantly reduce the memory footprint during rendering because data required to complete each frame is reduced.

The second method is tile accumulation, which achieves progressive rendering by dividing the frame into multiple smaller tiles, and tracing rays within one tile in one frame. In particular, tile accumulation can be combined with interval accumulation, so that the renderer will focus on the current tile and will continue to render the tile across multiple frames until the tile fully converges (i.e., all the samples are accurate), at which point the renderer moves to the next tile. This approach can further reduce the amount of data required by each frame, thus avoiding cache congestion.

Finally, we consider that a frame is "converged" when all the traversal requests are fulfilled (i.e., the minimum LoD value returned by the traversal feedback is no greater than the requested LoD for the traversal). An additional figure to illustrate our rendering strategy is provided in the supplementary material.

4. Applications to Different Volume Organizations

As previously discussed, we use an abstract hierarchical representation to represent volumes. We emphasize flexibility as a key feature of our design, and consequently only a small amount of traversal code must be changed in order to render different kinds of volume data once the data is properly prepared.

In Section 5, we will demonstrate our system operating with two different volume organizations: large-scale octree-accelerated volume rendering (represented by TAMR) and BVH-accelerated volume rendering (represented by RBF particle volume data). We now describe the brick formats and associated traversal algorithms for these two organizations. Additionally, we also illustrate here how a brick format and associated traversal algorithm could be implemented for regular grid volumes. However, for our evaluation we focus on TAMR and particle data, as existing volume streaming systems can already handle regular grid volumes efficiently.

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

```
struct NodeState {
  bool isLeaf; uint8 childrenMask; uint64 childrenOffset;
  union { vec2f range; T value; } data;
};
T SampleTAMR(vec3f position, Payload* payload) {
  TraversalStack stack[64];
  TraversalStack* stackPtr = /* push root */
  while (stackPtr > stack) {
    --stackPtr;
    NodeState state = accessNode(stackPtr, payload);
    if (not state.isLeaf) stackPtr = /* push children  */
    else return state.data.value; /* leaf & cache miss */
  }
  return INVALID_VALUE;
}
```

Listing 1: *Pseudo-code to traverse our TAMR data. We require only slight modifications (highlighted in red) to the standard octree traversal algorithm to enable streaming. The* `accessNode` *function can be found in the supplementary material.*

```
struct NodeState {
  Node* node; bool isLeaf; ParticleData data;
  uint64 child0; uint64 child1;
};
T SampleRBF(vec3f position, Payload* payload) {
  uint64 stack[32], index = 0, stackPtr = 0;
  while (1) {
    NodeState state = accessNode(index, payload);
    if (state.isLeaf) {
      return evaluateRBF(state.data, position);
    } else {
      InnerNode* inner = InnerNode* state.node;
      /* test with Axis-Aligned Bounding Box (AABB) */
      if (inAABB(inner->bounds[0], position)) {
        if (inAABB(inner->bounds[1], position)) {
          stack[stackPtr++] = state.child1;
          index = state.child0; continue;
        } else {
          index = state.child0; continue;
        }
      } else if (inAABB(inner->bounds[1], position)) {
        index = state.child1; continue;
      }
    }
    if (stackPtr == 0) return;
    index = stack[--stackPtr];
  }
}
```

Listing 2: *Pseudo-code to traverse our RBF particle volume data. We only need to slightly modify (highlighted in red) the BVH traversal algorithm. The* `accessNode` *function can be found in the supplementary material.*

4.1. Octree: TAMR Volumes

We start with a compact TAMR format based on previous work by Wang et al. [WMU*20], and consider each octant in the octree as a brick, so one brick contains exactly eight octree nodes in this case. If an octant contains empty nodes, zero-valued entries will be used. For each octree node, a 64-bit integer is used to encode its information (as illustrated in Figure 6). Particularly, the least significant bit of each node is used as a flag to indicate whether the node is a leaf node or not. For an inner node, an 8-bit value encoding the existence information of its eight potential children is stored, followed by a 55-bit index pointing to its first child (i.e., the child brick, as child nodes are stored contiguously). For a leaf node, instead of storing a mask and an index, the voxel's value is encoded using the remaining 63-bit. For a double-precision volume, one mantissa bit will be given up. In this paper we used single-precision volumes.

However, since the core idea of our method is to continuously stream and cache new data from high-performance storage, while also aggressively evicting unused data, we must make some modifications to the basic structure. First, at the end of each brick, we add two floating point values (yellow boxes in Figure 6) to store the value range of the brick (i.e., eight nodes). Then, we need to convert each child index into a block index and a block offset. Since we are using 64KB-worth-sized blocks, we reserve 16 bits for each block offset. Finally, we can use the standard octree traversal algorithm [WMU*20] to traverse the hierarchy with some slight modifications as shown in Listing 1. In our implementation, we use the value range's middle value to approximate the voxel value if a traversal is terminated at a inner node.

4.2. BVH Volumes: RBF Particle Volumes

A radial basis function (RBF) is a continuous scalar function $\phi(d)$ of distance d from a center of a particle. A RBF scalar field is defined by summing the kernels for all particles i contributing to a point x in space: $d_i(x) = ||x - x_i||,\ \Phi(x) = \sum_i \phi_i(d_i(x))$. In our implementation, ϕ is defined as a Gaussian: $\phi_i(x) = w_i \exp[-\frac{1}{2}\frac{d_i(x)^2}{r^2}]$, where w_i is the weight and r_i is the Gaussian radius. Thus, in our context, a particle is defined by a position, a radius and a weight.

In this subsection, we present another specialized brick format for RBF volume rendering. This particular implementation is developed based on the "particle" volume defined by the OpenVKL library. In particular, a binary BVH is constructed over all particles, with exactly one primitive (i.e., the particle) per leaf node. Each node contains a BVH node header which defines common properties such as the pointer to the parent node, and data to determine whether a node is a leaf or not. For an inner node, two child pointers and two bounding boxes are stored. For a leaf node, a bounding box and a primitive index is stored.

To enable streaming, first we need to pack each primitive data that describes a particle into the corresponding leaf node (as illustrated by Figure 7b). This is because we want to stream a part of the BVH together with all the corresponding particle attributes using one I/O operation. Then, we need to convert all pointers into byte distances relative to the root node. Next, we need to define an operation to create approximated values for traversals terminated at inner nodes. In our case, a "fake" particle is created by using the node value range's middle point as the particle weight, the bounding box's center as the particle coordinate, and the bounding box's size as the particle radius. Finally, we slightly modify the BVH traversal algorithm to enable streaming (as shown in Listing 2).

4.3. Regular Volumes

Although we focus on TAMR and RBF particle data in this work, we describe here how regular volumes could be implemented in our system. Regular volumes can be represented as a hierarchy of smaller grids. Each grid is a brick; therefore, the grid size is determined by the brick size (e.g., 16^3). For each voxel in a low-resolution brick, an address pointer and a value range are stored. The address pointer points to the child brick of this voxel, and the value range is the value range of the child brick. An encoded value is stored directly for each voxel in the highest resolution bricks (i.e., the leaf brick). Note that an internal brick occupies twice as much space as a leaf brick. Such a size difference does not affect the correctness of streaming and rendering as long as we always store two

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

Table 1: *We tested each dataset using 8 methods. Because our data streaming techniques provide better scalability, they were also evaluated with upsampled data (marked with ∗). In this table, we report the data **file** size and the memory footprint (**mem.** for short) during rendering. Results were measured in GB.*

	Disney Cloud		S3D-AMR		Meteor-20k		Meteor-46k		Exajet	
	File	Mem.	File	Mem.	File	Mem.	File	Mem.	File	Mem.
Immed. Static	17.7	20.1	14.7	16.9	1.85	2.99	3.31	4.56	7.70	9.29
Prog. Static	17.7	20.1	14.7	16.9	1.85	2.99	3.31	5.65	7.70	9.30
Immed. Block	17.7	6.01	14.7	-	1.85	5.41	3.31	5.49	7.70	6.32
Prog. Block	17.7	5.49	14.7	6.00	1.85	5.31	3.31	5.31	7.70	5.73
*Prog. Block	139	6.32	118	-	118	5.81	212	6.19	72.0	6.67
*Tile Block	139	5.48	118	5.79	118	5.39	212	5.58	72.0	5.91
Immed. Treelet	22.7	6.66	18.4	-	2.43	5.51	4.25	5.76	9.06	6.64
Prog. Treelet	22.7	5.62	18.4	6.27	2.43	5.31	4.25	5.31	9.06	5.99
*Prog. Treelet	178	6.69	148	-	147	5.97	267	6.19	61.5	6.69
*Tile Treelet	178	5.61	148	6.18	147	5.44	267	5.84	61.5	6.09

Table 2: *The frame converging times for different rendering methods. Results are measured in seconds. For both static methods, data are loaded into the system before rendering, therefore the results are reported as "$X+Y=Z$", where X is the data loading time and Y is the rendering time of the 1st frame.*

	Disney Cloud	S3D-AMR	Meteor-20k	Meteor-46k	Exajet
Immed. Static	31.7 + 11.1 = 42.8	29.4 + 50.0 = 79.4	2.7 + 2.9 = 5.6	4.9 + 4.0 = 8.9	11.1 + 5.0 = 16.1
Prog. Static	29.4 + 15.9 = 45.3	29.1 + 61.9 = 91.0	2.9 + 3.9 = 6.8	4.8 + 4.6 = 9.4	11.2 + 7.8 = 19.0
Immed. Block	207.9	-	66.4	45.0	100.6
Prog. Block	42.9	240.2	10.6	12.4	21.7
*Prog. Block	137.8	-	69.4	72.1	173.1
*Tile Block	182.3	728.8	112.7	120.1	94.3
Immed. Treelet	69.9	-	34.6	16.2	43.3
Prog. Treelet	22.0	95.7	6.4	7.5	12.2
*Prog. Treelet	66.3	-	36.9	40.7	111.8
*Tile Treelet	99.6	314.9	72.1	79.7	61.7

leaf bricks together (or pad some empty space). To traverse the hierarchy, the standard traversal algorithm for traversing a N^3 tree can be used [CNLE09].

5. Results and Evaluation

We implemented our streaming design in the Intel Open Volume Kernel Library (OpenVKL), and evaluated the system with a thorough analysis. We chose OpenVKL because it's a part of the OSPRay library, which has been used by production software tools such as ParaView [AGL05] and VisIt [CBW*12], such that our system can be leveraged directly by domain scientists for rendering large-scale TAMR and RBF particle volume data. In this section, our evaluation focuses on TAMR data, whose format is more compact allowing us to evaluate our design using large datasets. Limited by space, the RBF particle volume rendering results can be found in supplementary materials. We evaluate several critical aspects of our system from the perspectives of image accuracy, image converging speed, rendering performance, memory footprint, and I/O bandwidth utilization. Notably, we measured the memory footprint by recording the process's peak memory consumption before exit through the `PeakPagefileUsage` statistics available on Windows. Then, we utilized an external system monitoring tool [MM] to record the I/O bandwidth utilization.

To mainly test the scalability of our system, we performed our benchmark entirely on a mid-range desktop running Windows 11, with an Intel i5-10500 CPU and 16GB RAM clocked at 2133MHz. The operating system was installed in a Sabrent Rocket 4 plus 1TB SSD running at PCI-E Gen3 speed. All datasets used for benchmarking were loaded from an Intel 660p 1TB SSD. By default, Windows will setup its virtual memory in the Sabrent SSD and may utilize it to accelerate file I/O. To achieve controlled experiments, we always preloaded a large data before each experiment, and we also monitored the activity of all SSDs during all experiments to guarantee that significant disk activity was observed only on the Intel SSD. We ran all the experiments with a 4GB LRU cache, 32 progressive intervals, and tiles of size 128×128 if tile accumulation is enabled. We evaluated our system's rendering performance, interactivity, and scalability using four datasets:

- **Disney Cloud** [Stu] is a sparse volumetric cloud dataset. We converted it into a TAMR of 1.48B voxels and 12 levels.
- **S3D-AMR** is a multivariate combustion simulation produced by the S3D framework [TBB*17]. The data contains 1.26B voxels. We used the mass fraction field of $OC_{12}H_{23}OOH$.
- **Meteor** is the deep water impact dataset from LANL [PG17]. We used two timesteps, each containing 283M voxels.
- **Exajet** is a simulation of the airflow around a jet produced by NASA [CH14]. This model contains 656M voxels across 4 AMR levels, representing half of the plane.

We performed our benchmark by varying the strategies for TAMR volume traversal and progressive volume rendering. For traversal, we first created the baseline algorithm by re-implementing the work of Wang et al. [WMU*20] (which is referred to as the *static* method). Then we enabled data streaming and tested it with two configurations. The first configuration applied the treelet optimization method mentioned in Section 3.1.2, and aligned all blocks with 64KB in both file and RAM (*treelet*). The second configuration did not apply this optimization and uses blocks without additional padding (*block*). For rendering, we have three choices: disable progressive rendering (*immediate*), enable progressive rendering with just interval accumulation (*progressive*), or enable progressive rendering with both interval accumulation and tile accumulation (*tile*). To stress the scalability of our system, we also upsampled all the datasets. Disney Cloud, S3D-AMR and Exajet were upsampled by 8×, while Meteor-20k and Meteor-46k were upsampled by 64×.

5.1. Image Accuracy and Converging Speed

We compared images rendered by our streaming and progressive rendering methods with the baseline images generated on a much larger machine. We observe that our results are accurate: The *treelet* and *block* methods produce the same image. Images generated with progressive rendering can produce pixel differences. These differences are expected: When rendering with a normal ray marching algorithm, samples along the ray are taken regularly. When progressive rendering is used, each frame will only render a smaller segment, thus taking samples at different positions and creating a slightly different final result. In the supplementary material, we provide the image difference between an image rendered using the progressive treelet method and an image generated using the immediate static method (which implements a normal ray marching

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

Table 3: *Framerate comparisons between different methods. For immediate methods, only one framerate is reported per method. For other methods, because their framerates can vary, framerate results are reported as "X ~Y (Z)" where X is the maximum framerate, Y is the minimum, and Z is the framerate during a user interaction.*

	Disney Cloud	Meteor-20k	Meteor-46k	Exajet
Immed. Static	0.09	0.35	0.25	0.20
Prog. Static	68~0.67 (4.25)	54~3.49 (5.15)	48~2.59 (3.36)	67~0.92 (6.16)
Immed. Block	0.04	0.15	0.12	0.10
Prog. Block	60~0.29 (1.60)	53~1.45 (1.74)	56~1.45 (1.20)	60~0.41 (2.47)
*Prog. Block	60~0.09 (1.29)	52~0.23 (1.08)	51~0.32 (0.83)	65~0.33 (1.82)
*Tile Block	60~0.47 (-)	52~0.89 (-)	49~1.17 (-)	58~0.53 (-)
Immed. Treelet	0.07	0.24	0.19	0.16
Prog. Treelet	59~0.55 (3.40)	53~2.29 (3.58)	52~2.10 (2.48)	61~0.62 (4.75)
*Prog. Treelet	61~0.18 (2.52)	53~0.36 (2.05)	51~0.46 (1.63)	60~0.44 (3.53)
*Tile Treelet	59~0.77 (-)	55~1.51 (-)	48~1.80 (-)	60~0.80 (-)

algorithm). The images are different, but no obvious visual differences can actually be found.

Another important aspect of our evaluation is the frame converging speed. In Table 2, we show the time needed to complete the first fully converged frame in each experiment. Particularly, for *static* methods, we reported both data loading time and the first frame time, and we consider their sum as the frame converging time. For streaming methods, since the data loading process is overlapped with rendering, we reported the total elapsed time when the first converged frame was produced. In our implementation, this frame converging time is generally affected by the number of progressive intervals (i.e., the K-buffer depth) and the tile size. Increasing the intervals number (or decreasing the tile size) essentially decreases the rendering workload per frame, leading to a higher framerate, but we would need more frames to produce a converged image, which often leads to a longer converging time. Thus, one should do so only when the computation power is limited.

5.2. Compared with Baselines

To evaluate the rendering performance, we compared our method with the OpenVKL's *unstructured* volume rendering algorithm. In particular, we use the equilateral hexahedron as the base primitive to represent TAMR voxels in this experiment, with one primitive per BVH leaf. Because the geometry information of every primitive is explicitly stored and a BVH is built during the rendering, the memory footprint of the *unstructured* algorithm is enormous. In fact, in our experiment, for the full resolution Disney Cloud dataset, OpenVKL's *unstructured* algorithm required nearly 100GB memory space during rendering. Therefore, we conducted this comparison using a 8× downsampled version of the Disney Cloud dataset.

The work done by Wang et al. [WMU*20] (i.e., the *static* method) represents the state-of-the-art for CPU-based TAMR ray tracing. We compared it to our implementations with results shown in Figure 9. We found that the *progressive* method can significantly improve rendering FPS as the blue, green and red lines are well above the dotted line in the grey-shaded area, and the yellow line is also well above the dashed line in the same area. However, *treelet* and *block* streaming methods can increase the rendering complexity, thus the red and green lines are mostly below the blue line. Moreover, our streaming implementation also allows the TAMR rendering to generally outperform OpenVKL's *unstructured* volume rendering method as the orange and green lines are mostly above the dashed line.

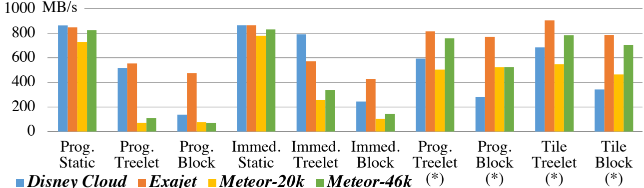

Figure 8: *The peak I/O performance of different configurations. For static methods, the sequential read bandwidth is reported. For other (streaming) methods, the peak I/O rate can be comparable with or even faster than sequentially reading the data.*

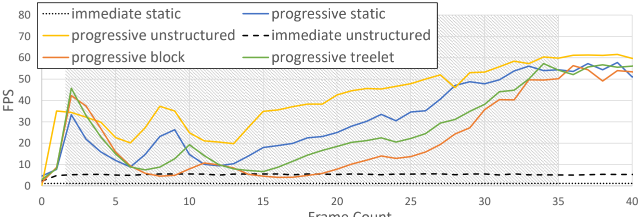

Figure 9: *Rendering performance comparison with two baselines. The first baseline uses OpenVKL's unstructured volume rendering method to render TAMR voxels (dashed line). The second method uses the TAMR method proposed by Wang et al. [WMU*20] (dotted line). The 8× downsampled Disney Cloud data was used in this test, because OpenVKL unstructured algorithm requires a lot of system RAM to process the full resolution version. Our progressive rendering takes the first few frames to warm up its K-buffer.*

5.3. Streaming Algorithm and Treelet Optimization

Our streaming method can significantly reduce the memory footprint of volume rendering. As indicated by Table 1, the memory footprint of the streaming experiments (i.e., the *treelet* and the *block* methods) did not increase linearly as the total data size increased. For experiments with large data, they produce a memory footprint as low as 2.2% of the data size (as highlighted by the blue color). Note that all the streaming experiments are performed with a 4GB cache (i.e., Figure 2c). Because the memory footprint is measured as the peak value in bytes of the *Commit Charge* during the lifetime of a process, it is reasonable to see experiments with an even smaller memory footprint if the cache is not fully utilized.

Our streaming implementation can also utilize the I/O bandwidth efficiently. Figure 8 showcases the peak data read rate measured using the system monitoring tool. For baseline experiments (i.e., *static* methods), data are loaded sequentially using the standard blocking I/O API call with file buffering and memory paging enabled. Their sequential read bandwidths are reported in the comparison. Sequential data access is usually significantly faster than random accesses that occur naturally during rendering. However, for some streaming experiments, the peak I/O rate measured during rendering can be comparable with or even faster than sequentially reading the data. Our treelet optimization further enhances this result as the I/O bandwidth utilization for *treelet* experiments are virtually higher than all the corresponding *block* ex-

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

periments. Notably, the simple I/O method used in the baseline experiments cannot utilize all the bandwidth provided by the SSD. To achieve the maximum read performance, a well-engineered low-latency queuing mechanism that can simultaneously execute many asynchronous read tasks is required. However, as many visualization systems do not provide such a sophisticated mechanism for sequential I/O, the simpler (and more representative) method is being compared in our experiments. Moreover, using such a mechanism might improve the data loading times for the baseline methods, but would not improve rendering speed itself.

Streaming can also improve rendering performance. As indicated by the green entries in Table 2, the *progressive treelet* method can produce a converged image faster than the *static* methods. Although the *static* method offers a higher rendering rate, it requires a significant amount of time to load the data before rendering can even start. The treelet optimization contributes significantly to this result because it outperforms almost all the *block* experiments in terms of framerates in Table 3, and the *progressive block* method generally underperforms compared against the *static* method.

5.4. Progressive Rendering Improves Scalability

Data streaming (with caching) alone cannot entirely solve the big data challenge. This is because the algorithm cannot guarantee a converged image when the cache is completely filled with data required by the current frame. In this situation, newly loaded data cannot be committed into the cache without evicting data required by other parts of the same frame. This dilemma essentially prevents the image from converging. Such a situation was frequently observed when experimenting with the *immediate treelet* and the *immediate block* methods (for the S3D-AMR data as well as **all** the upsampled data, which are not shown in Table 1 and Table 2).

Progressive rendering solves this issue because it can limit each individual frame's total amount of data. The result of this is that the *progressive treelet* and *progressive block* can render most of the upsampled datasets successfully. This is because the rendering workload for each frame is also reduced by the interval accumulation method. Moreover, although more frames would be needed to get a fully converged image, the rendering system can be more interactive and allow users to tweak rendering parameters before a fully converged frame is produced. For the upsampled S3D-AMR data, we found that using interval accumulation alone could not produce a converged image due to cache congestion. However, they can be rendered after enabling tile accumulation.

Generally speaking, tile accumulation will further increase the frame converging time. However, an outlier was observed for the upsampled Exajet data as indicated by the blue entries in Table 2, where experiments with tile accumulation converged faster than experiments without tile accumulation. Figure 10 provides more details for this case. For the data without upsampling, the dip in Figure 10A is caused by having rays waiting for streaming requests, and more rendering workloads are required after data is loaded. In this case, because all the required data can fit into the data cache simultaneously, exactly one FPS dip can be found for each experiment. When it comes to upsampled data, the data required by each interval no longer fits into the cache. Some streaming requests are

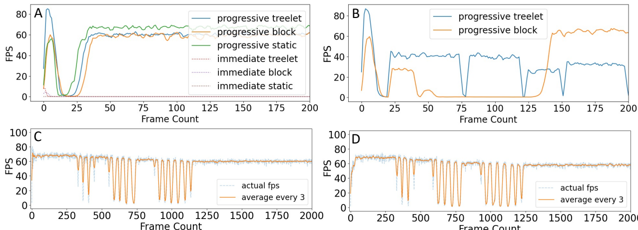

Figure 10: *Real-time framerates of our system rendering the Exajet data. a) Framerates for the original resolution data. b) Using the progressive treelet and the progressive block methods to render the 8× upsampled data. c) The 8× upsampled data with the tile treelet method. d) The 8× upsampled data with the tile block method.*

suspended temporarily until some of the data are evicted from the cache. Therefore, multiple dips can be found. However, because the suspension of streaming requests is not carefully scheduled, the same data might be repeatedly loaded and evicted by different rays. This causes the rendering performance to deteriorate. For tile accumulation, this problem is solved because by further reducing the per-frame data requirement. This also allows data to be accessed with better locality, thus improving the cache performance and the rendering performance.

6. Conclusion

We present a flexible data streaming design for interactive visualization of large-scale volume data. In particular, our design is generalizable and can simultaneously utilize asynchronous data streaming, a concurrent LRU cache, and high-performance progressive rendering. We demonstrate our data streaming design with both tree-based AMR volume rendering and RBF particle volume rendering examples. Our test results show that the system design and implementation for TAMR scale well and can render large TAMR volumes up to 10× the host system's RAM capacity.

However, we also find limitations in our work. First, our system design is currently created for CPU-based ray tracing. Recent advances in GPU-based ray-tracing hardware have demonstrated great potential. Moving our design to GPU can be an exciting topic for new research. Second, our system currently does not support multivariate volumes. All scientific simulations produce multivariate data. Extending our design and implementation to support the rendering of multivariate data will benefit a broader user group.

Acknowledgments

This work was sponsored in part by the U.S. Department of Energy through grant DE-SC0019486 and an Intel oneAPI Center of Excellence grant.

References

[AGL05] AHRENS J., GEVECI B., LAW C.: ParaView: An End-User Tool for Large-Data Visualization. In *The Visualization Handbook*, Hansen C. D., Johnson C. R., (Eds.). Elsevier, 2005, pp. 717–731. 8

[BBCS19] BERGMAN S., BROKHMAN T., COHEN T., SILBERSTEIN M.: Spin: Seamless operating system integration of peer-to-peer dma between ssds and gpus. *ACM Trans. Comput. Syst. 36*, 2 (2019), 1–26. 4

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

[BC89] BERGER M. J., COLELLA P.: Local adaptive mesh refinement for shock hydrodynamics. *J. Comput. Phys. 82*, 1 (1989), 64–84. 3

[BHP15] BEYER J., HADWIGER M., PFISTER H.: State-of-the-art in gpu-based large-scale volume visualization. In *Computer Graphics Forum* (2015), vol. 34, Wiley Online Library, pp. 13–37. 2

[BO84] BERGER M. J., OLIGER J.: Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comput. Phys. 53*, 3 (1984), 484–512. 3

[CBW*12] CHILDS H., BRUGGER E., WHITLOCK B., MEREDITH J., AHERN S., PUGMIRE D., BIAGAS K., MILLER M., HARRISON C., WEBER G. H., KRISHNAN H., FOGAL T., SANDERSON A., GARTH C., BETHEL E. W., CAMP D., RÜBEL O., DURANT M., FAVRE J., NÁVRATIL P.: VisIt: An End-User Tool for Visualizing and Analyzing Very Large Data. *High Performance Visualization* (2012). 8

[CH14] CASALINO D., HAZIR A.: Lattice boltzmann based aeroacoustic simulation of turbofan noise installation effects. In *23rd International Congress on Sound and Vibration* (2014), pp. 1–8. 8, 14

[CNLE09] CRASSIN C., NEYRET F., LEFEBVRE S., EISEMANN E.: Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the symposium on Interactive 3D graphics and games* (2009), pp. 15–22. 2, 8

[DL19] DUBOIS J., LEKIEN J.-B.: Highly efficient controlled hierarchical data reduction techniques for interactive visualization of massive simulation data. In *EuroVis (Short Papers)* (2019), pp. 37–41. 3

[Eng11] ENGEL K.: Cera-tvr: A framework for interactive high-quality teravoxel volume visualization on standard pcs. In *IEEE Symposium on Large Data Analysis and Visualization* (2011), pp. 123–124. 2

[FAW10] FRAEDRICH R., AUER S., WESTERMANN R.: Efficient high-quality volume rendering of sph data. *IEEE transactions on visualization and computer graphics 16*, 6 (2010), 1533–1540. 3

[FK10] FOGAL T., KRÜGER J. H.: Tuvok, an architecture for large scale volume rendering. In *VMV* (2010), vol. 10, pp. 139–146. 2, 6

[FSK13] FOGAL T., SCHIEWE A., KRÜGER J.: An analysis of scalable gpu-based ray-guided volume rendering. In *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)* (2013), IEEE, pp. 43–51. 2

[GMG08] GOBBETTI E., MARTON F., GUITIÁN J. A. I.: A single-pass gpu ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer 24*, 7 (2008), 797–806. 2

[HBJP12] HADWIGER M., BEYER J., JEONG W.-K., PFISTER H.: Interactive volume exploration of petascale microscopy data streams using a visualization-driven virtual memory approach. *IEEE Transactions on Visualization and Computer Graphics 18*, 12 (2012), 2285–2294. 2

[JFSP10] JANG Y., FUCHS R., SCHINDLER B., PEIKERT R.: Volumetric Evaluation of Meshless Data From Smoothed Particle Hydrodynamics Simulations. In *IEEE/EG Symposium on Volume Graphics* (2010), Westermann R., Kindlmann G., (Eds.). 3

[KJM21] KNOLL A., JOHNSON G. P., MENG J.: Path tracing rbf particle volumes. In *Ray Tracing Gems II*. 2021, pp. 713–723. 3

[KWN*14] KNOLL A., WALD I., NAVRATIL P., BOWEN A., REDA K., PAPKA M. E., GAITHER K.: Rbf volume ray casting on multicore and manycore cpus. In *Computer Graphics Forum* (2014), vol. 33, Wiley Online Library, pp. 71–80. 3

[LCPT12] LABADENS M., CHAPON D., POMARÉDE D., TEYSSIER R.: Visualization of octree adaptive mesh refinement (amr) in astrophysical simulations. *Astronomical Data Analysis Software and Systems XXI 461* (2012), 837. 3

[LHJ00] LAMAR E., HAMANN B., JOY K.: Multiresolution techniques for interactive texture-based volume visualization. *Proceedings of the International Society for Optical Engineering* (11 2000). 2

[LLY06] LJUNG P., LUNDSTRÖM C., YNNERMAN A.: Multiresolution Interblock Interpolation in Direct Volume Rendering. In *EuroVis* (2006), Santos B. S., Ertl T., Joy K., (Eds.). 3

[LSMT99] LAW C., SCHROEDER W., MARTIN K., TEMKIN J.: A multi-threaded streaming pipeline architecture for large structured data sets. In *Proceedings Visualization* (1999), pp. 225–232. 2

[LVI*13] LEAF N., VISHWANATH V., INSLEY J., HERELD M., PAPKA M. E., MA K.-L.: Efficient parallel volume rendering of large-scale adaptive mesh refinement data. In *IEEE Symposium on Large-Scale Data Analysis and Visualization* (2013), pp. 35–42. 3

[MM] MARTIN MALIK R.: Hwinfo - free system information, monitoring and diagnostics. URL: https://www.hwinfo.com. 8

[Mon92] MONAGHAN J. J.: Smoothed particle hydrodynamics. *Annual review of astronomy and astrophysics 30*, 1 (1992), 543–574. 3

[Mus13] MUSETH K.: Vdb: High-resolution sparse volumes with dynamic topology. *tog 32*, 3 (2013), 1–22. 2

[Mus21] MUSETH K.: Nanovdb: A gpu-friendly and portable vdb data structure for real-time rendering and simulation. In *ACM SIGGRAPH 2021 Talks* (2021), SIGGRAPH '21. 2

[PG17] PATCHETT J., GISLER G.: *Deep water impact ensemble data set*. Tech. rep., Los Alamos National Laboratory, 2017. 1, 8, 14

[RKN*13] REDA K., KNOLL A., NOMURA K.-I., PAPKA M. E., JOHNSON A. E., LEIGH J.: Visualizing large-scale atomistic simulations in ultra-resolution immersive environments. In *LDAV* (2013), pp. 59–65. 3

[SCRL19] SARTON J., COURILLEAU N., RÉMION Y., LUCAS L.: Interactive visualization and on-demand processing of large volume data: a fully gpu-based out-of-core approach. *IEEE transactions on visualization and computer graphics 26*, 10 (2019), 3008–3021. 2

[SM05] SCHROEDER W. J., MARTIN K. M.: The visualization toolkit. In *The Visualization Handbook*, Hansen C. D., Johnson C. R., (Eds.). Elsevier, 2005, pp. 593–614. 2

[Stu] STUDIOS W. D. A.: Clouds data set - walt disney animation studios. URL: https://www.disneyanimation.com/data-sets. 8, 14

[TBB*17] TREICHLER S., BAUER M., BHAGATWALA A., BORGHESI G., SANKARAN R., KOLLA H., MCCORMICK P. S., SLAUGHTER E., LEE W., AIKEN A., ET AL.: S3d-legion: An exascale software for direct numerical simulation of turbulent combustion with complex multicomponent chemistry. In *Exascale Scientific Applications*. 2017, pp. 257–278. 8, 14

[WBUK17] WALD I., BROWNLEE C., USHER W., KNOLL A.: Cpu volume rendering of adaptive mesh refinement data. In *SIGGRAPH Asia 2017 Symposium on Visualization* (2017), SA '17, pp. 9:1–9:8. 6

[WJA*16] WALD I., JOHNSON G. P., AMSTUTZ J., BROWNLEE C., KNOLL A., JEFFERS J., GÜNTHER J., NAVRÁTIL P.: Ospray-a cpu ray tracing framework for scientific visualization. *IEEE transactions on visualization and computer graphics 23*, 1 (2016), 931–940. 3

[WMU*20] WANG F., MARSHAK N., USHER W., BURSTEDDE C., KNOLL A., HEISTER T., JOHNSON C. R.: Cpu ray tracing of tree-based adaptive mesh refinement data. In *Computer Graphics Forum* (2020), vol. 39, pp. 1–12. 3, 7, 8, 9

[WSC*] WHITE S., SHARKEY K., COULTER D., BATCHELOR D., AIGNER R., SATRAN M.: Interlocked variable access - win32 apps. URL: https://docs.microsoft.com/en-us/windows/win32/sync/interlocked-variable-access. 5

[WWJ19] WANG F., WALD I., JOHNSON C. R.: Interactive rendering of large-scale volumes on multi-core cpus. In *IEEE 9th Symposium on Large Data Analysis and Visualization* (2019), pp. 27–36. 2

[ZWE*00] ZIMMERMANN K., WESTERMANN R., ERTL T., HANSEN C., WEILER M.: Level-of-detail volume rendering via 3d textures. In *IEEE Symposium on Volume Visualization* (2000), pp. 7–13. 2

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.