

Octree Rasterization: Accelerating High-Quality Out-of-Core GPU Volume Rendering

Baoquan Liu, Gordon J. Clapworthy, Feng Dong, and Edmond C. Prakash

Abstract—We present a novel approach for GPU-based high-quality volume rendering of large out-of-core volume data. By focusing on the locations and costs of ray traversal, we are able to significantly reduce the rendering time over traditional algorithms. We store a volume in an octree (of bricks); in addition, every brick is further split into regular macrocells. Our solutions move the branch-intensive accelerating structure traversal out of the GPU raycasting loop and introduce an efficient empty-space culling method by rasterizing the proxy geometry of a view-dependent cut of the octree nodes. This rasterization pass can capture all of the bricks that the ray penetrates in a per-pixel list. Since the per-pixel list is captured in a front-to-back order, our raycasting pass needs only to cast rays inside the tighter ray segments. As a result, we achieve two levels of empty space skipping: the brick level and the macrocell level. During evaluation and testing, this technique achieved 2 to 4 times faster rendering speed than a current state-of-the-art algorithm across a variety of data sets.

Index Terms—GPU techniques, out-of-core volume rendering, octree, ray casting

1 INTRODUCTION

HIGH-QUALITY volume rendering is an important technique in medical visualization, but there are several barriers to adopting it in the clinic, including the large data size, the quality of the output, the overall performance and the hardware cost. **Although real-time out-of-core volume rendering has been performed on GPUs, performance is usually gained at the cost of image quality.** Custom server level hardware solutions have been developed to provide high-quality rendering, but the cost involved limits their widespread adoption; for example, the distributed memory GPU-accelerated algorithm in [11] is based on multi-GPU clusters at a scale of 256 GPUs. In contrast, **this paper proposes a method to achieve high-quality volume rendering (using tricubic interpolation) of large data sets on commodity graphics hardware.**

In today's clinical environment, one often needs to deal with multiple volume studies, dynamic time-varying data sets, and multiple-user interactivity, which can present severe challenges for real-time rendering.

The key requirement for the clinical use of a visualization system is its support of fast and accurate detection and diagnosis, but currently this is often compromised by the limited computing power available in the clinic. The modest CPU resources available at most hospitals can impose severe restrictions on the algorithmic complexity that can be used in the visualization, which will often sacrifice quality for runtime performance in order to adhere to the clinical time frames.

Our target is to make high-quality visualization more practical in the clinic by the use of a consumer-level GPU; this is a cheaper solution than alternative custom server-level hardware solutions, the cost of which prohibits widespread adoption. While superficially attractive, cloud computing solutions will involve data transference outside the clinical environment and thus create data privacy issues. The use of efficient GPU-based computing resources can support clinical exploration by allowing users to experiment interactively with visualization models and data parameters on a local computer thus avoiding the security and legal problems associated with using external computing resources.

Performance improvement assists the enhancement of individualization, as it makes it feasible to undertake a greater level of data investigation, and thus build a richer picture of the individual patient, by allowing frequent parameter adjustment while still remaining within clinically acceptable time intervals. This will become increasingly important as time-varying volume data becomes more widely available. The proposed GPU solution can use advanced visualization techniques, such as tricubic interpolation, to improve the accuracy of the output whilst executing faster than previous comparable methods.

Modern CPUs greatly benefit from a sophisticated execution pipeline with instruction reordering, branch prediction and other clever optimizations. Hence, branching generally incurs almost no performance penalty. In contrast, a GPU generally has a much simpler control flow logic. Instead, it benefits from having many Arithmetic Logic Units performing arithmetic operations in parallel. So GPUs are efficient in terms of parallel arithmetic operations, but not in terms of branching-intensive serial operations. Most applications consist of a mixture of such serial and parallel tasks, so it can be advantageous to use GPUs to perform heavy arithmetic computation, whilst running complex serial tasks on the CPU, provided that the computation as a whole remains well integrated. If this is the case, the result is a heterogeneous system in which

- The authors are with the Department of Computer Science and Technology, Faculty of Creative Arts, Technologies and Science, University of Bedfordshire, D108, Park Square, Luton, Bedfordshire LU1 3JU, United Kingdom. E-mail: {Baoquan.Liu, Gordon.Clapworthy, Feng.Dong, Edmond.Prakash}@beds.ac.uk.

Manuscript received 23 Aug. 2011; revised 23 Mar. 2012; accepted 22 June 2012; published online 2 July 2012.

Recommended for acceptance by K. Mueller.

For information on obtaining reprints of this article, please send e-mail to: tcvg@computer.org, and reference IEEECS Log Number TVCG-2011-08-0195. Digital Object Identifier no. 10.1109/TVCG.2012.151.

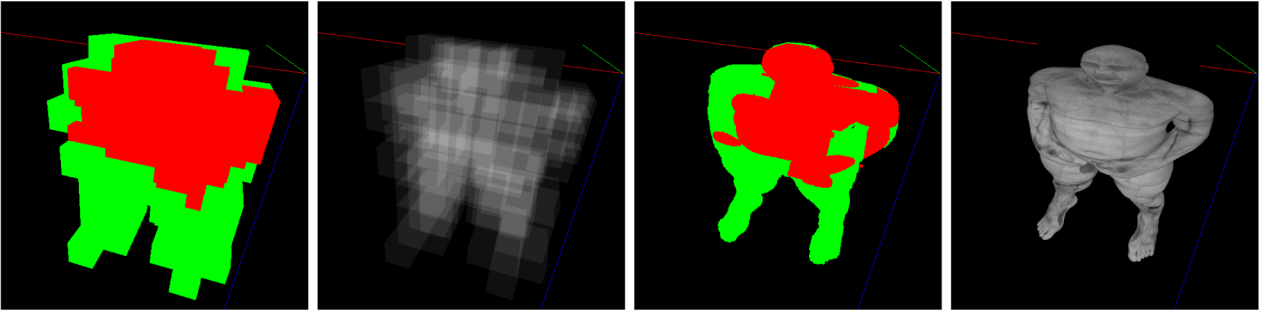



Fig. 1. Octree rasterization of the Visible Human data set (a $2,048^3$ cubical grid). Left: rasterized result of the bricks (in a view-dependent cut of the octree) at different LoDs (red is the finest level, green is the second finest level); each is rasterized as a cube. Second left: visualizing the length of the per-pixel list, that is the number of active bricks captured by each ray. Second right: rasterized result of all the macrocells inside the bricks. Right: final raycasting result using tricubic interpolation, rendered on a GTX480 GPU at 40.4 fps.

the CPU and GPU operate asynchronously, enabling concurrent execution on both.

Volume rendering is typical of such an application, where ray traversal issues are encountered in two stages pertaining to where, when, and how often: 1) the hierarchical data structure, such as octree or kd-tree, is traversed and 2) the voxel data are sampled during ray traversal within the subvolumes of the hierarchy. The traversal of the accelerating structure (octree or kd-tree) is branch intensive, and complex structures (such as stacks) or recursion may be involved, while the raycasting operation itself (trilinear/tricubic interpolation, gradient computation, and color composition) involves heavy, highly parallel computation. This is our motivation for moving the octree traversal from the raycasting loop on the GPU to the CPU, and then using octree rasterization on the GPU to capture a list of ray segments to ease the subsequent GPU raycasting. In this way, we benefit from the asynchronous execution of the CPU and GPU, with each performing only the tasks to which it is most suited.

In the proposed method, a volume is organized and stored in an octree of bricks. To reduce the operational overheads, the depth-level of the octree should be shallow, which produces bricks of large granularity. In addition, to allow empty spaces inside the bricks to be skipped, each brick is further split into regular macrocells; only the nonempty macrocells are rasterized to capture valid ray segments. We take advantage of recent GPU features from OpenGL 4, which allow random access (read/write) and atomic operations into GPU memory, to generate a per-pixel list on the GPU by rasterizing proxy geometries of the active bricks from the octree. Further, we refine the depth ranges of the per-pixel list by rasterizing the active cells inside each brick. In this way, we can capture a per-pixel list of tighter ray segments (one for each brick that a ray penetrates) so the subsequent raycasting pass can be accelerated efficiently by casting rays only inside these valid depth ranges, thus skipping the empty spaces efficiently at the accuracy of the macrocell level. Some intermediate results of the rasterization are shown in Fig. 1.

The main contributions in this paper are as follows:

1. Introduce a new  of-core GPU volume rendering framework, which combines object-order and image-order advantages and acts as a general acceleration technique, which makes more complex

visualizations possible, while maintaining interactivity at the same time. By rasterizing an octree to generate tight ray segments, the method can avoid the traditional need of performing costly per-ray octree traversal inside the GPU raycasting loop (Section 3.2).

2. Achieve two levels of empty-space skipping (brick level and macrocell level) by a unified rasterizing solution for bricks that are at different Levels of Detail (LoDs). A single unified HistoPyramid building procedure is used for all of these bricks at mixed LoDs, which reduces the amount of rendering context switching compared with the naive nonunified method, which involves a great deal of rendering context switching between the interleaved rasterizing and raycasting passes for each brick (Section 4.3).
3. Take advantage of new features in OpenGL 4 to capture a per-pixel list of tight ray segments, one for each brick. Having these ray segments makes the subsequent raycasting pass much easier. Furthermore, in order to avoid the costly memory cycle (between CPU to GPU), we propose a new rendering technique by using another new feature in OpenGL 4: draw-indirect (Section 4.3).
4. Use a single unified pass to cast rays into all the bricks (at mixed LoDs) through which the rays pass, and a simple and accurate method to compute the adaptive sampling rate for each ray, based on Nyquist theory (Section 4.4).
5. Provide an efficient and accurate method for computing the projected area of a brick under perspective projection (Section 4.2.1).

Experiments showed that the proposed algorithm can achieve 2-4 times faster rendering speed than a current state-of-the-art algorithm for large out-of-core data sets, while also producing high-quality rendering using tricubic interpolation.

2 RELATED WORK

2.1 Volume Visualization

In volume visualization, images are created from sampled data defined on a multiple dimensional grid. The two most prominent approaches typically employed are Direct Volume Rendering (DVR) [14], [15], [22] and isosurface visualization [24], [29].

2.2 GPU-Based Volume Ray Casting

GPU-based volume rendering is well established—for recent surveys see [9], [16], [17]. It generally stores the entire volume in a single 3D texture and drives a fragment program casting rays into the volume. All samples along the ray are composited front-to-back, which involves volumetric integration and trilinear/tricubic interpolation.

A volume data set normally has many inactive voxels (such as those containing only air or other transparent regions depending on the transfer function) which occupy a large proportion of the volume but make no contribution to the final rendering. The high computational cost for each sampling step makes it economical to avoid taking samples in these voxels; this is known as empty space skipping. Without it, each ray would have to test all the samples in the full depth range from the near plane to the far plane.

Volume rendering can be accelerated by reducing the length of the ray path as this, in turn, reduces the number of samples computed, which determines, to a considerable extent, the final speed of rendering. Empty space skipping techniques are widely used and they normally fit into one of the following two categories.

Image-order empty space skipping acts within the ray casting loop. It is generally implemented by use of some hierarchical accelerating structure (such as octrees [16] or kd-trees [19], [31]) and encodes only the nonempty space. Then, during the raycasting pass, the current sampling position is used to traverse the hierarchical accelerating structure to sample only inside the nonempty regions or to define a larger step size.

Object-order empty space skipping computes the sampling ranges of the rays per frame in a ray set-up pass (via rasterization of some proxy geometry) before the raycasting pass is executed. It was first introduced by Krüger and Westermann [23] who rasterized a cube proxy-geometry (the data set bounding box) to specify the start and end points for the rays. Hong et al. [18] used a similar cube proxy-geometry for object-order raycasting in which the front and back faces of all of the cubic bricks were projected and rasterized to set up the rays before the raycasting passes. However, these methods require multiple rendering passes for each active brick, so the rendering contexts are switched several times between the rasterizing pass and the raycasting pass for each brick. Further, the multiple bricks were rendered in a serialized manner (that is, rasterize first brick, raycast first brick; rasterize second brick, raycast second brick, ...), exacerbating the context switching between the interleaved rasterizing and raycasting passes and resulting in significant performance loss on GPUs [18].

In addition, previous object-order methods, including some recent work [25] can handle only small data sets, not the large out-of-core data sets on which our method focuses.

2.3 GPU-Based Out-of-Core Volume Rendering

By GPU-based out-of-core, we mean that the volume data is too large to fit into the GPU memory, even if the visible part does. This definition is in the spirit of recent innovative work [1], [6], [12], [13], on the shoulders of which our method stands.

For larger volumes, even if the GPU has sufficient memory, the algorithm would be slowed by the large

number of sampling steps that must be marched per ray. Recent results from out-of-core techniques [1], [6], [12], [13] have extended GPU-based volume rendering potentially to data sets of unlimited size by employing LoD and progressive rendering to achieve interactive performance. In these, the most important parts of the data set are rendered at an appropriate LoD using the fact that, for a given viewpoint, the entire volume does not have to be in memory. By using spatial subdivision to organize the data, empty parts can be left without subdivision and distant parts can be replaced by lower mipmap levels, thus leading to lower memory requirements. As another advantage, mipmapped antialiasing can be achieved as a by-product without extra cost. A multiresolution volume rendering system typically relies on flat [28] or hierarchical [6], [13] bricking schemes. In a hierarchical bricking scheme, the brick size in voxels is kept constant from level to level, and the spatial extent of bricks increases until a single brick covers the entire volume. The flat blocking technique, instead, represents a volume as a fixed grid of blocks and varies the resolution of each block to achieve adaptivity. The disadvantage of this fine-grained approach in comparison with our hierarchical approach is that the number of blocks is constant and the method remains working only if individual blocks are within a small range of sizes.

The efficiency of the hierarchical bricking scheme often suffers from per-pixel traversal of the hierarchical acceleration structure, which is a branch-intensive operation and may not perform well on modern GPU architectures, which have a heavy performance penalty for divergent branching.

To traverse the octree structure along a ray, Crassin et al. [6] used an iterative descent from the tree root, similar to the kd-restart algorithm adapted from the stackless ray traversal method for kd-trees. To avoid a restart from the tree root for each traversal, Gobbetti et al. [13] retained a more complex octree structure, in which pointers to neighboring bricks had to be stored in both CPU and GPU memory.

Due to the traversal costs, all of these methods provided rendering based only on trilinear interpolation, rather than the more computationally demanding tricubic form which tends to produce fewer artifacts.

3 METHOD OVERVIEW

In Section 4, we shall provide details of our method, but we start here with a high-level explanation of where it differs most substantially from previous methods, depicted in Fig. 2.

3.1 Previous Pipeline

Direct volume rendering can be accelerated by reducing the number of samples taken along the rays; often this involves space skipping and adaptive methods. For large out-of-core data sets, it is typically achieved by image-order methods that traverse acceleration structures inside the raycasting loop. Fig. 2(right) depicts the conceptual model of the previous out-of-core GPU DVR pipeline [6], [13].

In this context, large volume data is typically preprocessed by blocking methods, and a multiresolution structure (such as an octree) is built, so that at runtime a

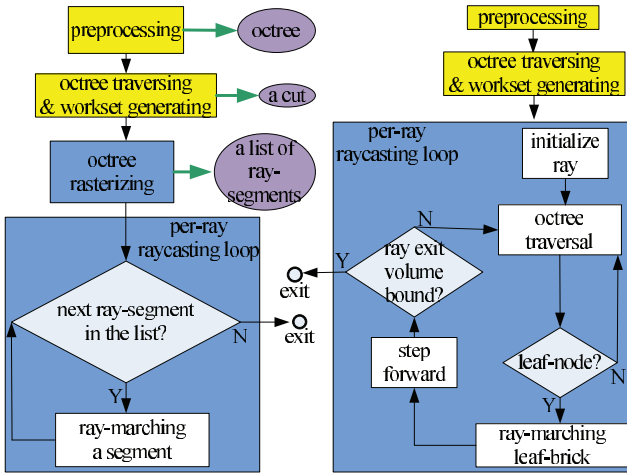


Fig. 2. Flowcharts of our algorithm (left) and previous out-of-core GPU volume rendering algorithms (right). Yellow boxes are executed on the CPU, all other boxes on the GPU. Black arrows indicate control flow; green arrows indicate output of intermediate data.

view and transfer-function-dependent working set of voxels can be adaptively fitted within GPU memory.

The actual per-ray raycasting loop then combines two traversal stages. The first uses the current sampling position of the ray to traverse the octree structure using some stackless ray traversal methods (such as kd-restart traversing). This avoids the need to have a complex stack structure on the GPU, but at the cost of restarting the traversal from the root-node whenever a ray exits from a leaf-node. When a leaf-node is found, the second stage—standard ray marching—is applied, and the brick contents are sampled until the ray leaves the current brick. The next new sampling position then serves as the origin for the octree traversal to find the next leaf-brick.

These methods rely on the ability to traverse an octree structure on the GPU using complex stackless traversal methods, but they suffer from three problems. First, empty spaces inside nonempty bricks cannot be skipped. Second, the octree structure has to be maintained in GPU memory, as well as in CPU memory. Third, the branch-intensive octree traversal has to be performed multiple times per ray (once for each leaf-brick encountered along the ray) on the GPU inside the raycasting loop, which may degrade the performance. As the GPU does not excel at such branch-intensive operations, the cost of traversing the structure often outweighs the gains from computing fewer samples, as pointed out by Knoll et al. [22], who also showed that while GPU approaches are faster for smaller data, when rendering large volumes, CPU-based Bounding Volume Hierarchy (BVH) traversal is significantly faster than previous GPU approaches.

3.2 The New Pipeline

The most significant difference from previous GPU-based out-of-core DVR algorithms is that we have moved the octree traversal out of the raycasting loop, performing it now on the CPU rather than the GPU. Our method does not maintain the octree structure in GPU memory; instead, we generate a list of tight ray-segments in GPU memory by rasterizing. The flowchart is depicted in Fig. 2(left).

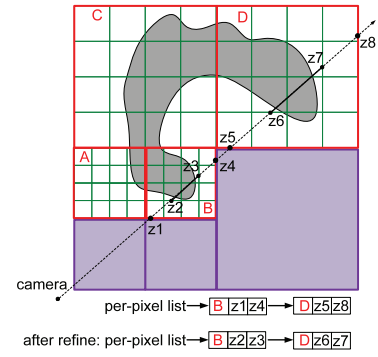


Fig. 3. Algorithm philosophy. The octree-cut is composed of all of the active bricks (red—A, B, C, and D) at different LoDs depending on the current view. Each active brick is subdivided into macrocells (green). Empty bricks (purple) are never added to the octree-cut. We first rasterize active blocks so that the fragment shader can capture all of the active blocks that the ray penetrates (B and D) into a list in front-to-back order. Then we rasterize nonempty macrocells and refine the z-values in the list in order to get a tighter ray segment (shown as the solid black line) for each brick at the accuracy of the macrocell level.

During preprocessing, we build a sparse octree structure from the original large data set and store it on the hard disk. At runtime, for each frame, our approach has three stages: workset generation, octree rasterization, and raycasting.

It uses the CPU to traverse the octree structure and employs an adaptive loader to update a view- and transfer-function-dependent working set of bricks (a cut of the octree), which is incrementally maintained in the CPU and GPU memories by asynchronously fetching data from the out-of-core octree. Further, we use the new OpenGL features to rapidly rasterize the octree nodes and capture a list of tight ray segments (in front-to-back order), one for each brick penetrated by the ray, so that the subsequent ray casting pass can know the depth range inside each brick. Since this range is further refined by rasterizing all of the active macrocells inside a brick, this depth range is at the accuracy of the macrocell level. Fig. 3 shows how our algorithm works.

The experimental results show that this strategy really benefits the overall rendering performance over previous GPU methods, see Table 1 for details.

4 THE ALGORITHM IN DETAIL

This section provides details of the steps of the algorithm.

4.1 Preprocessing

As in previous GPU-based hierarchical bricking schemes, we chose a sparse octree structure, which is well adapted to storing regular data such as voxels—“sparse octree” means that empty bricks (i.e., those with only zero voxels) are excluded from the tree structure, so some nodes of the octree may have fewer than eight children.

The original volume data is decomposed into small cubical bricks with a predefined constant voxel resolution B_{res}^3 . These bricks are the leaf nodes of the octree—the finest LoD representation of the volume. Coarser bricks are built as the inner nodes of the hierarchy by downsampling eight neighboring bricks at the finer level using an averaging filter. This also effects mipmapped antialiasing

TABLE 1
Performance Comparison for Different Data Sets Using
Our Method and *PerRayTraversal*, Which Is Our
Implementation of [13]

datasets	data resolution	octree depth	PerRay Traversal	ours	ratio
<i>Toutatix</i>	$2048^2 \times 1861$	5	4.5	9.8	2.18
<i>Abdomen</i>	$2048^2 \times 1534$	5	4.9	12.0	2.45
<i>Obelix</i>	$1024^2 \times 4096$	6	3.9	8.5	2.18
<i>Amnesix</i>	$1024^2 \times 3248$	6	5.4	14.2	2.63
<i>Maco</i>	$1024^2 \times 1924$	5	2.7	9.5	3.52
<i>heart</i>	2048^3	5	3.2	10.1	3.16
<i>Tragicomix</i>	$1536^2 \times 1818$	5	2.8	6.7	2.39
<i>Melanix1</i>	$1024^2 \times 3696$	6	1.3	5.5	4.23
<i>Melanix2</i>	$1024^2 \times 3696$	6	2.1	5.6	2.67

Speed-up ratios of our method over *PerRayTraversal* are shown in the last column. All the timings are measured using average framerates (frames per second) for the images shown in Fig. 10 with a screen resolution of 512^2 . Tricubic texture interpolation is used throughout and the cubic gradient computation and Phong lighting are enabled for the last four data sets.

[3] as the mipmapped parent bricks are already effectively low-pass filtered and can remove high frequencies that may lead to artifacts in texture minification. This can also take some of the antialiasing burden off the real-time renderer. Hence, a large volume data set is represented as a multiresolution hierarchy (a sparse octree) maintained out of core. Each tree node contains a pointer to a brick with a constant resolution that approximates the part of the volume corresponding to the octree's node.

Bricks are made self-contained, as in [13], by storing extra overlapping neighboring voxels at each border in order to support runtime operations that require access to adjoining bricks, such as tricubic interpolation and gradient computations.

An obvious advantage of having a constant resolution for all of the bricks at different levels is that the bricks all occupy the same amount of memory space, so it is much easier to pack the active bricks (at mixed LoDs) tightly into a brick-pool in GPU memory as a single 3D texture. It allows for arbitrary placement of bricks to replace any unused bricks in memory without wasting any space. This facilitates the update mechanisms of the bricks (from CPU memory to GPU memory) that are needed to produce every new output image.

Note that the octree is independent of transfer function so, to assist transfer-function-based runtime brick culling, for each octree node we also store the min-max scalar values of the corresponding brick, so that transparent bricks can be culled efficiently at runtime by using a summed-area table of the transfer function [16].

Since the nonempty bricks may still contain many empty spaces, for more efficient culling we decompose each brick further into macrocells, which are smaller cubical subvolumes, each containing M_{res}^3 voxels. So, for each brick at different LoDs, along with the voxel data we also store macrocell 3D texture data, in which each macrocell stores only the min-max scalar values of the corresponding subvolume. This will be used to cull empty macrocells at runtime, based on the transfer function.

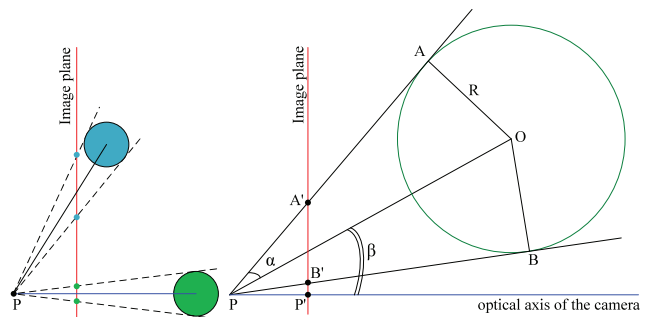


Fig. 4. Left: two spheres with same radius and at the same distance from the camera can have different projected sizes on the image plane (red), due to their different subtended angles (β) from the camera axis (blue). Right: under perspective projection, the projected size of the sphere (center O , radius $R = |OA| = |OB|$) on the image plane is $|A'B'|$. P is the camera position. $\alpha = \angle OPA = \angle BPO$.

4.2 Generating a View-Dependent Working Set

At each frame, we choose a **cut** (that is, a set of nonempty tree nodes) of the out-of-core octree and use it to update the GPU brick-pool for rendering. The cut is chosen to represent the whole volume at different resolution levels depending on the current viewer position and transfer function.

The basic principle is to ensure that, in a brick, the projected size of a voxel is never smaller than one pixel. At each frame, as in [13], we update the cut (initialized as the root node) by refining each node that is considered too coarse, replacing it by its nonempty children, or by collapsing a node that is considered too fine (possibly due to it being too far from the camera), replacing it by its parent (that is, a low-pass filtered version). This procedure stops when all nodes in the cut are considered adequately refined, or no more space is available in the GPU brick-pool to contain a further subdivision. As our brick granularity is large, the octree is very shallow, allowing for fast updating.

A summed-area table of the transfer function is used to determine if a node is transparent—transparent nodes and their children are excluded from being refined into the cut. View frustum culling is implemented using the six bounding planes of the frustum to cull bricks that are outside the frustum; these bricks and their children are not included in the tree-cut.

To generate the working set, we implement a breadth-first octree traversal on the CPU so that we can maintain the cut from frame to frame. We traverse the octree from the root and sort all the nodes in the cut to enable us to refine the bricks in decreasing order of their projected size.

In the following sections, we give more details of the algorithm.

4.2.1 Computing the Projected Size of a Brick

Previous methods for computing the projected size of a brick are based only on the distance to the viewpoint [7]. This is not sufficiently accurate since, under perspective projection, the subtended angle can make a significant difference to the projected size—Fig. 4(left).

We propose a more accurate method for computing the projected size under perspective projection, in which the bounding sphere of a brick is used as a proxy. As shown

in Fig. 4(right), the projected size of the sphere on the image plane is $|A'B'|$, which can be computed as

$$\begin{aligned}\overrightarrow{P'A'} &= c * \tan(\alpha + \beta), \\ \overrightarrow{B'P'} &= c * \tan(\alpha - \beta), \\ |\overrightarrow{B'A'}| &= |\overrightarrow{P'A'} + \overrightarrow{B'P'}| = c * |(\tan(\alpha + \beta) + \tan(\alpha - \beta))|,\end{aligned}\quad (1)$$

where c is a coefficient related to the camera configuration (screen size and field-of-view angle) that scales the length of $A'B'$ to the screen space in pixels. We use the following basic trigonometric formulae to calculate it.

$$\begin{aligned}\tan(\alpha + \beta) &= (SC + CS)/(CC - SS), \\ \tan(\alpha - \beta) &= (SC - CS)/(CC + SS), \\ SC &= \sin(\alpha) * \cos(\beta), \quad CS = \cos(\alpha) * \sin(\beta), \\ CC &= \cos(\alpha) * \cos(\beta), \quad SS = \sin(\alpha) * \sin(\beta), \\ \sin(\alpha) &= R/|\overrightarrow{PO}|, \\ \cos(\beta) &= \text{DotProduct}(\overrightarrow{Axis_{cam}}, \text{Normalize}(\overrightarrow{PO})),\end{aligned}\quad (2)$$

where $\overrightarrow{Axis_{cam}}$ is the optical axis of the camera.

Note that in Fig. 4(right), $\alpha < \beta$, but the same formulae apply also when $\alpha \geq \beta$, that is, when the sphere intersects the optical axis.

After the projected size (in pixels) of a brick is obtained, we stop refining the node and label it as $node_{cut}$ if $|A'B'| < B_{res}$, which means that, within the brick, the projected size of a voxel is already smaller than a pixel, so it is considered adequately refined with respect to the volume data discretization. Otherwise, further refinement is carried out.

4.2.2 View-Dependent Sorting of the Bricks

As the nodes in the cut must be rasterized in a front-to-back order at the next stage, we now sort them and store the result in a pointer list for later use. While, for orthogonal projections, the order is same at all tree levels, this is not the case for perspective projections. We compute the order of all nodes in the cut using one pass of a CPU-based depth-first octree traversal.

An STL list stores the pointers of the traversed cut-nodes. During the depth-first traversal, a traversed node is always replaced by its children in a front-to-back order, until a node labeled as $node_{cut}$ is found. This is implemented by pre-computing an 8×8 lookup table, each row of which encodes a possible order of the eight octants in octree space according to the viewpoint. This order is precomputed, based on the octant of the current octree subdivision in which the viewpoint is located (relative to the center of the parent node), and this octant is used as a bitmask to look up the table at runtime. An example is shown in 2D in Fig. 5. The eight rows of the table encode eight orders for eight possible locations of the viewpoint. At runtime, only a look-up of the table is needed to find the order of the eight children, using the position bitmask of the viewpoint as the entry.

4.2.3 Memory Management of the Working Set

After the refinement process, the nodes in the cut define the current working set, which is considered adequate for the

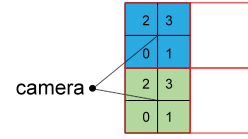


Fig. 5. For the blue and green subdivisions shown, the directions to the camera are within octants 0 and 2, respectively, (used as bitmask), so the precomputed order is 0-1-2-3 and 2-0-3-1, respectively.

multiresolution representation of the volume at the current frame. The working set now contains the corresponding brick data and macrocell data of these nodes in the cut.

They are incrementally transferred into GPU memory by asynchronously fetching data from the out-of-core octree; in effect, the CPU memory acts as a secondary level cache from which the GPU cache is updated.

The working set is uploaded into GPU as two memory pools: a brick-pool and a macrocell pool. These are managed as two 3D textures, which are organized into slots of a fixed size. Both pools contain $P_x \times P_y \times P_z$ slots (each corresponds to an octree-node in the cut). In the brick-pool, each slot stores exactly one brick of volume data. Each slot-index acts as a unique brick-ID, B_{ID} , for the node that occupies the corresponding slot. The macrocell 3D texture data of each brick in the cut is packed in the macrocell pool, at the same slot position as B_{ID} .

At runtime, our algorithm makes sure that all nodes in the cut for the current frame are in the GPU cache. This enables rendering of the volume to take place at an appropriate resolution relative to the current viewpoint. When the viewpoint is moved or the transfer function is changed, nodes in the cut are fused or collapsed, based on the resolution required. Each node in the GPU cache has a time-stamp that is reset to zero upon use and increased by one after each frame. If an octree node has to be subdivided, new bricks are transferred to the GPU cache at the slot positions that were previously occupied by the oldest (thus unused) bricks. This strategy is referred to as Last Recently Used (LRU) [13].

4.3 Proxy-Geometry Rasterization

After finding the cut of the octree, we need to raycast the bricks corresponding to it. A straightforward solution would be to raycast the bricks one by one [18]; in this way, each brick would require a few rasterizing passes to set up the ray segments (start/end positions for each ray) and a raycasting pass for rendering. However, rendering the bricks in this way would result in many interleaved rasterizing and raycasting passes, producing a great deal of rendering context switching between the rasterizing and raycasting passes for each brick, which is inefficient for large data.

We propose a more efficient solution. As shown in Fig. 6, we first rasterize the proxy geometries of the whole working set of bricks (at different LoDs) in a unified rendering pass without any switching of the rendering context, and capture all of the bricks that a ray penetrates into a per-pixel list by using the new features of OpenGL 4. In a second rasterizing pass, we rasterize all of the nonempty macrocells (of all the bricks) and refine the per-pixel list, in which each element will contain a ray-segment corresponding to the brick that the ray penetrates. As a

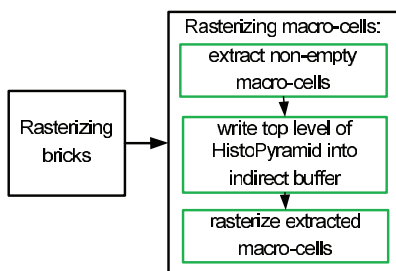


Fig. 6. Flowchart of proxy-geometry rasterization, which is composed of two steps, shown as the black boxes; the second step is composed of three substeps, shown as the green boxes.

result, we can skip empty space at a macrocell level and provide, for the subsequent raycasting pass, much tighter ray segments than would be obtained simply from the bounding box of the brick.

In the following sections, we give more details of these two rasterizing passes.

4.3.1 Rasterizing Bricks of Mixed LoD

To render an image with single-pass GPU raycasting, the fragment shader must be able to enumerate, in a front-to-back order with respect to the camera position, all of the bricks pierced by the associated view ray, so this rasterizing pass should capture all the bricks through which the ray passes.

This order is already stored in a list—see Section 4.2.2. Each brick is rasterized as a unit cube (six quad rendering primitives) from an OpenGL Vertex Buffer Object (VBO). In the vertex shader, the unit cube is scaled and translated to its proper size and position according to the actual spatial extent and position of the brick. Since there are, at most, hundreds of bricks in the cut, this rasterizing pass is quite fast.

In the past, pixel shaders have lacked scatter capability, making it difficult to construct complex data structures (such as a per-pixel linked list) on the GPU. Recently, software-based rasterizers using CUDA [27], [20] have been proposed to implement multilayer depth peeling in a single pass. However, this leaves the hardware rasterizing unit idling while extensive use is being made of the programmable cores. These rasterizers also need the rasterized fragments to be sorted explicitly, which could create a bottleneck when the depth complexity of the geometry is high. However, recent APIs (e.g., DirectX 11 and OpenGL 4) and hardware features include the ability to perform random access (read/write) and atomic operations at arbitrary memory locations, such as the new OpenGL four functions [2]: *imageAtomicIncWrap()*, *imageAtomicAdd()*, *imageAtomicMax()*, *imageStore()*, etc.

We describe a method that uses these new features to construct dynamically on the GPU a per-pixel list for the bricks (at different LoDs) that the ray penetrates. The method is fast enough for real-time rendering and can capture multiple ray-segments per pixel in a single pass.

In the fragment shader, we use these new functions to implement a per-pixel list for each ray (similar to [32], which used DirectX 11). By using *imageAtomicIncWrap()*, we can capture all the rasterized fragments and store them in the

per-pixel list. Each node of the list stores three elements (B_{ID} , min_z , and max_z), which are, respectively, a brick-ID, and the min and max z values of the rasterizing result for one brick. If the ray pierces N bricks, the per-pixel list will have N nodes, and since we rasterize the bricks in front-to-back order, the N nodes will also be in a front-to-back order and so will not need per-pixel sorting on the GPU. Note that, since GPU shaders do not execute in a deterministic order, to guarantee the correct front-to-back order of the bricks captured in the list, we call a memory barrier function (*glMemoryBarrierEXT*) before each brick's draw-call API, to ensure that all GPU memory writing is completed before we start to render the next brick.

We preset the length of the per-pixel list to a preallocated budget. In all of our experiments, we found that 16 is big enough for the length of the per-pixel list. If this budget expires, further bricks beyond the frontmost 16 bricks will be skipped. In careful investigations, we did not find any difference between our rendering results and the ground truth.

4.3.2 Rasterizing Macrocells

We need to refine the ray segments in the per-pixel list in order to make the subsequent raycasting more efficient by skipping empty spaces inside the nonempty bricks. To achieve this, in this pass we rasterize each nonempty macrocell of the bricks in the cut as a proxy sphere [25]. Using a sphere as the proxy geometry involves only one vertex operation, as opposed to the many operations associated with a cube (i.e., 12 triangle primitives), as used in previous proxy geometry techniques [23], [18].

Extracting nonempty macrocells. Each brick has $(B_{res}/M_{res})^3$ macrocells, but we can expect that only a small proportion of them are nonempty, though this is, of course, dependent upon the transfer function. Thus, for efficiency, we first extract, on the GPU, only the nonempty macrocells for all of the bricks in the working set.

For this, we employ an efficient GPU stream compaction algorithm, HistoPyramid [33], which is a special case of the prefix-sum scan algorithm, highly optimized to exploit the texture locality of GPUs' texture processing units. But, instead of building a HistoPyramid texture for each brick, we propose a unified solution that builds only one HistoPyramid texture for all the bricks. By this means, only a single rendering pass is needed to rasterize all of the macrocells in all of the bricks. This eliminates rendering context switching and thus greatly improves the overall rendering efficiency.

Since the nodes in the cut are at different tree levels, we propose a customized pyramid building and traversal method. To build a unified pyramid texture for all bricks in the working set, the input is the macrocell pool containing the packed macrocell structure for all of the bricks in the cut, which we uploaded into the GPU memory (see Section 4.2.3). The slots of the macrocell structure contain the min-max values of each macrocell inside a brick. The summed-area table of the transfer function is used to determine whether a macrocell is transparent or not, and the result (0 or 1) is stored into the basis texture of the pyramid in a full screen GPGPU pass. The HistoPyramid algorithm then builds the upper level textures of the

pyramid in a bottom-up, layer-by-layer manner. Note that we do not need to build the pyramid to the exact top level containing only one texel; instead, we stop building at the pyramid level, L_t , at which each texel exactly corresponds to a brick in the working set, and the texel-value represents the total number of active macrocells that the corresponding brick can have.

Using draw-indirect to avoid memory cycles. Previous HistoPyramid approaches [33], [8], [26] have needed to read the top texture level (L_t) of the HistoPyramid texture that was built on the GPU back to CPU memory, so that the CPU knows the values of the parameters for the corresponding OpenGL draw-calls, before the draw-calls can be submitted from the CPU to the GPU. To avoid this costly memory round-trip, we employ another new feature in OpenGL 4: draw-indirect (ARB_draw_indirect), which provides a mechanism to source the parameters of draw commands from within a buffer object stored in high-performance OpenGL server memory, instead of from CPU client memory. This is particularly useful for applications in which the parameter values for the subsequent OpenGL draw commands are generated on the GPU, rather than on the CPU, hence it allows the GPU to consume these parameters without a round-trip to the CPU or the expensive synchronization that would be involved.

To take advantage of this feature, we run a full-screen GPGPU pass to write an index buffer object (in the fragment shader) by reading the values stored in the top level (L_t) of the pyramid texture on the GPU. As a result, the CPU does not need to know how many nonempty macrocells each brick can have, since all the parameters for the instanced draw commands (*DrawArraysIndirect*) are already stored on the server memory.

Rasterizing extracted macrocells. In this pass, all of the extracted active cells (belonging to the bricks in the cut) are rendered by submitting one instanced draw command (*DrawArraysIndirect*) for each node in the cut. No input parameters from the CPU are needed for the draw-calls. Each draw command rasterizes all of the nonempty macrocells of the corresponding brick by retrieving parameters from the index buffer object.

In the vertex shader, the brick-ID is used to access the corresponding texel at the top level (L_t) of the pyramid texture, and *gl_VertexID* is used as a key for HistoPyramid traversal to acquire the relative position of each macrocell inside the brick. The number of point primitives to be rendered for the brick is fetched from the index buffer object, so no CPU-GPU synchronization is needed. When we traverse to the bottom level of the pyramid, the position (inside the brick) of the corresponding macrocell can be retrieved. Note that the bricks in the cut are at different tree levels, so their corresponding macrocells have differing spatial extents, so we scale and translate this macrocell to its target size and position in object space, to enable it to be rasterized as a single rendering primitive (*GL_POINT*), that is, a disk with a properly computed radius and center in screen-space. The process for finding the position and size of the disk is described in detail in [26]; it uses simple high-school trigonometry. We compute the disk in this vertex shader, which can guarantee to cover all of the pixels on to which the bounding sphere of the macrocell projects.

In the fragment shader, the rasterizing results of a macrocell will be captured into the per-pixel list (created in Section 4.3.1) using a bucket sort in which each node of the list acts as a bucket corresponding to a brick B_{ID} . We use the ID of the brick to which the macrocell belongs as a key to compare with each node of the per-pixel list. When a match with a node's element B_{ID} is found, we employ two GPU atomic functions, *imageAtomicMin()* and *imageAtomicMax()*, and use the rasterized z value as argument to update the node's other two elements— min_z and max_z . After all of the nonempty macrocells inside a brick have been rasterized, min_z and max_z will form a tight valid ray-segment for the brick; these ray-segments will be in a front-to-back order in this list, as shown in Fig. 3. These ray-segments are based on the finer proxy geometry of the macrocell, rather than the brick as a whole, and the interval must be tighter than, or at the very worst equal to, the interval defined by the bounding box of the brick.

In this way, we need only one rasterizing pass of the cells to produce a list of ray-segments, one for each of the bricks in the cut. This is a great improvement over traditional depth-peeling [10] which peels only one depth layer per rasterizing pass. Further, instead of needing a separate pass for each brick, all of the macrocells in all of the bricks are now rasterized together in a single pass, despite the fact that the bricks have different spatial extents.

Kainz et al. [20] implemented a skipping strategy for small in-core data by rasterizing a precomputed bounding geometry (a triangular mesh) using a software rasterizer via CUDA. Their customized rasterizer performs per-pixel depth sorting to produce a series of depth intervals, whereas our method does not need the per-pixel sorting. Further, precomputing the triangular geometry for a large volume can be very slow, so it is not easy for their method to dynamically change the transfer function at runtime for a large data set.

4.4 Raycasting

After the per-pixel list is generated by the rasterization, the raycasting pass is straightforward: the two end points of a ray-segment define a depth interval to perform the GPU raycasting for the corresponding brick.

We simply drive a fragment program that traverses the per-pixel list. For each node (corresponding to a brick with ID B_{ID}) of the list, we cast rays into the volume data of the brick only within the tight ray segment defined by min_z and max_z . As mentioned previously, this allows all bricks in the list to be traversed in a front-to-back order by a single raycasting pass and, at the same time, empty space skipping to be performed at the macrocell level.

For each ray-marching step, we transform the sampling position into texture space and use B_{ID} to locate the slot of the brick-texture inside the brick-pool. We then sample the volume texture using tricubic interpolation [30], apply the transfer function after opacity adjustment [9] according to the varying sampling distance, compute the tricubic gradient and Phong lighting, and finally perform the color composition in a front-to-back order.

Thanks to our CPU sorting of the bricks, this happens only once for each frame—all of the nodes in the per-pixel

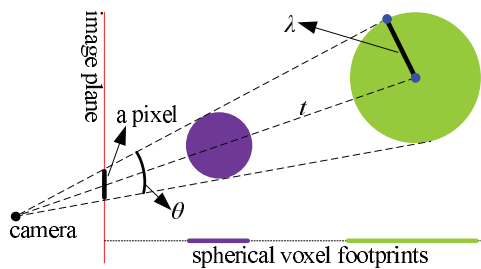


Fig. 7. The per-sample adaptive sampling step-size λ is computed from the distance t to camera and the solid angle θ subtended by each individual pixel.

list are naturally in a near-to-far order which obviates the need for explicit per-pixel sorting in a fragment shader, which could be time-consuming.

Computing the adaptive sampling rate. To achieve high rendering quality, our ray marching uses an adaptive sampling strategy based on Nyquist theory.

Uniform sampling ignores an important component of the convolved volume rendering integral and its resulting Nyquist limit. Thus, it undersamples features close to the viewpoint relative to those further away. To remedy this, Knoll et al. [21] employ a sampling strategy that uses the ray distance itself as a sampling metric.

Here, we propose a more intuitive solution, which is based not only on the ray distance, but also on the solid angles subtended by individual pixels. Specifically, it improves on [21] in that it accounts for the fact that projections occur differently for different pixels in the image (i.e., they have different sized solid angles).

The part of the scene visible at a pixel is not, in fact, most appropriately represented by what is present along a single ray line (which has no thickness), but rather what is present within a cone defined by the viewpoint and the pixel's subtended solid angle θ , as depicted in Fig. 7 [4]. This cone may enclose not just a single voxel, but large groups of voxels, which form a proxy spherical voxel at the truncated position (shown as the green and purple spheres in Fig. 7) depending upon the distance from the camera; when the camera is far away, the truncated diameter of the cone can become quite large.

To avoid aliasing when simulating this behavior with a single very thin sampling ray line, one should apply mipmapping and a properly computed sampling rate (based on Nyquist frequency). Since a suitable mipmapping level for the current brick has already been identified in Section 4.2.1, we now calculate an appropriate adaptive sampling rate using a very simple formula.

Referring to the pixel-cone, a proper sampling step-size should be equal to the radius of the spherical voxel at the current sampling location in order to match the Nyquist frequency, which requires two samples inside a spherical voxel of the truncated cone at the current sampling location. Since the solid angles, θ , of different pixels differ under perspective projection, to simplify the runtime computation, we precompute the various values of $\sin(\theta/2)$ for each pixel, which are related only to the camera's field-of-view angle and the screen resolution, on the CPU, and store them in a texture. In the raycasting fragment shader, at runtime,

we can then compute the adaptive sampling step-size λ using a simple formula involving only one multiplication

$$\lambda = t * \sin(\theta/2), \quad (3)$$

where t is the distance of the current sampling point to the camera, and $\sin(\theta/2)$ is retrieved from the texture. As a result, the adaptive sampling step-size is equal to the radius of the spherical voxel of the truncated cone, which corresponds exactly to the Nyquist sampling frequency. This per-sample method is a simple and accurate way to compute the adaptive sampling rate at any sampling position along the ray.

5 RESULTS

The system was implemented using OpenGL/GLSL 4 on a desktop PC (Intel 2.27 GHz CPU with 8 GB RAM), with an NVIDIA GeForce GTX 480 GPU having 1.5 GB video RAM.

We tested our system on a variety of high resolution models. The first is the Visible Human male data set (*Male*). This has an original resolution of $1,760 \times 1,024 \times 1,878$, and it was embedded in a $2,048^3$ cubical grid to construct the octree. The second is a CT scan of a female (*Melanix*) with an original resolution of $1,024 \times 1,024 \times 3,696$, which was embedded in a $4,096^3$ cubical grid. The third model is the Richtmyer Meshkov instability CFD simulation data (*RM*); this has an original resolution of $2,048 \times 2,048 \times 1,920$ and it was embedded in a $2,048^3$ cubical grid. More large medical data sets are shown in Fig. 10 and Table 1. In each cases, an octree was generated at the preprocessing stage from the cubical grid. This offline data preprocessing takes less than 20 minutes on the CPU for each of the data sets tested.

5.1 Tricubic versus Trilinear Interpolation

Due to our effective octree rasterization, the captured ray-segments are already very tight (at the accuracy of the macrocell level), so we can employ tricubic texture interpolation inside the raycasting loop at a very low additional cost. This provides superior image quality compared to the conventional trilinear interpolation, as can be seen in Fig. 8. Note that, in this paper tricubic texture interpolation was used in all images produced by our method, apart from Fig. 8 (left).

5.2 Comparison of Rendering Quality

ImageVis3D (*iv3d*) [1], [12] is a GPU-based out-of-core LoD renderer recently introduced for large-scale volume rendering.

Since its source code is publicly available (for which we express our appreciation to the authors), it is possible for us to make a side-by-side comparison of rendering quality by using the same data sets, transfer functions, viewing configurations and hardware platform.

Without changing any algorithmic part of its code, the progressive rendering of *iv3d* was disabled to render at its highest quality, in order to compare with our rendering results shown in Fig. 9.

From this, we can see that our method produces much smoother rendering (especially at the areas of the head and feet for *male*, silhouettes and teeth for *female*, and the base for the *RM* data), thanks to our tricubic interpolation and

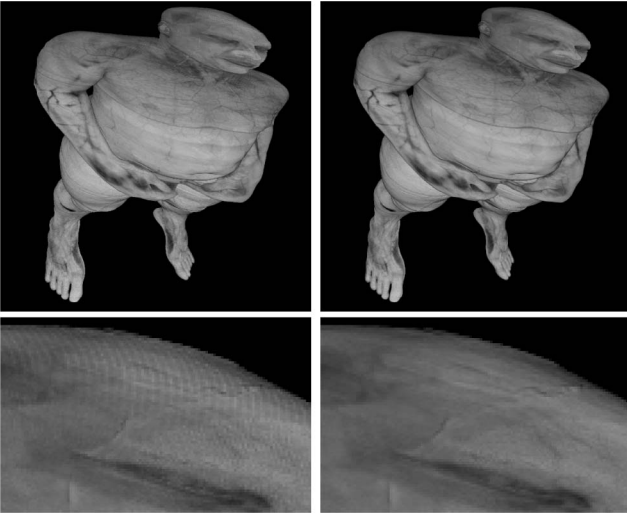


Fig. 8. The images were produced using the same adaptive sampling strategy and identical rendering configurations; the only difference is the interpolation method used: trilinear (left) and tricubic (right); the respective close-up views are shown in the bottom row. The images were rendered at 1024^2 screen resolution with framerates of 13.2 fps and 10.6 fps, respectively.

view-dependent adaptive sampling, which takes more samples at positions close to the camera.

Since *iv3d* renders the data set at full resolution, while ours renders an octree cut of it with adaptive resolution, we will not compare rendering speed with it here. Instead, we compare rendering speed in the following section with a GPU octree traversing algorithm.

5.3 Comparing the Rendering Speed with a GPU Octree Traversing Algorithm

The algorithms most closely related to ours are [6] and [13], both of which use stackless per-ray octree traversing on GPU, which is different from our octree rasterization. As the source codes for these algorithms are not publicly available, we were not able to make a direct side-by-side comparison with them on the same hardware. So, in order to evaluate the performance benefit of our octree rasterization over the previous per-ray octree traversal method, we implemented the per-ray octree traversing algorithm on the GPU according to the pseudo code from [13]—this implementation is referred to as *PerRayTraversal*. For the comparison, we chose [13] rather than [6] because it is targeted at medical applications like ours, whereas [6] is targeted mainly at gaming.

Please note that in the following comparisons, both our algorithm and *PerRayTraversal* used the same workset generation and leaf-brick ray-marching algorithm. The only difference lies in how the leaf bricks are produced for the fragment shader to raycast them—our method uses octree rasterization to obtain the per-pixel list of bricks, whereas *PerRayTraversal* uses per-ray octree traversal on the GPU to produce the leaf bricks. In both cases, these bricks are used by the fragment shader to perform ray marching, so both methods produce exactly the same rendering results, as shown in Fig. 10, where multiple medical data sets are rendered with very close views. The rendering performance statistics (including the data resolution of each data set,

frame rates of both methods, and the speed-up ratios) of these images are shown in Table 1.

This experiment shows that the branch-intensive octree traversal method on the GPU is 2 to 4 times slower than our octree rasterization method, since all other parts of the code are exactly the same for the two methods tested. Both methods used exactly the same working set, cubic sampling scheme, sampling rate, transfer function and lighting computation. The only difference from *PerRayTraversal* is that we have moved the octree traversal out of the raycasting loop, so there is no longer a need to maintain the octree structure in GPU memory; instead of traversing the octree on the GPU, we use the new OpenGL features to rapidly rasterize the octree nodes and capture a list of ray segments (in front-to-back order), so that the subsequent ray marching pass can know the brick-list and the depth ranges to perform raycasting. The difference between the two flowcharts is depicted in Fig. 2.

5.4 Performance Analysis by Comparison with a Naive Object Order Algorithm

To gain further insight into the runtime performance of the proposed technique, we compare it with a naive object order algorithm, which rasterizes and renders the active bricks one by one (instead of using the proposed unified rasterizing solution). This means that, for each brick, it first rasterizes a cube proxy geometry to find the depth range (start and end depths) of the brick using dual-depth-peeling [5], and then runs a single ray-marching pass to render this brick. After that, the two depth buffers can be reused for depth-peeling the next brick. To render all of the bricks in this way leads to many interleaved rasterizing and ray-marching passes, producing a great deal of rendering context switching.

Our experiments show that the proposed technique is around 20 times faster than this naive implementation (using the Visible Human data set rendered in Fig. 1). Except for the proxy geometry rasterizing strategy, all other parts (including the working set generation and ray-marching algorithm) had exactly the same form in both cases. This means that the proposed unified proxy-geometry rasterization solution is the key factor contributing towards the performance boost.

We profile the percentage of time spent on the individual stages of the proposed method at runtime (using the Visible Human data set). We found that the cost of the first stage (building the view-dependent working set and transferring it to GPU memory) accounts for 91 percent of the total rendering time for a frame, while the other two stages (rasterizing and ray-marching) account for only 7 and 2 percent, respectively. In contrast, for the naive implementation, the same first stage accounts for only 30 percent of the total rendering time, while the rasterizing and ray-marching stages (which are interleaved in this method) account for 70 percent of the total rendering time. This means that our algorithm has dramatically changed the timing percentages for the different pipeline stages by having attacked the rendering bottleneck (having reduced it from 70 to 9 percent). The ray-marching is so fast because our algorithm produces a list of very tight ray segments and avoids the time-consuming per-ray octree traversal on the GPU.

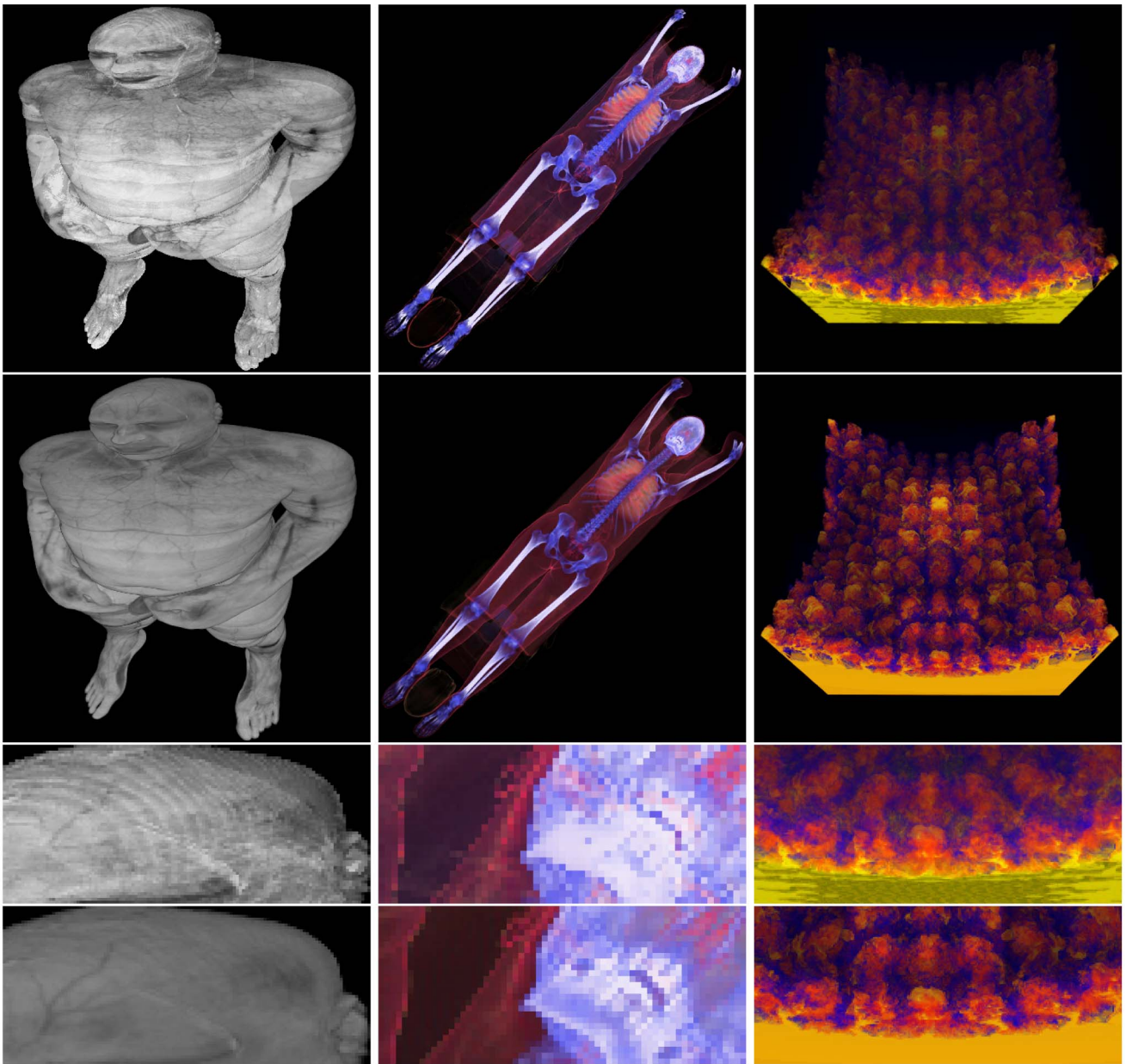


Fig. 9. Quality comparison of iv3d (top row) with our algorithm (second row); side-by-side comparisons of the details are shown in the respective close-up views below (with iv3d in the third row and ours in the bottom row). Our smoother rendering is achieved by using tricubic interpolation and adaptive sampling, which takes more samples near the viewpoint, while iv3d uses the trilinear interpolation and uniform sampling. These images (from left to right: *Male*, *Melanix*, and *RM*) are rendered by our algorithm at frame rates of 26.2, 34.4, and 15.8, respectively, with screen resolution of 512^2 .

The same first stage is required in all out-of-core GPU methods, so the proposed method has concentrated on accelerating the GPU raycasting procedure by employing highly efficient octree rasterization. As a result, there has been a dramatic change in the timing percentages for the different pipeline stages. This makes it feasible to apply more complex raycasting algorithms, such as the use of higher sampling rates or higher-order interpolation, at relatively little additional cost.

Doubling the screen resolution from 512^2 to $1,024^2$ roughly halves the frame rate, even though four times as many pixels have to be rendered (an example is shown in Fig. 8). This is not only because the runtime rendering speed of our system is influenced mainly by the working-set generation stage, rather than the raycasting stage, but

also because our empty-space skipping algorithm belongs to the object-order category, so its speed is not very sensitive to the viewport size. In contrast, previous pure image-order methods are strongly dependent upon the viewport size [16].

As in all previous volume rendering methods, the rendering speed is closely related to the viewing distance. When the object is far from the camera (as in Fig. 1), the working set is small and most bricks selected for rendering are at the upper level of the octree, so the speed can be faster than 40 fps. When the camera is very close to the object (as shown in Fig. 10), the working set is large and all of the bricks selected are at, or near, their finest levels; even so, our system can still achieve interactive speed even for the close-up views shown in Fig. 10 and the accompanying videos.

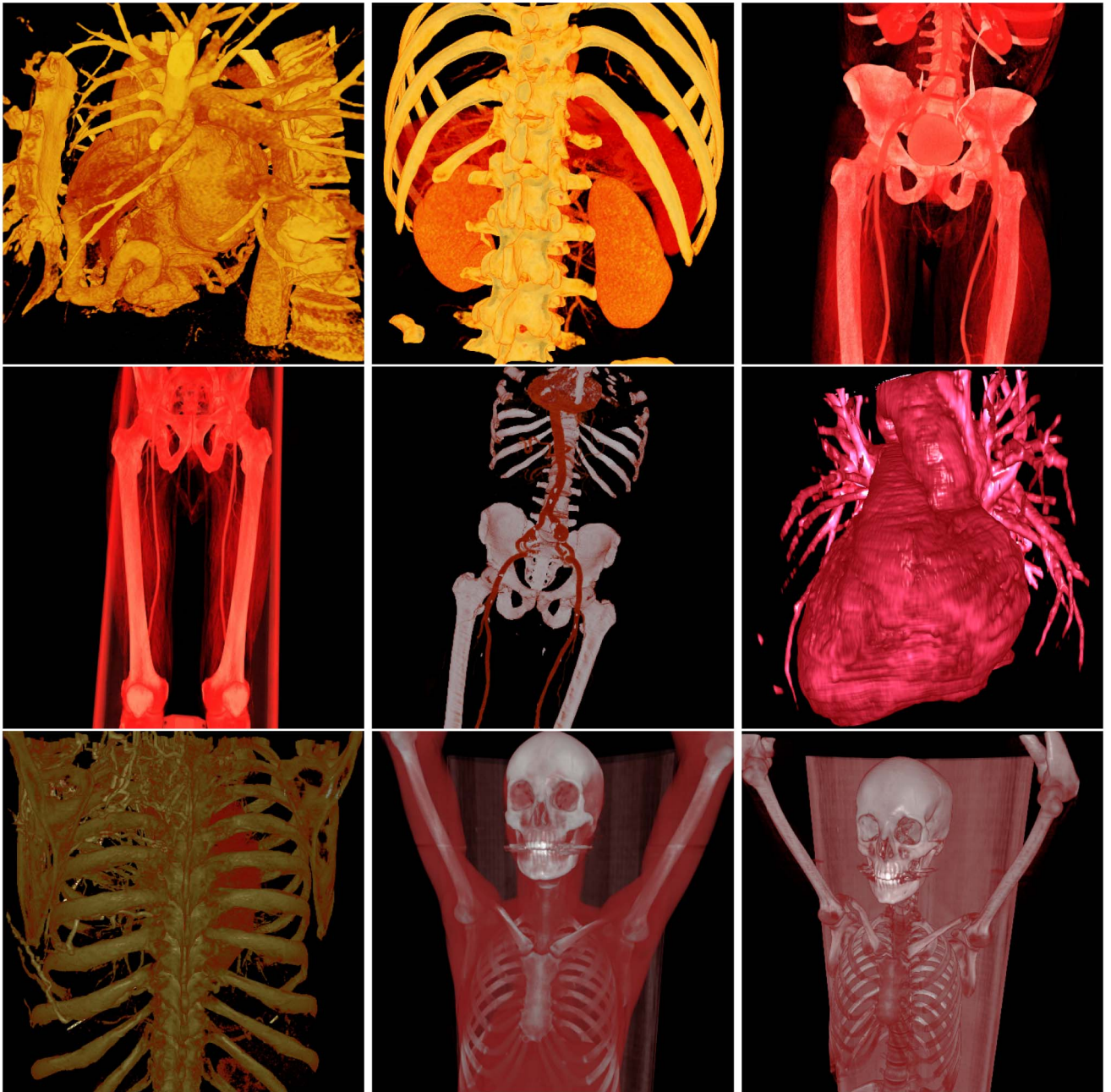


Fig. 10. Closeup rendering (close to camera) of more medical data sets (from left to right, top to bottom: Toutatix, Abdomen, Obelix, Amnesix, Maco, Heart, Tragicomix, Melanix1, and Melanix2) rendered by our methods. Please zoom-in to see the details. The rendering performance statistics of these images are given in Table 1.

Since our method employs the HistoPyramid [33] to efficiently extract all of the active macrocells inside the bricks of different LoDs, it also has the benefit of supporting arbitrary changes to the transfer function at runtime with no extra cost, as demonstrated in the accompanying videos.

5.5 Limitations

Our main goal is to design a method capable of achieving the real-time rendering of very large volumetric data on a general PC that has only a cheap consumer-level GPU. Thus, we seek to load into GPU memory only the data that is visible, and that only at the resolution required—in other words, a view-dependent cut of the octree. We make heavy use of frame-to-frame temporal and spatial

coherence by reusing data that was loaded by previous frames. Once the data is loaded into GPU memory, we want it to stay there as long as possible in order to maximize its reuse among multiple frames and minimize the cost of transferring data from system memory. This is the typical use case in which the users slowly rotate or move the objects using a mouse, as shown in our demo videos; in such cases, most of the currently visible data bricks will already have been loaded for the previous frames and only a very few newly appearing data bricks will need to be loaded for the current frame, particularly as we typically achieve a frame rate of 5-14 fps (see Table 1). As a result, the data transfer should not be a bottleneck in typical use.

The rate achieved for the transfer of the data bricks from the CPU memory of the host to the GPU memory was, at most, 1.4 GB/second. Although this rate is far less than the peak bandwidth of the PCI Express 2.0×16 bus interface used (8 GB/second), it is still fast enough for smoothly varying the viewing conditions (by exploiting temporal and spatial coherence between consecutive frames due to our coherently generated view-dependent working set for each frame), as shown in the accompanying videos. This is because only a few new bricks (those that have not appeared in previous frames) need to be transferred to GPU memory; all other bricks are already cached there.

However, if the user rotates or translates the objects very quickly, for example, by suddenly moving from a front view to a back view, then most of the currently visible data bricks will be different from those previously loaded and will thus have to be transferred from system memory to GPU memory. For such a nontypical use case, data transfer (of the entire brick-pool) could become the bottleneck and make the rendering very slow. Such cases are not favorable for the approach presented, and it is possible that a CPU-based in-core approach (such as [22], which uses superlarge CPU memory of 32 GB to hold the entire data set and its accelerating structure in-core) might do better. How to reduce data transfer between the CPU and GPU could also constitute an area that deserves further research.

5.6 GPU Memory Consumption

For the results reported, we used a granularity of brick and macrocell at $B_{res} = 128$ and $M_{res} = 8$, respectively. By experimentation, we found that these values provide an optimal balance among the octree operating overhead, the cost of rasterization and the efficiency of the empty-space skipping. The larger these two parameters are, the lower will be the octree-related operating overhead and the rasterizing cost, but at the same time the less will be the efficiency of the empty-space skipping.

We found that, at runtime, the total GPU memory consumption of our system is fixed at 1,419 MB, which is data set-independent since only the octree-nodes in the current view-dependent cut are loaded into the GPU memory.

The majority of the GPU memory is consumed by the brick-pool, which contains $P_x \times P_y \times P_z = 8 \times 8 \times 9 = 576$ slots. Each slot occupies 2 MB of voxel data of a brick, so altogether 1,152 MB is consumed by the brick-pool.

Although the macrocell pool has the same number of slots, its memory size is much smaller than the voxel data of the brick. Each slot contains only $(B_{res}/M_{res})^3 = 16^3 = 4,096$ texels, each with two channels: the min-max scalar values. The macrocell pool thus consumes only $4,096 \times 2 \times 576 = 4.6$ MB. The remaining GPU memory is occupied by other data textures and all the shaders.

6 CONCLUSIONS AND FUTURE WORK

We have presented a fast GPU-based out-of-core volume ray caster, which moves the branching intensive octree traversal out of the GPU raycasting loop and executes it on the CPU. Greater control is exercised over the rendering process by introducing tighter depth ranges for GPU

raycasting by using the hardware rasterizing unit to effectively generate a per-pixel list of tighter ray-segments.

Experiments showed that the method presented provides higher quality at the same time as faster performance. It offers more sophisticated empty space skipping, which makes it possible to make interactive use of advanced features such as cubic interpolation for large out-of-core data sets.

Since the method presented provides a general accelerating scheme (by rasterizing an octree to generate tight ray segments), it could also be used to improve the performance of other visualization systems, such as in multiple user activities, comparative settings and multiple volume studies, or time-varying data visualization in future work.

ACKNOWLEDGMENTS

The authors would like to thank Thomas Fogal for the helpful suggestions, and Mark Duchaineau for the data set. This work was partially supported by the European Commission within the Marie Curie project GAMVoIV is (FP7-PEOPLE-IF-2008-236120).

REFERENCES

- [1] ImageVis3D: A Real-time Vol. Rendering Tool for Large Data. Scientific Computing and Imaging Inst. (SCI), <http://www.imagevis3d.org>, 2009.
- [2] The OpenGL registry, Sept. EXT_shader_image_load_store, http://www.opengl.org/registry/specs/EXT/shader_image_load_store.txt, 2010.
- [3] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-Time Rendering*, third, ed., A.K. Peters, Ltd., 2008.
- [4] J. Amanatides, "Ray Tracing with Cones," *ACM SIGGRAPH Computer Graphics*, vol. 18, no. 3, pp. 129-135, 1984.
- [5] L. Bavoil and K. Myers, "Order Independent Transparency with Dual Depth Peeling," Nvidia Graphics SDK 10.5, 2008.
- [6] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann, "Gigavoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering," *Proc. Symp. Interactive 3D Graphics and Games (I3D '09)*, pp. 15-22, 2009.
- [7] C. Crassin, F. Neyret, M. Sainz, and E. Eisemann, *Efficient Rendering of Highly Detailed Volumetric Scenes with GigaVoxels. In book: GPU Pro*. A.K. Peters Ltd., 2010.
- [8] C. Dyken, G. Ziegler, C. Theobalt, and H.-P. Seidel, "High-Speed Marching Cubes Using Histograms," *Computer Graphics Forum*, vol. 27, no. 8, pp. 2028-2039, Dec. 2008.
- [9] K. Engel, M. Hadwiger, J. Kniss, C. Rezk-Salama, and D. Weiskopf, *Real-Time Volume Graphics*. A.K. Peters Ltd., 2006.
- [10] C. Everitt, "Interactive Order-Independent Transparency," Research Report, NVIDIA Corporation, 2001.
- [11] T. Fogal, H. Childs, S. Shankar, J. Krüger, R.D. Bergeron, and P. Hatcher, "Large Data Visualization on Distributed Memory Multi-GPU Clusters," *Proc. Conf. High Performance Graphics (HPG '10)*, pp. 57-66, 2010.
- [12] T. Fogal and J. Krüger, "Tuvok, An Architecture for Large Scale Volume Rendering," *Proc. 15th Int'l Workshop Vision, Modeling, and Visualization*, pp. 57-66, Nov. 2010.
- [13] E. Gobbetti, F. Marton, and J. Iglesias Guitián, "A Single-Pass GPU Ray Casting Framework for Interactive Out-of-Core Rendering of Massive Volumetric Data Sets," *The Visual Computer*, vol. 24, nos. 7-9, pp. 797-806, July 2008.
- [14] S. Grimm and S. Bruckner, "Memory Efficient Acceleration Structures and Techniques for Cpu-Based Volume Raycasting of Large Data," *Proc. IEEE/SIGGRAPH Symp. Volume Visualization and Graphics*, pp. 1-8, 2004.
- [15] S. Guthe, M. Wand, J. Gonser, and W. Strasser, "Interactive Rendering of Large Volume Data Sets," *Proc. IEEE Conf. Visualization*, pp. 53-60, 2002.
- [16] M. Hadwiger, P. Ljung, C.R. Salama, and T. Ropinski, "Advanced Illumination Techniques for GPU Volume Raycasting," *Proc. ACM SIGGRAPH*, pp. 1-166, 2009.

- [17] M. Hadwiger, C. Markus, H. Sigg, K. Scharsach, M. Bühler, and Gross, "Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces," *Computer Graphics Forum*, vol. 24, no. 3, pp. 303-312, 2005.
- [18] W. Hong, F. Qiu, and A. Kaufman, "GPU-Based Object-Order Ray-Casting for Large Data Sets," *Proc. Fourth Int'l Conf. Volume Graphics*, pp. 177-185, 2005.
- [19] D.M. Hughes and I.S. Lim, "Kd-Jump: A Path-Preserving Stackless Traversal for Faster Isosurface Raytracing on GPUs," *IEEE Trans. Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1555-1562, Nov./Dec. 2009.
- [20] B. Kainz, M. Grabner, A. Bornik, S. Hauswiesner, J. Muehl, and D. Schmalstieg, "Ray Casting of Multiple Volumetric Data Sets with Polyhedral Boundaries on Manycore GPUs," *ACM Trans. Graphics*, vol. 28, pp. 152:1-152:9, Dec. 2009.
- [21] A. Knoll, Y. Hijazi, R. Westerteiger, M. Schott, C. Hansen, and H. Hagen, "Volume Ray Casting with Peak Finding and Differential Sampling," *IEEE Trans. Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1571-1578, Nov./Dec. 2009.
- [22] A. Knoll, S. Thelen, I. Wald, C. Hansen, H. Hagen, and M. Papka, "Full-Resolution Interactive CPU Volume Rendering with Coherent BVH Traversal," *Proc. IEEE Pacific Visualization Symp. (PacificVis '11)*, pp. 3-10, 2011.
- [23] J. Krüger and R. Westermann, "Acceleration Techniques for GPU-Based Volume Rendering," *Proc. IEEE 14th Visualization*, pp. 287-292, 2003.
- [24] M. Levoy, "Display of Surfaces from Volume Data," *IEEE Computer Graphics Applications*, vol. 8, no. 3, pp. 29-37, May 1988.
- [25] B. Liu, G.J. Clapworthy, and F. Dong, "Accelerating Volume Raycasting Using Proxy Spheres," *Computer Graphics Forum*, vol. 28, no. 3, pp. 839-846, 2009.
- [26] B. Liu, G.J. Clapworthy, and F. Dong, "Fast Isosurface Rendering on a GPU by Cell Rasterisation," *Computer Graphics Forum*, vol. 28, no. 8, pp. 2151-2164, 2009.
- [27] F. Liu, M.-C. Huang, X.-H. Liu, and E.-H. Wu, "Single Pass Depth Peeling via CUDA Rasterizer," *Proc. ACM SIGGRAPH*, 2009.
- [28] P. Ljung, "Adaptive Sampling in Single Pass GPU-Based Raycasting of Multiresolution Volumes," *Proc. Eurographics/IEEE Int'l Workshop Volume Graphics*, pp. 39-46, 2006.
- [29] W.E. Lorensen and H.E. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *ACM SIGGRAPH Computer Graphics*, vol. 21, no. 4, pp. 163-169, 1987.
- [30] C. Sigg and M. Hadwiger, "Fast Third Order Texture Filtering," *GPU Gems 2*, M. Pharr and R. Fernando, eds., pp. 313-329, Addison-Wesley, 2005.
- [31] V. Vidal, X. Mei, and P. Decaudin, "Simple Empty-Space Removal for Interactive Volume Rendering," *J. Graphics Tools*, vol. 13, no. 2, pp. 21-36, 2008.
- [32] J.C. Yang, J. Hensley, H. Grün, and N. Thibieroz, "Real-Time Concurrent Linked List Construction on the GPU," *Computer Graphics Forum*, vol. 29, no. 4, pp. 1297-1304, 2010.
- [33] G. Ziegler, A. Tevs, C. Theobalt, and H.-P. Seidel, "On-the-Fly Point Clouds Through Histogram Pyramids," *Proc. Int'l Workshop Vision, Modeling and Visualization (VMV '06)*, pp. 137-144, 2006.



Baoquan Liu received the PhD degree in computer graphics in 2007 from State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences. He joined the Research Centre for Computer Graphics and Visualisation (CCGV) in the Department of Computer Science and Technology, University of Bedfordshire in 2008. His areas of interest and experience include computer graphics, rendering, GPU, and visualization.



Gordon J. Clapworthy received the BSc degree (Hons, Class 1) in mathematics, the MSc degree (dist.) in computer science from The City University, London, and the PhD degree in aeronautical engineering from the University of London. He is a professor of computer graphics in the Department of Computer Science and Technology and head of the Centre for Computer Graphics and Visualization (CCGV) at the University of Bedfordshire, United Kingdom.



Feng Dong received the BSc and MSc degrees in physics and the PhD degree in computer graphics, all from Zhejiang University, China. Currently, he is a professor of visual computing in the Department of Computer Science and Technology, University of Bedfordshire, United Kingdom. His research interests include fundamental computer graphics algorithms, texture synthesis, image-based rendering, medical visualization, volume rendering, human modeling, and rendering, and VR.



Edmond C. Prakash received the BE, ME, and PhD degrees from Annamalai University, Anna University, and the Indian Institute of Science, respectively. He is a professor of computer games technology in the Department of Computer Science and Technology at the University of Bedfordshire. His research focus is on exploring the applications of virtual reality in engineering, science, medicine, and finance.

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