Sort-First Parallel Volume Rendering

Brendan Moloney, Marco Ament,
Daniel Weiskopf, *Member*, *IEEE Computer Society*, and Torsten Möller, *Senior Member*, *IEEE*

Abstract—Sort-first distributions have been studied and used far less than sort-last distributions for parallel volume rendering, especially when the data are too large to be replicated fully. We demonstrate that sort-first distributions are not only a viable method of performing data-scalable parallel volume rendering, but more importantly they allow for a range of rendering algorithms and techniques that are not efficient with sort-last distributions. Several of these algorithms are discussed and two of them are implemented in a parallel environment: a new improved variant of early ray termination to speed up rendering when volumetric occlusion occurs and a volumetric shadowing technique that produces more realistic and informative images based on half angle slicing. Improved methods of distributing the computation of the load balancing and loading portions of a subdivided data set are also presented. Our detailed test results for a typical GPU cluster with distributed memory show that our sort-first rendering algorithm outperforms sort-last rendering in many scenarios.

Index Terms—Volume rendering, sort-first parallelization, visualization, dynamic load balancing, early ray termination, shadow, ray coherence.

1 Introduction

Many scientific simulations and measurements result in enormous volumetric data sets. Volume rendering is an essential tool for visualizing and gaining insight from such data. The process of exploring volumetric data can also benefit greatly from volume rendering, but only if the user can interactively alter the viewing conditions. To perform interactive volume rendering, even of small data sets, requires a tremendous amount of computational power. Recently, Graphics Processing Units (GPUs) have provided a cost-efficient method of rendering small to medium sized data sets at interactive frame rates. However, larger data sets still require us to distribute the workload and data set among a number of processing units.

GPUs have their own dedicated high-speed memory to maximize the bandwidth available to the processing core. In a parallel environment, this extra layer of memory further complicates the problem of simultaneously distributing the data set and the rendering workload evenly. This is largely the reason for the focus on static sort-last distributions for GPU accelerated parallel volume rendering. While sort-last methods do ultimately demonstrate better data scalability than sort-first methods, we show that sort-first can give better performance in many scenarios where data scalability is required. The performance difference comes from the increased ability to leverage occlusion and the lack of alpha compositing overhead.

The partitioning strategy used to distribute the rendering workload and data set among the processing units can limit the types of algorithms that are applicable within the parallel rendering environment. In particular, many image-space algorithms cannot be efficiently adapted to work with a sort-last distribution since it does not keep the data and processing along each (virtual) ray local to a single processing unit. These algorithms (which we call ray coherent) can provide tremendous speedups through visibility culling, more informative images through sophisticated lighting models that include shadowing effects, more accurate and consistent load balancing, and potentially many other benefits.

The target platform is a cluster of machines with GPUs and distributed memory. The algorithms are agnostic in regards to the type of interconnect used for communication, but for our tests, gigabit ethernet is used due to its availability and affordability. To make our results relevant to more cluster configurations, we also estimate the performance for higher bandwidth network interconnects. Since the processing units in our cluster have the same amount of RAM as the entire cluster's GPU memory, there is currently no need to send volume data over the network. The basic rendering approach used is three-dimensional texture slicing but alternative techniques such as ray casting could be used as well (except when half angle slicing is used for shadowing).

1.1 Goals and Contributions

The majority of the state-of-the-art research on parallel GPU volume rendering has focused on sort-last distributions. Our goal is to show that sort-first methods can outperform sort-last methods in many scenarios that require data scaling and allow for efficient parallelization of a number of existing volume rendering algorithms that would otherwise be intractable. Experimental results for each stage of the

- *B. Moloney is with the AIRC—Advanced Imaging Research Center, Oregon Heath Science University, 3181 SW Sam Jackson Park Rd, L452, Portland, OR 97239. E-mail: moloney@ohsu.edu.*
- *M. Ament and D. Weiskopf are with the VISUS—Visualization Research Center, Universität Stuttgart, Allmandring 19, D-70569 Stuttgart, Germany. E-mail: marco.ament@vis.uni-stuttgart.de, weiskopf@visus.uni-stuttgart.de.*
- *T. Möller is with the School of Computing Science, Simon Fraser University, 8888 University Drive, Burnaby BC, Canada V5A 1S6. E-mail: torsten@cs.sfu.ca.*

Manuscript received 29 Apr. 2009; revised 26 Jan. 2010; accepted 3 May 2010; published online 8 Sept. 2010.
Recommended for acceptance by K.-L. Ma.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCG-2009-04-0086.
Digital Object Identifier no. 10.1109/TVCG.2010.116.

1077-2626/11/$26.00 © 2011 IEEE Published by the IEEE Computer Society

parallel rendering pipeline are provided and we compare sort-first and sort-last distributions as fairly as possible.

In Section 4, we discuss our contributions toward two of the fundamental components of data-scalable sort-first volume rendering: load balancing and data redistribution. An existing load balancing algorithm which provides consistently good results is used and we reduce its overhead by lowering the amount of communication required. To reduce the sudden dips in performance associated with data redistribution, a proximity-based caching scheme that predictively loads data in close proximity to the frustum is described.

In Section 5, a number of ray coherent algorithms and their benefits are discussed. Two of the algorithms are implemented and discussed in detail. The first algorithm leads to efficient and effective early ray termination for parallel GPU volume rendering, which is not possible with existing sort-last methods. Our improved algorithm incurs almost no overheads when there is little volumetric occlusion while achieving superior performance when volumetric occlusion does occur. The second algorithm uses a hybrid sort-first and sort-last distribution which allows us to adapt an image-based volumetric shadowing algorithm to a parallel environment. Ray coherence along the shadow rays is achieved by performing a sort-first distribution of the light's image-space. This essentially gives us a sort-last distribution from the camera's point of view, which requires us to composite the intermediate images in order to get the final result.

In Section 6, results for each algorithm and stage of the parallel rendering pipeline are given. We show that our data-scalable sort-first distribution can often give better performance than a sort-last distribution, and that our proximity caching scheme performs better than methods with least recently used (LRU) caching. Similarly, we justify our choice of load balancing algorithm by showing that it provides better and more consistent results than the more commonly used alternative. Occlusion culling is shown to scale dramatically better with a sort-first distribution compared to sort-last, allowing large performance gains. We also provide a look at the overall performance for large real-world data sets.

This paper builds upon previous work on parallel volume rendering by Moloney et al. [1]. Where we directly utilize techniques from that previous paper, we just include a reference to that paper in order to avoid duplication of paper contents. This paper adopts text from [1] only in small parts in Sections 2, 3, and 4.

2 Related Work

In this section, we look at related work in GPU accelerated volume rendering as well as parallel volume rendering.

2.1 GPU Accelerated Volume Rendering

Three-dimensional texture slicing [2], [3] is an easy way to interactively perform high-quality volume rendering of small to medium sized data sets. This is the basic rendering method used for the generation of intermediate images on the nodes of our cluster-based rendering system. For the special case of volume shadowing, a variant of 3D texture slicing called half angle slicing [4], [5] is adopted. This slicing technique produces a single set of slices that can be used to render the data from two different points of view. The direction perpendicular to the slices is chosen so that it is the half vector of the two view directions if they both lie in the same hemisphere, or the half vector of one view direction and the inverse of the other if they lie in opposite hemispheres. By using half angle slicing and alternating between rendering each slice from the point of view of the camera and the point of view of the light source, a shadowing effect can be produced [4], [5].

Early volume ray casting algorithms for the GPU used a multipass approach due to the limited number of instructions that could be executed in a shader program on older GPUs [6], [7]. These multipass approaches were able to update the depth buffer between each pass, and set it to kill fragments above some opacity threshold using depth culling. This approach to early ray termination is applicable to any iterative front to back algorithm, and has been adapted for data sets subdivided into bricks [8].

Newer GPUs facilitate single-pass ray casting [9], which allows for effects like reflection, refraction, and self-shadowing isosurfaces. Although our implementation is restricted to 3D texture slicing, GPU ray casting could replace slicing in the sort-first approach of this paper because the parallelization strategy and domain decomposition is independent from the core volume rendering technique. The only exception is volume shadowing which is tightly linked to half angle slicing.

Subdividing the data into bricks is a popular method of empty space leaping on GPUs due to the fact that it can reduce not only the amount of computations but also the amount of texture memory required. Bricking has been used both for slice-based rendering [10] and ray casting [11]. More accurate methods of reducing the computations on empty voxels [8], [12] exist, but do not save any additional texture memory. Bricking has also been used in parallel rendering for data distribution [13] and load balancing [11], [14].

2.2 Parallel Rendering

A variety of methods have been proposed for distributing the rendering workload among a number of machines. Molnar et al. [15] classify these into groups based on where in the rendering pipeline primitives are sorted in regards to viewing conditions. The sort-last method probably is the most common for parallel volume rendering. One of the primary research topics for sort-last algorithms is how to efficiently transfer and composite the intermediate images created by the processing units in the cluster. A parallel pipeline approach [16] from parallel polygon rendering organizes n processors in a circular ring and divides the z-buffer of each node into n disjoint regions. The subimages are circulated along the ring and accumulated in a pipelined fashion. Binary swap [17] and direct send [18], [19] compositing schemes are easy to implement and do a fair job of distributing the compositing workload among the render nodes. SLIC [20] improves direct send compositing primarily through better load balancing and scheduling. Overlapping local ray casting and compositing [21] reduces network congestion because smaller messages are sent over

the network throughout the entire process of rendering instead of sending large messages when all ray casting processes are completely finished.

The focus of our paper is sort-first methods for parallel volume rendering. In general, these methods either replicate the data set across all render nodes [22] or transfer data over the network [13]. Algorithms that replicate the full data set on each GPU can only scale performance, but not the maximum data set size. Algorithms that transfer data over the network, or cache data locally, can allow for data sets larger than the memory available to a single processing unit. The work of Bethel et al. [13] takes a detailed look at the amount of data communication required for data-scalable sort-first volume rendering but it does not consider the effects of different caching techniques.

Neumann [23] compares the communication costs for sort-first versus sort-last volume rendering. He concludes that dynamic sort-first distributions can have much worse communication costs than sort-last. However, this does not consider the possibilities of caching, asynchronous loading, or avoiding loading of occluded data. It also does not consider the need for load balancing for sort-last distributions, which would increase the communication requirements substantially.

Eilemann et al. [24] present a generic framework for parallel rendering which handles a variety of different data types and applications. Sort-first and sort-last approaches are also compared but the results are difficult to compare to our own. The sort-last algorithm shows superlinear scaling since the data are out of core for small numbers of nodes. The sort-first algorithm on the other hand is not data-scalable when rendering volume data and thus shows consistently sublinear scaling.

The shadow rendering technique mentioned in the previous section can be adapted to work in a parallel environment using a sort-last distribution [25]. The sort-last distribution allows for good data scalability but it requires two rendering and composting passes. The first pass generates a global shadow map and the second pass generates the rendered image. Such an approach would be prohibitively slow on gigabit ethernet even with moderate image resolutions. With a higher bandwidth network interconnect the additional processing and synchronization would be the main concern. In contrast, our approach described in Section 5.2 only requires a single rendering and compositing pass.

Load balancing is an important research subject for parallel volume rendering, as the overall performance is limited by the slowest component. Static load balancing algorithms do not perform well under a variety of viewing conditions unless overpartitioning is used. Overpartitioning the image-space causes much more data replication when using a data-scalable sort-first approach and overpartitioning the object-space causes much higher compositing overheads. An important overpartitioning strategy in sort-last load balancing is the so called k-way replication [26] that originates from parallel polygon rendering. Every rendering primitive is replicated k times on n nodes ($k \ll n$). In this way, load imbalances coming from zooming into the data set are alleviated because, in contrast to a static one-way distribution, starving nodes can dynamically change the subset of primitives in a view-dependent way without exchanging excessive amounts of geometry data during rendering. However, the amount data replication used in the paper is slightly higher than what is needed by our sort-first algorithm for the same number of nodes (about one quarter of the data set stored on each of 24 nodes). Additionally, the amount of data replication for our sort-first algorithm decreases with additional nodes while the amount of data replication for k-way replication remains constant for a given quality of load balancing.

Dynamic load balancing tends to assign each unit a single partition and thus avoids these problems. Each processing unit's partition is updated as the camera moves in order to maintain good load balancing. A common technique uses the relative performance of each rendering node in the previous frame. This has been done with sort-last algorithms [11], [14] that subdivide the volume into bricks and reassign bricks to nodes that had a higher frame rate (less workload) in previous frames. Despite being sort-last, these methods require volume data to be sent over the network or cached locally.

Sort-first algorithms have also used this method of load balancing [22], redistributing the image-space instead of the object-space. Any such method relies on frame-to-frame coherence and thus cannot guarantee any tight bounds on the level of load balancing. Sort-first load balancing algorithms that do not rely on frame-to-frame coherence tend to estimate the rendering cost for different portions of the screen and then divide up these portions evenly. The mesh-based adaptive hierarchy decomposition (MAHD) [27] does this for surface-based rendering by dividing the screen into tiles and then computing a weighted sum of the primitives that project to that tile. However, these rendering primitives are not applicable in volume rendering.

With the worker farm paradigm [28] a master process issues work items, in this case a block of pixels, to distributed worker processes. After a time-out period has expired, all workers inform the master how much time they need to process the current item. Workers that are idle or expect completion soon are assigned new items in order to keep a good load balance. A similar sort-first approach originates from parallel polygon scan conversion [29] and shared-memory parallel rendering [30]. A processor that runs out of tasks searches for the processor with the highest load among the others and splits the work in half or steals entire tasks from the queue to redistribute the load evenly. Although these methods are generic and are not restricted to polygon rendering they need to communicate during frame processing because of task redistribution, which disturbs efficient rendering, especially in modern graphics environments that are highly sensitive to stalls in the rendering pipeline.

The cost of computing a pixel's color is directly related to the number of participating fragments along the view ray. Calculating the number of intersections of the ray with the cell faces of the grid [31] is close to our approach, but we also account for the length of a ray segment within each brick, which allows a more accurate estimate of the costs. The load balancing scheme for sort-first volume rendering we published previously [1] computes the cost of rendering each pixel (or small group of pixels) on the

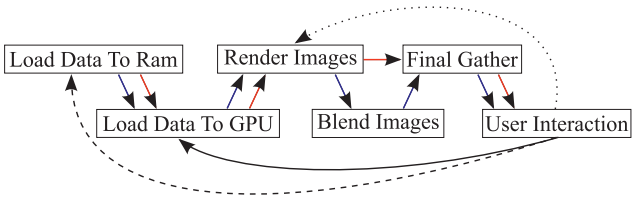


Fig. 1. A generic overview of the parallel DVR pipeline.

GPU by taking the associated view ray and adding up the lengths of all of its intersections with the portions of the data set. In Section 4.3, we improve our method with a staged communication pattern to reduce the overhead of load balancing computation.

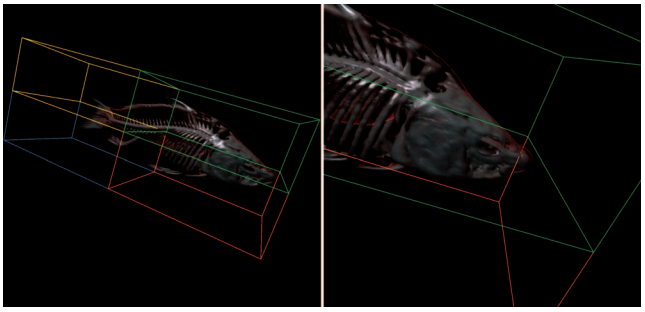
3 Partioning Strategies for Parallel Rendering

There are two main reasons for using multiple processing units to render volumetric data. The first is that the amount of processing required might take too long to achieve interactive frame rates, and the second is that the data itself might be too large to fit into the local memory of a single unit. How well a parallel workload distribution addresses the former issue can be called its "performance scaling" and how well it addresses the latter issue can be called its "data scaling." Often, it is difficult to balance both of these goals reliably for all viewing conditions.

In Fig. 1, a generic parallel rendering pipeline is illustrated, as well as the paths that several algorithms take. The red and blue paths through the pipeline correspond to the sort-first and sort-last algorithms, respectively. There are then three possible points to loop back to in each frame. The dotted line shows the path taken by the sort-last algorithm when static load balancing is used and by the sort-first algorithm without data scaling. When the data fit into the RAM of a single processing unit, both the sort-last algorithm with dynamic load balancing and the sort-first algorithm with data scaling take the path with the solid line. When the data does not fit into the RAM of a single processing unit, both algorithms must take the path shown with the dashed line.

Recently, there has been a focus on sort-last distributions due to their good data scaling. With a simple static distribution such an approach provides nearly ideal data scaling and reasonable performance scaling when the data set is viewed globally. However, as illustrated in Fig. 2, once the user starts to zoom in to look at smaller features of the data set (an increasingly common behavior with larger data sets), such a data distribution is no longer sufficient. Overpartitioning the data can reduce this problem but at the cost of increased compositing and reduced data scalability. Dynamic load balancing on the other hand requires data to be redistributed as the viewpoint changes.

A sort-first distribution does not need to alpha composite intermediate images, and thus can provide better performance scaling in certain scenarios. The main drawback to sort-first approaches is the difficulty of achieving data scalability. As illustrated in Fig. 3, for different viewing conditions each node may require completely different parts of the data to render their respective portions of the image-space. An important advantage of the sort-first rendering is the ability to adapt a number of algorithms that are not efficient (or sometimes even feasible) with a sort-last approach. Algorithms that require information from a previously rendered sample on a ray may require too much synchronization when the rays are split up among different rendering nodes. Sort-first rendering can keep one set of rays local to each machine and thus allow for such algorithms to be utilized efficiently.



(a) (b)

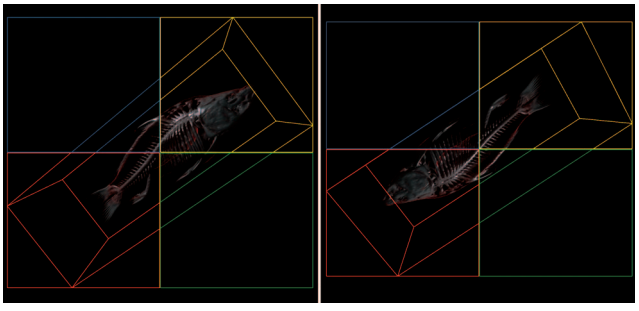
Fig. 2. An example of a sort-last (object-space) distribution scheme with four nodes. (a) Global view of the data set with each node using a different color when rendering the bounding box of their respective portions of the object-space. (b) Zoomed view which illustrates the problem of load balancing with a static object-space distribution (only the green and red nodes are doing work).

4 Data-Scalable Sort First Volume Rendering

There are several issues that must be addressed in order to make sort-first rendering viable for data-scalable volume rendering. These include efficiently rendering and loading portions of a subdivided data set, caching portions of the data set in an intelligent manner, and providing a consistently good level of load balancing.

4.1 Bricking

The data set is divided into a uniform grid of evenly sized bricks based on a user-defined parameter for the size of the bricks. A bit mask corresponding to the scalar values that



(a) (b)

Fig. 3. An example of an image-space distribution scheme with four nodes. Each node colors their image-space bounding rectangles and the bounding box of the volume with a different color. For the two viewpoints used to make the images, the data that each node needs to render are completely different.

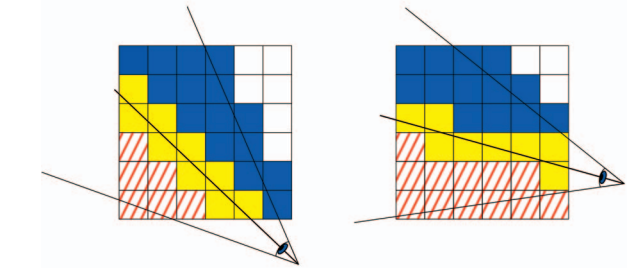


Fig. 4. A 2D illustration of how bricking allows data scalability albeit with a memory overhead. The red hatched bricks are loaded into textures by the node with the left-view frustum, the solid blue bricks are loaded by the node with the right-view frustum, and the solid yellow bricks must be loaded by both. As the viewpoint changes ((a) versus (b) image) the bricks required by each node can change.

occur within each brick is computed so that transparent ones can be culled quickly. Data scalability is achieved by having each rendering node load only the bricks intersected by its view frustum, as illustrated by a 2D example in Fig. 4.

When choosing a brick size, we must balance the benefits of having a finer granularity in object-space and the increased overheads from having a larger number of bricks. A finer granularity reduces data replication between rendering nodes along shared planes of the nodes' view frustums. This replication is illustrated in Fig. 4. However, there is a texture size overhead for each brick since adjacent bricks must share one data value at every point on their border so that the trilinear interpolation is consistent across bricks. When using a preintegrated transfer function [32], two data values must be replicated so that one can access the values for the backsides of the slabs at the boundary. When bricks are culled based on the transfer function, having a finer granularity can allow us to perform a more accurate culling, thus reducing the rendering workload and the amount of data that must be loaded. A hierarchy of brick sizes has often been used to help balance these factors but, in turn, has its own associated overheads.

A significant per brick performance overhead (when using slice-based rendering) is the increased number of vertices that must be generated on the CPU, and sent to the GPU, for the proxy geometry of each brick. To tackle this issue a slice templating technique [1] is used to generate a single set of slices that can be used to render every brick of the same size. This reduces the amount of computation on the CPU as well as the amount of data that must be transferred to, and stored on, the GPU.

4.2 Caching

The amount of bricks that need to be loaded on any given frame depends on the size of the frustum relative to the bricks, the level of frame-to-frame coherence, the viewing conditions, and the method of caching bricks. Since the size of the frustums decreases as processing units are added, the number of bricks that need to be loaded also decreases (as long as the frustums are larger than a couple of bricks). Frame-to-frame coherence is usually a fairly safe assumption in interactive volume rendering, with the exception being time-varying data, which has to be loaded on every frame anyway. When the camera is rotating around the volume from a distance, the amount of data loading is much higher than when the camera is zooming in or panning.

Two caching schemes are investigated in this paper. The simplest method of caching bricks is to load them as they intersect the frustum and, once memory runs out, start swapping bricks out in LRU order. The LRU method is simple to implement but cannot take advantage of asynchronous loading and it typically suffers from sudden spikes in the amount of loading that must be done in a frame. If there is good frame-to-frame coherence, then the bricks that will be needed in upcoming frames can be predicted and loaded ahead of time. One way of doing this is to cache bricks that are in close proximity to the frustum but not yet intersecting. The proximity caching method approximates the frustum with a cone and records the distance from the center of each brick to the surface of the cone. The bricks that are farthest away from the frustum can then be swapped out of memory and the bricks that are closest can be precached. A user-specified limit on how many bricks can be precached in a frame prevents spikes in loading. A more advanced prediction technique may favor bricks on a certain side of the frustum based on a prediction of the camera movement.

While some general-purpose GPU programming APIs allow simultaneous data transfer and kernel execution on the GPU, OpenGL currently lacks this capability. With OpenGL transfers to the GPU can only be asynchronous on the CPU side. Even without this capability there is still a benefit to predictive caching to the GPU. Performance can be stabilized by spreading the loading costs across multiple frames instead of having large spikes in the amount of data that needs to be loaded in a single frame.

Since each processing unit in our cluster has as much system memory as the entire cluster's combined GPU memory, we only have a single layer of caching between the two. Since the data set is replicated in each nodes system memory there is no cache synchronization between nodes. This is not a limitation of our sort-first approach but rather a limitation of our implementation. If we were to scale our cluster further, and render very large data sets, we would either need more system memory or a second layer of caching. The second caching layer would swap data in from network or storage devices to system memory. The bandwidth over gigabit ethernet is significantly less than the bandwidth to the GPU but-high speed networks can provide more bandwidth than OpenGL texture uploads. The amount of system memory available for caching is also much larger and the loading could be asynchronous to the rest of the rendering process. Out-of-core rendering algorithms employ similar predictive caching methods between storage devices and system memory like we do between system memory and GPU memory. Varadhan and Manocha [33] implemented priority-based prefetching of large geometric data from disk with respect to level-of-detail. Precaching data from disk storage by taking advantage of frame-to-frame coherence was presented by Corrêa et al. [34] in a visibility-based approach.

In a future implementation, both layers of caching could be combined with the goal to take maximum advantage of asynchronous loading rather than just reducing spikes in the amount loaded per frame. Provided that there is

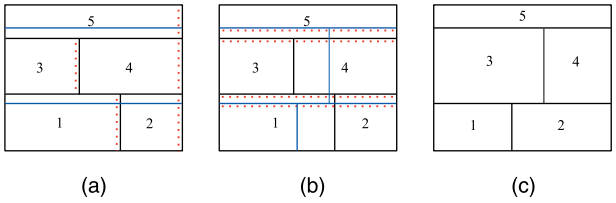


Fig. 5. An illustration of the communication pattern for computing the load balancing in parallel with five processing units. The distribution of the screen space among processing units is illustrated by the numbered regions with black outlines. (a) Shows how only the right most column of each node's SAT (corresponding to the columns of pixels highlighted with a red dotted lines) needs to be communicated in the first stage so that the new horizontal split lines (solid blue lines) can be found. (b) Shows the rows of SAT information (highlighted with red dotted lines) needed in the second stage to find the new vertical split lines (solid blue lines). (c) Shows how the new split lines provide the image decomposition for the next frame.

adequate bandwidth, the asynchronous loading would incur no additional overhead.

4.3 Consistent Load Balancing

In order to achieve data-scalable sort-first rendering, a predictable load balancing method is required. The most commonly used load balancing method uses a simple heuristic where processing units that were slower in the previous frames are given less screen space and faster units are given more. While this is simple to implement and can give acceptable load balancing in some situations, it does not give predictable or consistent results.

We utilize our previously published load balancing method that strictly uses data from the current frame to compute the cost of each pixel (or block of pixels) [1]. A summed area table (SAT) of the pixel costs can then be used to divide the screen into regions of equal cost which are assigned to the processing units. In Section 6.2, we show that this cost-based technique provides a more consistent load balancing than the heuristic method, even when frame-to-frame coherence is good. As the frame-to-frame coherence decreases, the cost-based method outperforms the heuristic method by an increasing margin.

The trade-off for the cost-based load balancing technique is an increased computational overhead needed for the evaluation of the cost estimate. One way to decrease this overhead is to have each processing unit compute the pixel cost and SAT for a portion of the screen. In our original work, the entire SAT is then gathered at a single processing unit which computes the screen space distribution. Much less of SAT actually needs to be gathered if the communication is done in stages. The number of stages of communication corresponds to the number of levels in the hierarchy used to divide the screen space. A two-level mesh hierarchy is proposed to divide the screen, which allows us to communicate just a couple of rows and columns of the SAT in two stages. This communication pattern is illustrated in Fig. 5 by an example.

While a one-level hierarchy (horizontal or vertical strips) would be the most efficient for parallelizing the load balancing computations, we want to use the same screen decomposition to distribute both the rendering and the load balancing so that the load balancing computations are well distributed. A one-level hierarchy would cause significantly more data loading for our data-scalable sort-first distribution since the amount of loading increases with the surface area of the sides of the processing unit's frustums (which is determined by the length of the viewport's perimeter). A two-level hierarchy (a two-dimensional ragged array) is chosen since it reduces the communication requirements of the load balancing algorithm while keeping the surface area of the sides of the view frustums small.

5 Ray Coherent Algorithms

One of the main reasons for exploring sort-first approaches to data-scalable parallel volume rendering is due to their compatibility with many volume rendering algorithms. In this section, we focus on existing single-GPU algorithms that benefit from ray coherence when adapted to a parallel environment. All of the algorithms benefit from keeping the information and processing along each ray local to a single processing unit.

The first algorithm, discussed in Section 5.1, takes advantage of the locality of all the information along a viewing ray in order to perform visibility culling. In contrast, each processing unit in a sort-last distribution can only cull data from its local portion of the data set. This can be very inefficient: imagine the case where some units' portions of the object-space are completely occluded by data on the other units. The second algorithm, discussed in Section 5.2, is an image-space shadowing algorithm that alternates between rendering from the light's and camera's point of view. To parallelize this algorithm, we must do a sort-first decomposition from the light's point of view, so that all the information along the shadow rays is available locally. Last, in Section 5.3, we discuss other algorithms that could benefit from ray coherence when being adapted to a parallel environment.

5.1 Visibility Culling

It has long been observed that many of the fragments processed when rendering a volume do not contribute anything to the final image (see [35]). Typically, these fragments are separated into two groups: fragments that have zero opacity (empty fragments) and fragments that are occluded by one or more fragments which have a total opacity of one (occluded fragments). Skipping empty fragments is easily supported by culling empty bricks, which is possible in sort-first and sort-last volume rendering alike.

However, early ray termination of occluded volume elements introduces view dependency and, therefore, is not effective for sort-last volume rendering. We propose two methods of avoiding processing of occluded parts of the data in sort-first rendering. The first is a direct adaptation of an existing single-GPU approach [6], [7], [8] which uses the early depth culling ability of GPUs to speed up the processing of occluded fragments. The culled fragments still have some, though greatly reduced, processing cost and per brick overheads cannot be avoided. The second method uses the occlusion query feature on GPUs to test if entire bricks are completely occluded. Loading and rendering of the occluded bricks can then be avoided for that frame.

The early depth culling feature on GPUs uses the depth buffer to mask regions of the screen for which the fragment shader should not be executed. While originally designed to speed up rendering of occluded surfaces, it has also been used to speed up the rendering of occluded volume data [6], [7]. Periodically doing an extra pass to update the depth buffer incurs an overhead proportional to the number of updates (and to a lesser degree, the number of pixels updated). An extreme case of frequent updates is from Ruijters and Vilanova who update once for every brick in a subdivided data set [8] by rendering the front faces of each brick's bounding box into the depth buffer before rendering the volume inside that brick.

Since many bricks do not overlap at all in image-space, we have found that it is beneficial to update the depth buffer less frequently. Therefore, we render a chunk of bricks at a time, and update the depth buffer in between each chunk. Since we cannot capture any occlusion happening between bricks in the same chunk, we would like the bricks in a chunk to be spread out over the image-space rather than overlapping. We can achieve this by generating our front to back order slab by slab, where we choose the set of slabs perpendicular to the axis most aligned with the view direction. The ideal size for the chunks depends on the data set and brick size, hence it must be chosen accordingly.

While reducing the number of update passes is going to have the biggest effect on performance, we also aim at minimizing the cost associated with each update pass. To do this, we do not change the render target (as is required in multipass ray casting [6], [7]) but instead just disable color output for the update pass. The number of pixels processed in the update pass is reduced by keeping track of an approximate image-space bounding box for each chunk of bricks.

In conjunction with early depth culling to kill occluded fragments, we use the occlusion query feature on GPUs to cull full bricks which are completely occluded. Occlusion queries allow a program to know how many fragments were actually rendered (passed the depth test) for a group of primitives. Thus, if we were to render the bounding box of a brick and we get a fragment count of zero, then we know that the brick can be skipped entirely. This results in a small additional increase in rendering performance but it also allows bricks to not be loaded.

5.2 Volumetric Shadowing

Shadowing effects can provide an additional depth cue to a user exploring a volumetric data set. In the past, this was done by creating a corresponding shadow volume which describes the amount of light arriving at any point in the data [36]. Computing such a shadow volume is expensive and must be done every time that the light position or transfer function changes. The ability to interactively change the light position and transfer function is key to efficient volume exploration. Also, shadow volume approaches can suffer from attenuation leakage due to insufficient resolution and increased memory requirements.

A recent image-space approach to volume shadowing avoids these problems [4], [5]. Instead of rendering the slices so that they are aligned with the camera, they are aligned with the half angle between the camera and the light. This allows the same slice to be rendered from both the camera's and light's point of view. We can then render the volume slice by slice, with each slice being rendered first from the camera's point of view, and then from the light's. When the next slice is rendered from the camera's point of view, the previous result from the light's point of view is mapped as a texture. The opacity of this texture then tells us how much light has been attenuated thus far. This approach allows for interactive updates of the light and transfer function, requires far less memory, and avoids issues with attenuation leakage. By combining this image-space shadowing algorithm with a hybrid partitioning scheme, we are able to perform interactive shadowed volume rendering on data sets which are too large to fit on a single GPU.

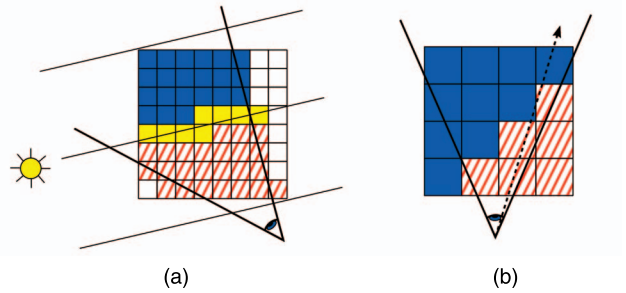


Fig. 6. (a) A 2D illustration of the hybrid partitioning. The light frustum is divided into two pieces and the solid blue bricks are rendered by the unit with the left frustum while the red hatched bricks are rendered by the unit with the right frustum. The bricks that are solid yellow must be clipped against the shared plane of the two frustums and each unit renders their respective portions. (b) Shows how the staircasing effect that occurs if the bricks are not clipped can create cycles in the compositing order. A viewing ray (shown with the dashed black line) passes from one unit's set of bricks to the other's and back again.

5.2.1 Hybrid Partitioning

In the same way that we exploited the coherence of viewing rays for performing visibility culling, we can use a sort-first distribution of the light's image-space in order to make the light rays coherent on each processing unit. The screen space for the light map is divided into regions and the corresponding frustums of each unit are intersected against the bricks. The bricks that intersect the light frustum must also be rendered from the camera's point of view. Obviously, when the camera's view is not perfectly aligned with the light's, the intermediate images produced by each node will overlap. Therefore, as in sort-last partitioning, we require a compositing stage to combine the samples along the viewing rays and create the final image. A 2D version of this hybrid partitioning scheme is illustrated in Fig. 6a.

The processing units cannot just render their portions of the data brick by brick as we have done for standard volume rendering. For many viewing conditions, there is no ordering of the bricks that will give correct compositing results for both the light and the camera. While this could potentially be overcome by rendering sets of bricks into different buffers and then combining the results, this would add significant complexity and computational overhead. Instead, we process the data slice by slice by consecutively rendering the pieces of each slice from each of the bricks it intersects. This incurs a significant overhead since it requires

Fig. 7. An example of shadowed volume rendering on two processing units. The left and center images show the intermediate images created by each of the processing units. The right image shows the final composited result with the brick outlines drawn in different colors by each processing unit to show the data distribution.

us to change some of the rendering state, such as the current texture, for every piece of every slice. Due to this additional overhead, the ideal brick size is much larger for shadowed rendering compared to standard volume rendering.

Bricks shared by neighboring processing units require special attention. It is usually not possible to just assign whole bricks to one node or the other and have a valid compositing order. Although the bricks themselves are convex, the set of bricks that are intersecting a unit's frustum are likely to have concavities due to staircasing. As shown in Fig. 6b, this creates cycles in the compositing order whenever viewing rays cross from one unit's set of bricks into another's and then back into the first unit's set again. Therefore, we clip the bricks with the frustum planes to create convex pieces, and have each unit render their respective portions of the shared bricks. The clipping can be seen in Fig. 7, which shows the results from a shadowed rendering with two processing units.

5.2.2 Direct Send Compositing

The compositing stage uses direct send compositing, due to its simplicity and efficiency when handling nonpower of two numbers of nodes. Binary swap compositing requires some processing units to remain idle for the first compositing stage if the number of units is not a power of two. For our method of distributing the screen space (as described in Section 4.3), both the number of rows and the number of columns in the screen space distribution would have to be powers of two in order to have no idle units during binary swap compositing. This is because, in order to avoid cycles in the compositing order, we must composite the images from the units that belong to the same row in the image-space distribution before we can composite the results from the different rows.

5.3 Other Potential Algorithms

Most algorithms which process the data along a set of rays are going to benefit from ray coherence when adapted to a sort-first parallel distribution. Even standard emission absorption volume rendering performance can benefit from ray coherence through reduced communication overheads. In particular, sort-first parallelization leads to good scalability with the number of pixels, e.g., for large displays. Algorithms that benefit from reduced synchronization overheads are the most interesting. In the case of the shadowed rendering with half angle slicing, the synchronization required for an object-space distribution is much greater than what is required by our approach.

Another potential application of ray coherent parallelization is the rendering of spectral effects like inelastic scattering and selective absorption [37], [38]. These effects depend of the spectrum of light traveling along the rays, which changes as the ray steps through the volume. Therefore, the rays need to be coherent to avoid synchronization problems between processing units.

Other examples are techniques that utilize depth peeling, such as opacity peeling [39] and feature peeling [40], for visualizing nested structures within the volume data. These techniques split the volume into layers based on some criteria which is evaluated as the rays pass through the volume. Both approaches require information along the ray to be available locally if adapted to a parallel environment.

Using an image-based metric for level-of-detail techniques [41] results in higher quality images than those acquired with a level-of-detail algorithm using an object-based metric. The reason that image-based metrics are superior, is that they can take the visibility of a brick into consideration when choosing its level-of-detail. The downside is that the image-based metric must be periodically recomputed as the viewpoint changes, which can be an expensive task. In a parallel environment, a ray coherent workload distribution would allow each processing unit to compute the image-space metric for the bricks they are rendering without any additional communication or synchronization.

6 Implementation and Results

This section provides a detailed performance analysis of each stage of our sort-first approach to data-scalable parallel volume rendering. We first look at the isolated results of each of the algorithms and techniques that we use, and then we put it all together and show the overall performance on a large real-world data set. Our results show that proximity caching, cost-based load balancing, and visibility culling allow data-scalable sort-first distributions to provide better

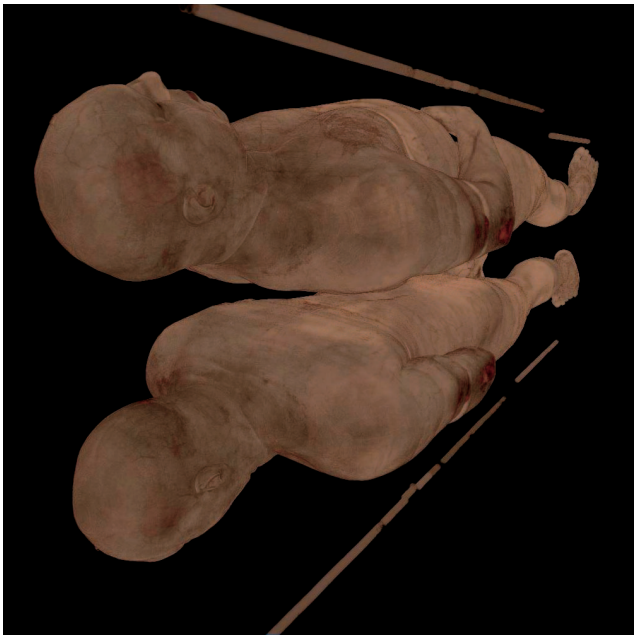


Fig. 8. The mirrored version of the visible male data set ($2{,}048 \times 2{,}048 \times 1{,}878$) using a high-opacity transfer function.

performance than sort-last distributions in many scenarios. The results for our visibility culling techniques show improved performance compared to previously published approaches and better efficiency when combined with a sort-first rather than sort-last distribution. Our shadow rendering algorithm is shown to allow for real-time rendering of data sets that are larger than the memory available to a single GPU. The overall performance results from rendering with various data sets, image resolutions, and transfer functions show that there are still plenty of performance gains possible if more processing units were utilized.

We focus on the results for multiple nodes here and refer to the Appendix, provided as supplemental material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TVCG.2010.116, for details on single node performance. Appendix A.1 gives details about the hardware and software used in our parallel rendering environment. Our implementation is based on C++, OpenGL, and GLSL (for GPU programs). The environment of the performance measurements for this paper consists of 32 workstations with 2.3 GHz quad-core AMD Opteron Processors (Shanghai architecture) and Nvidia GTX 285 GPUs connected over a PCI-E bus. In Appendix A.2, we discuss the transfer functions and the data sets used in our experiments. Most experiments use the visible male data set ($2{,}048 \times 1{,}024 \times 1{,}878$) or a double size version ($2{,}048 \times 2{,}048 \times 1{,}878$) with two copies mirrored back to back as shown in Fig. 8.

6.1 Rendering and Caching Bricks

We first consider the baseline performance of a single node for rendering a data set that has been divided into a number of different brick sizes. We minimize the overhead of generating slices for rendering bricks by using slice templating [1]. In Appendix A.3, we show that for a relatively small 512^3 data set the minimum brick size to be around 70^3 to 80^3 in order to avoid large performance overheads. In Appendix A.4, we show that culling empty bricks almost doubles the rendering performance for the transfer functions used to test the visible male data set. For this larger data set, we get optimal performance with a brick size between 115^3 and 125^3.

TABLE 1
Comparison of LRU and Proximity Caching Showing the Percentage of Frames of an Animation Where the Cache Misses Exceed the Threshold for the Maximum Number of Bricks to Preload with the Proximity Caching Algorithm

	Average Percentage of Frames above Threshold			
	Nine Render Units		Sixteen Render Units	
Threshold	LRU	Proximity	LRU	Proximity
2	27.2%	15.3%	25.3%	11.5%
4	19.5%	9.0%	17.0%	6.2%
6	12.9%	6.3%	11.6%	4.3%
8	10.4%	4.4%	8.9%	2.5%
10	9.3%	2.0%	6.8%	1.1%

The bricks also have to be uploaded to and cached on the GPU in an efficient manner. In Appendix A.5, we look at the bandwidth achieved with different texture formats and brick sizes. Since the bandwidth is relatively constant for the range of brick sizes that give acceptable rendering performance, this has little effect on our choice of brick size. With a brick size of 125^3 and single component textures the bandwidth is just under one gigabyte per second.

Provided that we have frame-to-frame coherence, the average number of bricks loaded on a frame will be quite low. However, the number of bricks being loaded on any single frame can be quite high. This is because the loading occurs in spurts where many bricks are loaded on one frame and then none are loaded on the next several frames. This is undesirable when trying to interactively explore a data set because of the sudden slowdown when a spurt of loading occurs. To combat this, we load some of the bricks in close proximity to the frustum on frames where the loading requirements are small. This will result in more bricks being loaded in total but in a more consistent fashion.

We compare the proximity caching algorithm to the naive LRU caching algorithm that just loads bricks as they intersect the frustum. We render the visible human data set with a brick size of 125^3. This results in 1.6 gigabytes of data after culling empty bricks. We set the maximum amount of texture memory to be used as a buffer by each render node to be 850 MB. For these tests, we use a recorded animation of a user exploring a data set. The animation includes rotation, panning, and zooming motions. We look at how many frames in the animation need to load more bricks than the threshold value. The results are compiled into Table 1.

The threshold value is the limit on the number of bricks being preloaded in the proximity caching algorithm. The results from using 9 and 16 render units are shown. We take the average number of frames above the threshold among all the render units. We can see that even with a preloading threshold as low as four bricks per frame the number of frames where a spurt of loading occurs is cut in half compared to LRU. With a threshold of 8 to 10 bricks per frame the loading spurts are almost eliminated. Finally, we can see that when we increase the number of rendering

TABLE 2
Comparison of the Cost-Based and Performance-Based Load Balancing Algorithms with Different Levels of Frame-to-Frame Coherence (Render Times in Milliseconds)

	Average Render Time Deviation	
Frames Skipped	Performance Based	Cost Based
0	5.67 ± 2.54	7.55 ± 2.23
2	12.39 ± 4.42	6.87 ± 1.06
4	18.08 ± 6.65	6.93 ± 1.18

nodes the data loading requirements decrease in tandem with the size of each render unit's frustum.

6.2 Load Balancing

In order to quantify how well the load balancing works, we take the difference between the render times of the fastest and slowest processing unit for each frame and average this over all frames. Using this metric, we compare the cost-based load balancing method to the heuristic performance-based method (that uses the rendering times from the previous frame) to see which one gives better and more consistent results. We render the visible human data set with 16 render nodes and a 2 megapixel image. We show the results for three different levels of frame-to-frame coherence in Table 2. The results are the average deviation in render time among processing units listed in milliseconds (lower is better). We also show the standard deviation to illustrate the consistency of the load balancing methods. To get different levels of frame-to-frame coherence, we take a prerecorded animation with good coherence and then skip some number of frames. With good frame-to-frame coherence, the performance-based load balancing gives slightly better results since it also balances the per brick overheads. However, with even a moderate decrease in frame-to-frame coherence, the cost-based load balancing gives much better and more consistent results. Using the communication scheme discussed in Section 4.3, we find that our cost-based load balancing has just a few milliseconds of overhead regardless of the image resolution or the number of render nodes.

6.3 Visibility Culling

In order to determine the ideal update frequency for the depth buffer, we must compare the amount of overhead incurred by each update to the additional culling achieved. As shown in Appendix A.6, the overhead grows rapidly with the number times the depth buffer is updated, but the performance is more than doubled, compared to rendering without visibility culling, with as little as six updates to the depth buffer. With 28 updates to the depth buffer, we lose less than one percent off the peak performance while incurring less than 2.5 milliseconds of overhead when no occlusion occurs. We also find that occlusion queries incur less than 0.5 milliseconds of additional overhead and can provide an eight percent increase in performance. The performance increase from the occlusion queries is small compared to the fragment culling, however, it would be possible to avoid loading bricks that are completely culled and the visibility information gained from the queries could be used for things like level-of-detail selection.

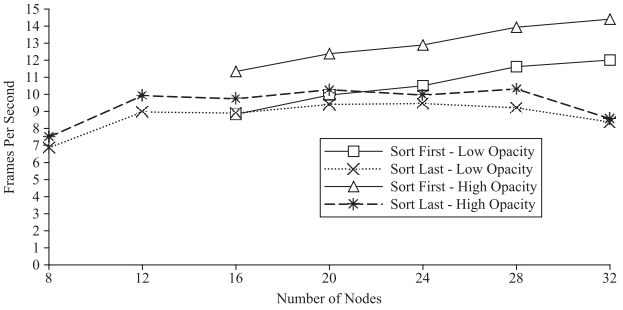


Fig. 9. The performance scaling for sort-first and sort-last distributions using the visible male data set ($2{,}048 \times 1{,}024 \times 1{,}878$). Results for both a high and low-opacity transfer functions are shown to illustrate the effect of visibility culling. For sort-first, a minimum of 16 nodes is necessary to meet memory requirements.

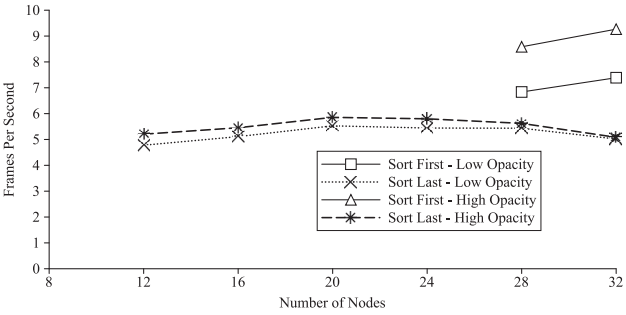


Fig. 10. The performance scaling for sort-first and sort-last distributions using the mirrored visible male data set ($2{,}048 \times 2{,}048 \times 1{,}878$). Results for both a high and low-opacity transfer functions are shown to illustrate the effect of visibility culling. For sort-first, a minimum of 28 nodes is necessary to meet memory requirements.

6.4 Sort-First versus Sort-Last

Doing a direct comparison of sort-first and sort-last distributions is difficult since many parameters influence rendering performance. We chose a static sort-last distribution and view the data set globally from a slight distance. We also disable empty brick culling to minimize the load imbalance in order to create fair conditions. Even though it does not account for occlusion, we use the pixel-cost-based load balancing for the sort-first experiments since it provides better data scalability. The brick size is set to 125^3 and the slice spacing is half the sample spacing.

Our alpha compositing algorithm does not use any compression but we do limit image transfers to the image-space bounding box of the data set. For the axial rotations used in our tests, this provides an effective culling of empty pixels. We also undersample the data in the image-space by using a 3 megapixel image size, which reduces the compositing cost for sort-last. The rotation animation also causes significant data loading for sort-first. We show the performance scaling for both distributions using the visible male and the larger mirrored visible male data sets in Figs. 9 and 10, respectively. The memory overhead inherent to the sort-first distribution (shown previously in Fig. 4) causes a larger number of nodes to be required for rendering each data set.

While we do not cull empty bricks we do use visibility culling. We use two different transfer functions in the experiments, one has a relatively high opacity and thus produces isosurface-like images while the other has a relatively low opacity and produces more cloud-like

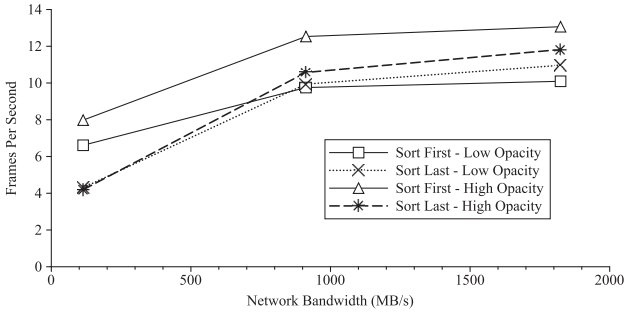


Fig. 11. Projected performance for rendering the mirrored visible male data set ($2{,}048 \times 2{,}048 \times 1{,}878$) with a 4 megapixel resolution and different network interconnect bandwidths.

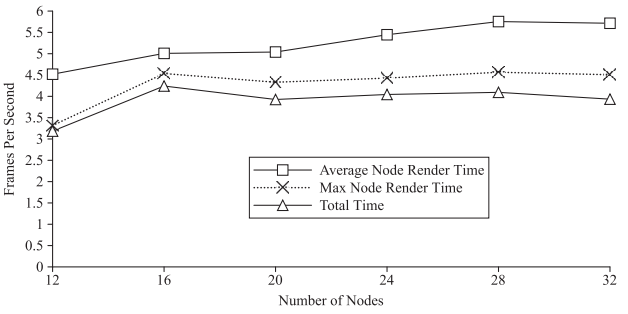


Fig. 12. The performance scaling of shadowed rendering with the visible male data set ($2{,}048 \times 1{,}024 \times 1{,}878$). A minimum of 12 nodes is necessary to meet memory requirements.

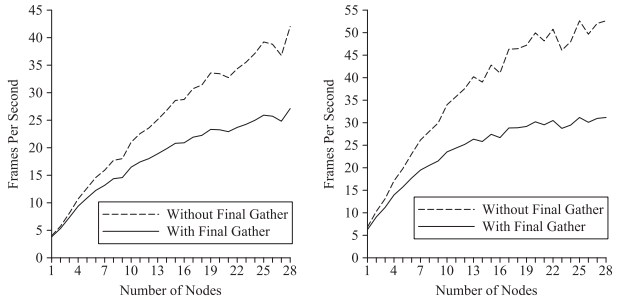


Fig. 13. The scaling results for a reduced portion of the visible male data set ($2{,}048 \times 1{,}024 \times 400$) with and without the final gather time included. The results for the low and high-opacity transfer functions are on the left and right, respectively.

images. We expect the sort-first approach to achieve a similar speedup from visibility culling regardless of the number of render nodes. It is clear that this is true for both the small and large data sets. With sort-last, the speedup clearly diminishes for the larger data set as the number of nodes increases. However, for the smaller data set, the speedup is already almost gone for as little as eight nodes. With the high-opacity transfer function, sort-first is consistently faster than sort-last and even with the low-opacity transfer function it is always at least as fast. It is important to note that the compositing cost increases with the data size due to the larger screen foot print.

Fig. 11 shows an estimate of how network bandwidth would effect the performance of the sort-first and sort-last distributions. We render the mirrored visible human data set with a 4 megapixel image size and 32 render nodes. We then scale the network transmission portions of the total time by the expected bandwidth. The positions of the data points along the x-axis correspond to gigabit ethernet and both single and dual data rate Infiniband. With gigabit ethernet, the sort-first still gives interactive frame rates while sort-last does not. With single data rate Infiniband sort-first matches or beats the performance of sort-last depending on the transfer function. With dual data rate Infiniband sort-first is slightly worse or slightly better than sort-last depending on the transfer function.

6.5 Volumetric Shadowing

The shadowed rendering algorithm must render the data one slice at a time rather than one brick at a time. This means that each slice must be rendered as a collection of smaller pieces from all the bricks that the slice intersects. For each piece of each slice, we are required to change some of the rendering states such as the texture that is bound and the transformation matrix. This incurs a much greater per brick overhead which we discuss in Appendix A.7. With different data sets and transfer functions, it is possible to see a net increase in performance from culling empty bricks, but the gain is likely to be small for all but the most extreme circumstances.

We also need to avoid rendering the parts of the data that are outside the light's frustum when we render from the camera's point of view. We use the clip planes built into OpenGL so that the processing cost for the culled fragments is reduced. Slice templates cannot be used since they also utilize the clip planes, but the vertex generation costs are already reduced due to the larger brick size and distance between slices.

We look at how well the performance scales when we use multiple processing units to render the visible human data set. In Fig. 12, we show the scaling results for a brick size of 230^3 and a 1 megapixel image size. We show three curves: the average node render time, the maximum node render time, and the total render time. While the average node render time continues to scale as we add render nodes, the maximum node render time and the total render time peak with just 16 render nodes. This is because we do not have an appropriate load balancing algorithm that will account for the increased per brick overheads. While the performance-based load balancing would automatically account for this, it does not provide reliable enough data scaling. A modification to our cost-based load balancing which also considers the number of bricks being rendered by each node should provide better scaling results. A better network interconnect would also help by reducing the alpha compositing overhead.

6.6 Overall Performance

The previous parts of this section mostly document and compare the isolated impact of different rendering techniques. Now we look at the overall performance and scalability of our sort-first approach. In Fig. 13, we show the full scaling results for rendering a reduced portion of the visible male data set with 1 to 28 nodes. To keep the final gather time small, we render to a 1 megapixel viewport. In order to provide a sufficient workload, we do not cull invisible bricks and we set the slice spacing to be one eighth of the sample spacing. We compare the scaling results with and without the final gather time and with both

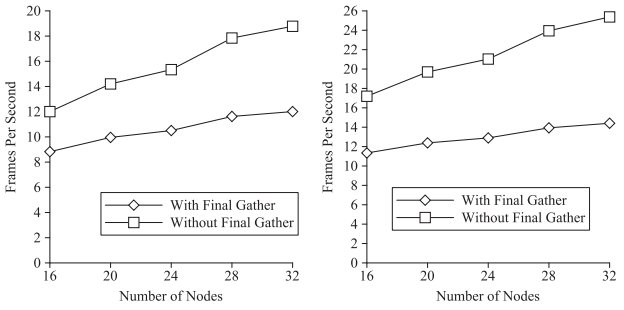


Fig. 14. The scaling results for the visible male data set ($2{,}048 \times 1{,}024 \times 1{,}878$) with and without the final gather time included. The results for the low and high-opacity transfer functions are on the left and right, respectively.

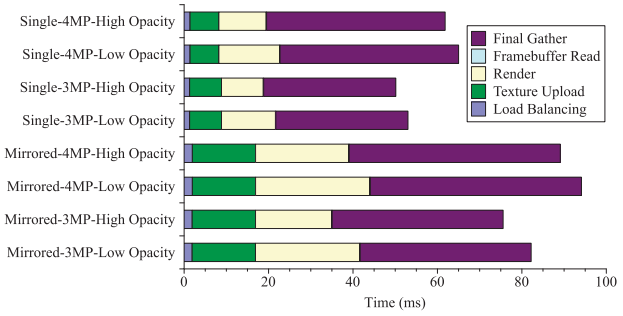


Fig. 15. A detailed breakdown of how the processing time is split up among the different stages of the parallel rendering pipeline. All four experiments are shown for two different data set sizes.

the low and high-opacity transfer functions. It is clear that the final gather time is hampering the performance scaling despite the relatively low-image resolution. Saturation effects [24] are also coming into play with the high-opacity transfer function due to the reduced rendering workload. In Fig. 14, we look at the effect of the final gather time for a larger data set and image resolution. We render the full visible male data set with a 3 megapixel image resolution using 16 to 32 nodes. We do not cull empty bricks and we set the slice spacing to be half the sample spacing. The results without the final gather time show the upper bound for the scalability of our approach with the given data set and rendering parameters. With image compression or a higher speed network, the scaling performance of our algorithm would approach this upper bound.

Finally, we explore the total performance of our sort-first rendering system using all 32 nodes and different data set sizes, image resolutions, and transfer functions. We use the full visible male data set as well as the mirrored version of this data set. We use the same high and low-opacity transfer functions as in Section 6.4. The transfer functions are designed so that the same number of empty bricks are culled for both the high and low-opacity versions. The camera is placed at a distance that tries to maximize the size of the data on the screen while keeping the frustum culling to a minimum. The animation rotates the camera around the data set at a constant rate which causes significant data loading.

We render into 3 megapixel ($1{,}772^2$) and 4 megapixel ($2{,}048^2$) images with the slice distance set to be half the largest grid distance. We cull occluded fragments with the depth test using about 28 updates to the depth buffer for each frame. Fully occluded bricks are culled using occlusion queries. We use our pixel-cost-based load balancing technique for calculating the cost with the resolution set to one quarter of the image resolution. The pixel cost load balancing cannot account for occlusion, but it allows us to consistently render larger data sets than what is possible with the performance-based load balancing. With the performance-based load balancing it is not uncommon for the screen distribution to jump around and require a processing unit to render more data than it can store in texture memory. We precache at most 10 and 20 bricks in each frame for the smaller and larger data sets, respectively. With the brick size set to 125^3 we have just under one gigabyte per second of bandwidth, resulting in about two milliseconds of overhead for every brick loaded. This compares favorably to the cost of alpha compositing in our parallel environment which is required for sort-last distributions.

We show a detailed breakdown of the average performance for all data sets, transfer functions, and image resolutions in Fig. 15. We average over all 32 nodes and all frames of the animation. The visibility culling results in better render times for the high-occlusion versus low-occlusion transfer functions. The load balancing time is consistently just a few milliseconds for both data sets and image sizes. The data loading time essentially doubles with the size of the data set but remains smaller than the rendering time even for these relatively low-image resolutions. The frame buffer readback is essentially inconsequential on PCI-E and scales linearly with the number of processing units when doing sort-first. With gigabit ethernet the total time is dominated by the final gather compositing. If we were to add more processing units we would expect the load balancing and final gather times to remain the same and all the other times to decrease. Therefore, we would expect to continue to see performance scaling for these data sets as we add more processing units, especially for higher image resolutions. The final gather time will eventually dominate the total time when using gigabit ethernet.

7 Conclusion and Future Work

The utility of sort-first workload distributions for parallel volume rendering has been demonstrated. We have shown that data-scalable sort-first distributions can outperform sort-last distributions in many scenarios. Our proximity caching algorithm reduces spikes in loading and could reduce the loading overhead when asynchronous transfers are possible. We have improved the parallel computation of a load balancing algorithm and demonstrated the load balancing algorithm we use is superior to the alternatives. Most importantly, we have shown how the locality of the data and processing along rays afforded by a sort-first distribution can allow for efficient adaptations of many existing volume rendering algorithms to a parallel environment.

This includes an improved visibility culling technique which provides a good speedup when occlusion occurs and almost no overhead when it does not. We have shown how visibility queries can provide some modest performance increase as well as visibility information about bricks which could potentially be used to reduce loading or select a level-of-detail. Volumetric shadowing can also be performed in

parallel on data sets that are too large for a single GPU using our hybrid sort-first and sort-last distribution. We also describe a number of other algorithms and techniques that could require or at least benefit from a sort-first distribution.

The proximity-based caching algorithm could be improved by considering the camera movement in the previous frames and trying to predict where the camera will move in the next frame. The caching overhead could also potentially be reduced by waiting to load bricks until after the occlusion queries are done, so that the loading of occluded bricks can be skipped. Ideally, the visibility of bricks could then also be considered when prioritizing bricks to be cached. Our current implementation is limited to rendering data sets that are not larger than system memory. However, we plan as a future work, a second layer of caching that asynchronously prefetches data from local storage (as with out-of-core algorithms) or over the network.

The early ray termination techniques that we described could show even greater benefits when used with more expensive rendering techniques. This includes out-of-core rendering, compressed volume rendering, rendering with higher order interpolants, and much more. Compressed rendering is an especially attractive pairing since it could significantly reduce the amount of data that needs to be loaded for the sort-first approach.

An interesting area for future work is a thorough comparison between the sort-first distribution and a sort-last distribution with dynamic load balancing. We suspect that the sort-last distribution could require less data redistribution when the camera is zoomed out and rotating around the data set, while the sort-first distribution could fair better when the camera is zoomed in. Perhaps this could even motivate some sort of hybrid approach that changes based on viewing conditions.

ACKNOWLEDGMENTS

This work was funded in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada and the Deutsche Forschungsgemeinschaft (DFG). The authors would like to thank the following sources for data sets used in the illustrations and experiments:

- Dr. Christof Rezk-Salama, University of Siegen, Germany and Dr. Michael Scheuering, Siemens Medical Solutions, Forchheim, Germany for the fish data set.
- Brown & Herbranson Imaging, Stanford Radiology, and The Rosicrucian museum, for the mummy data set.
- The National Library of Medicine for the visible human data set.

REFERENCES

[1] B. Moloney, D. Weiskopf, T. Möller, and M. Strengert, "Scalable Sort-First Parallel Direct Volume Rendering with Dynamic Load Balancing," *Proc. Eurographics (EG) Symp. Parallel Graphics Visualization (PGV)*, pp. 45-52, 2007.

[2] T.J. Cullip and U. Neumann, "Accelerating Volume Reconstruction with 3D Texture Hardware," Technical Report TR93-027, Univ. of North Carolina at Chapel Hill, 1993.

[3] B. Cabral, N. Cam, and J. Foran, "Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware," *Proc. IEEE Symp. Volume Visualization and Graphics (VolVis)*, pp. 91-98, 1994.

[4] J. Kniss, G. Kindlmann, and C. Hansen, "Multi-Dimensional Transfer Functions for Interactive Volume Rendering," *IEEE Trans. Visualization and Computer Graphics*, vol. 8, no. 3, pp. 270-285, July 2002.

[5] C. Zhang and R. Crawfis, "Shadows and Soft Shadows with Participating Media Using Splatting," *IEEE Trans. Visualization and Computer Graphics*, vol. 9, no. 2, pp. 139-149, Apr.-June 2003.

[6] J. Krüger and R. Westermann, "Acceleration Techniques for GPU-Based Volume Rendering," *Proc. IEEE Visualization (VIS)*, pp. 287-292, 2003.

[7] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl, and W. Straßer, "Smart Hardware-Accelerated Volume Rendering," *Proc. Eurographics (EG) Symp. Data Visualisation (VisSym)*, pp. 231-238, 2003.

[8] D. Ruijters and A. Vilanova, "Optimizing GPU Volume Rendering," *Winter School of Computer Graphics*, vol. 14, pp. 9-16, 2006.

[9] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl, "A Simple and Flexible Volume Rendering Framework for Graphics-Hardware-Based Raycasting," *Proc. Eurographics (EG) Workshop Volume Graphics*, pp. 187-195, 2005.

[10] X. Tong, W. Wang, W. Tsang, and Z. Tang, "Efficiently Rendering Large Volume Data Using Texture Mapping Hardware," *Proc. EG/IEEE TCVG Symp. Visualization*, pp. 121-132, 1999.

[11] C. Müller, M. Strengert, and T. Ertl, "Optimized Volume Raycasting for Graphics-Hardware-Based Cluster Systems," *Proc. Eurographics (EG) Symp. Parallel Graphics Visualization (PGV)*, pp. 59-66, 2006.

[12] T. Klein, M. Strengert, S. Stegmaier, and T. Ertl, "Exploiting Frame-to-Frame Coherence for Accelerating High-Quality Volume Raycasting on Graphics Hardware," *Proc. IEEE Visualization (VIS)*, pp. 223-230, 2005.

[13] E.W. Bethel, G. Humphreys, B. Paul, and J.D. Brederson, "Sort-First, Distributed Memory Parallel Visualization and Rendering," *Proc. IEEE Symp. Parallel and Large-Data Visualization and Graphics (PVG)*, pp. 41-50, 2003.

[14] S. Marchesin, C. Mongenet, and J. Dischler, "Dynamic Load Balancing for Parallel Volume Rendering," *Proc. Eurographics (EG) Symp. Parallel Graphics and Visualization (PGV)*, pp. 43-50, 2006.

[15] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, "A Sorting Classification of Parallel Rendering," *IEEE Computer Graphics and Applications*, vol. 14, no. 4, pp. 23-32, July 1994.

[16] T.-Y. Lee, C.S. Raghavendra, and J.N. Nicholas, "Image Composition Methods for Sort-Last Polygon Rendering on 2D Mesh Architectures," *Proc. IEEE Symp. Parallel Rendering (PRS '95)*, pp. 55-62, 1995.

[17] K.-L. Ma, J.S. Painter, C.D. Hansen, and M.F. Krogh, "Parallel Volume Rendering Using Binary-Swap Compositing," *IEEE Computer Graphics and Applications*, vol. 14, no. 4, pp. 59-68, July 1994.

[18] W.M. Hsu, "Segmented Ray Casting for Data Parallel Volume Rendering," *Proc. Symp. Parallel Rendering (PRS)*, pp. 7-14, 1993.

[19] S. Eilemann and R. Pajarola, "Direct Send Compositing for Parallel Sort-Last Rendering," *Proc. Eurographics (EG) Symp. Parallel Graphics and Visualization (PGV)*, pp. 29-36, 2007.

[20] A. Stompel, K.-L. Ma, E.B. Lum, J. Ahrens, and J. Patchett, "SLIC: Scheduled Linear Image Compositing for Parallel Volume Rendering," *Proc. IEEE Symp. Parallel and Large-Data Visualization and Graphics (PVG)*, pp. 33-40, 2003.

[21] K.-L. Ma, "Parallel Volume Ray-Casting for Unstructured-Grid Data on Distributed-Memory Architectures," *Proc. IEEE Symp. Parallel Rendering (PRS '95)*, pp. 23-30, 1995.

[22] F.R. Abraham, W. Celes, R. Cerqueira, and J.L. Elias, "A Load-Balancing Strategy for Sort-First Distributed Rendering," *Proc. Brazilian Symp. Computer Graphics and Image Processing (SIBGRAPI)*, pp. 292-299, 2004.

[23] U. Neumann, "Communication Costs for Parallel Volume-Rendering Algorithms," *IEEE Computer Graphics and Applications*, vol. 14, no. 4, pp. 49-58, July 1994.

[24] S. Eilemann, M. Makhinya, and R. Pajarola, "Equalizer: A Scalable Parallel Rendering Framework," *IEEE Trans. Visualization and Computer Graphics*, vol. 15, no. 3, pp. 436-452, May-June 2009.

[25] B. Domonkos and B. Csébfalvi, "Interactive Distributed Translucent Volume Rendering," *Proc. Winter School of Computer Graphics (WSCG '07)*, pp. 153-160, 2007.

[26] R. Samanta, T. Funkhouser, and K. Li, "Parallel Rendering with K-Way Replication," *Proc. IEEE Symp. Parallel and Large-Data Visualization and Graphics (PVG '01),* pp. 75-84, 2001.
[27] C. Mueller, "The Sort-First Rendering Architecture for High-Performance Graphics," *Proc. Symp. Interactive 3D Graphics (SI3D),* pp. 75-84, 1995.
[28] B. Corrie and P. Mackerras, "Parallel Volume Rendering and Data Coherence," *Proc. Symp. Parallel Rendering (PRS '93),* pp. 23-26, 1993.
[29] S. Whitman, "A Task Adaptive Parallel Graphics Renderer," *Proc. Symp. Parallel Rendering (PRS '93),* pp. 27-34, 1993.
[30] P. Lacroute, "Real-Time Volume Rendering on Shared Memory Multiprocessors Using the Shear-Warp Factorization," *Proc. IEEE Symp. Parallel Rendering (PRS '95),* pp. 15-22, 1995.
[31] J. Challinger, "Scalable Parallel Volume Raycasting for Non-rectilinear Computational Grids," *Proc. Symp. Parallel Rendering (PRS '93),* pp. 81-88, 1993.
[32] K. Engel, M. Kraus, and T. Ertl, "High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading," *Proc. ACM SIGGRAPH/EG Workshop Graphics Hardware (HWWS),* pp. 9-16, 2001.
[33] G. Varadhan and D. Manocha, "Out-of-Core Rendering of Massive Geometric Environments," *Proc. IEEE Visualization (VIS),* pp. 69-76, 2002.
[34] W.T. Correa, J.T. Klosowski, and C.T. Silva, "Visibility-Based Prefetching for Interactive Out-of-Core Rendering," *Proc. IEEE Symp. Parallel and Large-Data Visualization and Graphics (PVG),* pp. 1-8, 2003.
[35] K. Engel, M. Hadwiger, J.M. Kniss, C.R. Salama, and D. Weiskopf, *Real-Time Volume Graphics.* A K Peters, 2006.
[36] U. Behrens and R. Ratering, "Adding Shadows to a Texture-Based Volume Renderer," *Proc. IEEE Symp. Volume Visualization (VolVis),* pp. 39-46, 1998.
[37] H. Noordmans, H. van der Voort, and A. Smeulders, "Spectral Volume Rendering," *IEEE Trans. Visualization and Computer Graphics,* vol. 6, no. 3, pp. 196-207, July-Sept. 2000.
[38] M. Strengert, T. Klein, R. Botchen, S. Stegmaier, M. Chen, and T. Ertl, "Spectral Volume Rendering Using GPU-Based Raycasting," *The Visual Computer,* vol. 22, no. 8, pp. 550-561, 2006.
[39] C. Rezk-Salama and A. Kolb, "Opacity Peeling for Direct Volume Rendering," *Computer Graphics Forum,* vol. 25, no. 3, pp. 597-606, 2006.
[40] M.M. Malik, T. Möller, and M.E. Gröller, "Feature Peeling," *Proc. Graphics Interface,* pp. 273-280, 2007.
[41] C. Wang, A. Garcia, and H.-W. Shen, "Interactive Level-of-Detail Selection Using Image-Based Quality Metric for Large Volume Visualization," *IEEE Trans. Visualization and Computer Graphics,* vol. 13, no. 1, pp. 122-134, Jan.-Feb. 2007.

Brendan Moloney received the bachelor's degree in computer science from the University of Arizona in 2005 and the master's degree in computer science from Simon Fraser University in 2008. Since 2010, he has been a senior research assistant at the Advanced Imaging Research Center, Oregon Health Sciences University. His research interests include computer graphics, visualization, parallel processing, GPU processing, image processing, and MRI reconstruction.

Marco Ament received the Diplom degree in computer science from the University of Tübingen, Germany, in 2009. Currently, he is a PhD student at VISUS Visualization Research Center at the University of Stuttgart, Germany. His research interests include interactive direct volume rendering on distributed systems for visualization, global illumination, and fluid simulations on GPUs for computer graphics.

Daniel Weiskopf received the Diplom (MSc) and the PhD degrees in physics from Eberhard-Karls-Universität Tübingen, Germany, and the Habilitation degree in computer science from Universität Stuttgart, Germany. From 2005 to 2007, he was an assistant professor of computing science at Simon Fraser University, Canada. Since 2007, he has been a professor of computer science at the Visualization Research Center, Universität Stuttgart (VISUS) and at the Visualization and Interactive Systems Institute (VIS), Universität Stuttgart. His research interests include scientific visualization, GPU methods, real-time computer graphics, mixed realities, ubiquitous visualization, perception-oriented computer graphics, and special and general relativity. He is a member of the ACM SIGGRAPH, the Gesellschaft für Informatik, and the IEEE Computer Society.

Torsten Möller received the Vordiplom (BSc) in mathematical computer science from Humboldt University of Berlin, Germany, and the PhD degree in computer and information science from Ohio State University in 1999. He is an associate professor at the School of Computing Science at Simon Fraser University. He is a senior member of the IEEE and a member of the ACM, the Eurographics, and the Canadian Information Processing Society (CIPS). His research interests include the fields of visualization and computer graphics, especially the mathematical foundations thereof. He is the codirector of the Graphics, Usability and Visualization Lab (GrUVi) and serves on the Board of Advisors for the Centre for Scientific Computing at Simon Fraser University. He is the appointed vice chair for Publications of the IEEE Visualization and Graphics Technical Committee (VGTC). He has served on a number of program committees (including the Eurographics and the IEEE Visualization conferences) and has been papers cochair for IEEE Visualization, EuroVis, Graphics Interface, and the Workshop on Volume Graphics as well as the Visualization track of the 2007 International Symposium on Visual Computing. He has also co-organized the 2004 Workshop on Mathematical Foundations of Scientific Visualization, Computer Graphics, and Massive Data Exploration at the Banff International Research Station, Canada. He is currently serving on the steering committee of the Symposium on Volume Graphics. Further, he is an associate editor for the *IEEE Transactions on Visualization and Computer Graphics* (TVCG) as well as the *Computer Graphics Forum*.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.