Microsoft Tech Report MSR-TR-2010-72: Volume Rendering on Server GPUs for Enterprise-Scale Medical Applications

T. Sharp, D. Robertson, A. Criminisi

Microsoft Research, Cambridge, UK

Abstract

We describe a system for volume rendering via ray casting, targeted at medical data and clinicians. We discuss the benefits of server vs client rendering, and of GPU vs CPU rendering, and show how we combine these two advantages using nVidia's Tesla hardware and CUDA toolkit. The resulting system allows hopsital-acquired data to be visualized on-demand and in real-time by multiple simultaneous users, with low latency even on low bandwidth networks and on thin clients. Each GPU serves multiple clients, and our system scales to many GPUs, with data distribution and load balancing, to create a fully scalable system for commercial deployment. To optimize rendering performance, we present our novel solution for empty space skipping, which improves on previous techniques used with CUDA. To demonstrate the flexibility of our system, we show several new visualization techniques, including assisted interaction through automatic organ detection and the ability to toggle visibility of pre-segmented organs. These visualizations have been deemed by clinicians to be highly useful for diagnostic purposes. Our performance results indicate that our system may be the best-value option for hospitals to provide ubiquitous access to state-of-the-art 3D visualizations.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

1. Introduction

Volume rendering algorithms take an input signal defined on a three-dimensional domain and project it onto a two-dimensional image. Volume rendering has a long history [Lev90, Neu92, LL94, EKE01, EWRS*06, TSD08, SHC*09] with applications in a variety of different domains such as mechanical engineering [SMW*05, RBG07], scientific visualization [Lev90] and medical analysis [RTF*06, BCFT06, TSD08, SHC*09]. Although impressive visualizations have often been obtained, the main challenge has remained that of achieving such visualizations efficiently.

This paper addresses the problem of efficient remote rendering of medical volume data for a commercial system suitable for hospital-scale deployment. Unlike previous systems where rendering has been done on GPU workstations or on CPU clusters, we make the leap to GPU clusters, in order to provide clinicians with ubiquitous access to 3D visualizations, on thin clients and over low bandwidth internet connections. This approach yields challenges related to simultaneous clients, load balancing, transport and efficiency. Our system achieves rendering of diagnostic quality 3D images, as confirmed by radiology experts. It is being developed as part of a commercial radiology product.

In this paper we describe our implementation of volume rendering using CUDA, including a novel space skipping algorithm which improves on previous results. We go on to discuss the architecture of our enterprise solution for remote rendering of medical images using GPU-equipped servers and thin clients. We show performance evaluation results for individual renderings, and scalability results which demonstrate our ability to scale with the number of simultaneous clients. Finally we show some novel visualization techniques which are enabled by our remote rendering solution and discuss their clinical relevance.

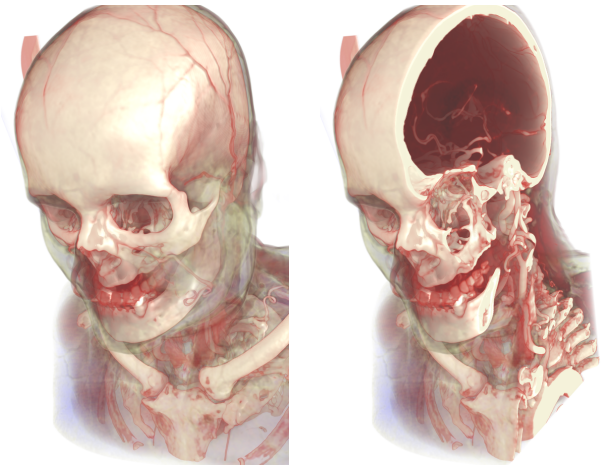

Figure 1: *A user adds a clipping plane with their mouse wheel to reveal internal structure.*

2. Related Work

Although the theory of direct volume rendering is well understood a number of expedients have been sought in order to make the rendering process more computationally efficient. Some of the early approaches used a surface representation [Lev88]. Other techniques relied on efficient geometric transformations to avoid the inefficiencies of ray-casting [LL94, Lac95]. Others exploited existing texture hardware [WE98, RSEB*00, LMC01]. Algorithm parallelism, empty-space skipping, tree-based data structure and frequency domain processing have also been used for computational efficiency and data compression [Neu92, SGL94, MPHK93, LMK03, WFM*05, KW03, TL93, VKG04, SMW*05, GWGS02]. Interactive remote rendering is addressed in [ESE00, HHN*02]. More recently, the introduction of multi-core graphics processors and the new nVidia CUDA platform has led to new, faster algorithms [PN01, Waa08, KGB*09].

The usefulness of volume rendering in medical applications has been demonstrated in a number of research papers [Lev90, BCFT06, RBE08, TSD08, KGB*09, SHC*09]. Specifically, volume rendering has been found useful in analyzing Computed Tomography data [HRS*99, PDSB05], fused CT and MRI data [RBE08], as well as functional MRI and in DTI tractography [HBT*05, RTF*06]. Specific medical applications include: vascular analysis [BA01], tissue classification [SWB*00] and diagnosis and treatment of aneurisms [THFC97]. The recent work of Smelyanskiy et al. [SHC*09] demonstrates that ray-casting produces the best diagnostic quality, medical renderings.

The problem of improving the diagnostic clarity of medical renderings has been tackled either by using multi-dimensional transfer functions [KKH02] or by using some level of segmentation to remove clutter and focus on the regions of interest [BCFT06, Bru06, RBG07]. The remote visualization of medical data was discussed in [EEH*00, THRS*01]. Parallel volumetric rendering on GPU clusters was addressed in [SMW*05].

More information on the vast volumetric visualization literature may be found in [Lju06, EWRS*06, SCCB, Yag, Bey09].

3. Rendering

At the core of our system is efficient volume rendering using CUDA. We perform volume rendering via ray casting, where we shoot one ray per pixel, through the camera centre and into the volume.

In previous work on GPU volume rendering [EWRS*06], setting up rays has been done by a vertex shader, rendering the faces of the bounding cube, with the 3D position of each vertex encoded in a texture coordinate. Since we are using CUDA rather than a graphics API, we have no need of this method, and instead simply back-project each pixel through an inverse projection matrix.

We are given a 3D scalar array of dimension (w,h,d) and use it to define a right-handed local coordinate system on the volume data. We use these discrete values in conjunction with standard trilinear texturing to define a sampling function $V(x,y,z):\mathbb{R}^3 \longrightarrow [0,1]$. We wish to render the volume to an ARGB image $I(x,y):\Omega \subset \mathbb{Z}^2 \longrightarrow [0,1]^4$. The client software provides a 4-by-4 transformation matrix P which projects from volume co-ordinates to image co-ordinates. The third row of P allows for near and far clipping planes to be specified.

For each render request, the volume-to-viewport matrix P is inverted on the CPU and passed to the CUDA kernel along with other rendering parameters. The method we describe here supports both affine and perspective cameras, which may be inside or outside the bounding volume. The ray-casting kernel begins by computing a ray $\mathbf{r}(\lambda) = \mathbf{a} + \lambda\hat{\mathbf{d}}$ for each pixel. For an image pixel with index (x,y), $\mathbf{a} = \mathbf{s}(0)$, where $\mathbf{s}(z) = \lfloor P^{-1}(x+0.5, y+0.5, z, 1) \rfloor$, and $\lfloor . \rfloor$ represents the operation of de-homogenizing a vector by dividing its entries by its last component. Then $\mathbf{d} = \mathbf{s}(1) - \mathbf{a}$, and $\hat{\mathbf{d}} = \mathbf{d}/|\mathbf{d}|$. We will proceed to integrate along the ray between $\lambda_{near} = 0$ and $\lambda_{far} = |\mathbf{d}|$.

Before integration, we clip each ray to all the active clipping planes. These planes include both the bounding box of the volume and any planes specified by the user. Each clipping plane π is a 4-vector such that a point (x,y,z) shall be visible if and only if $\pi^T(x,y,z,1) > 0$. λ_{near} and λ_{far} are updated by intersecting each ray with each clipping plane. At this stage, empty space skipping may also be performed (see section 4).

We may now define the sampling positions along the ray as $\{\mathbf{r}(\lambda_j)\}$ for $\lambda_j = \lambda_{near} + j.\lambda_{step}$ and $j \in [0, (\lambda_{far} -$

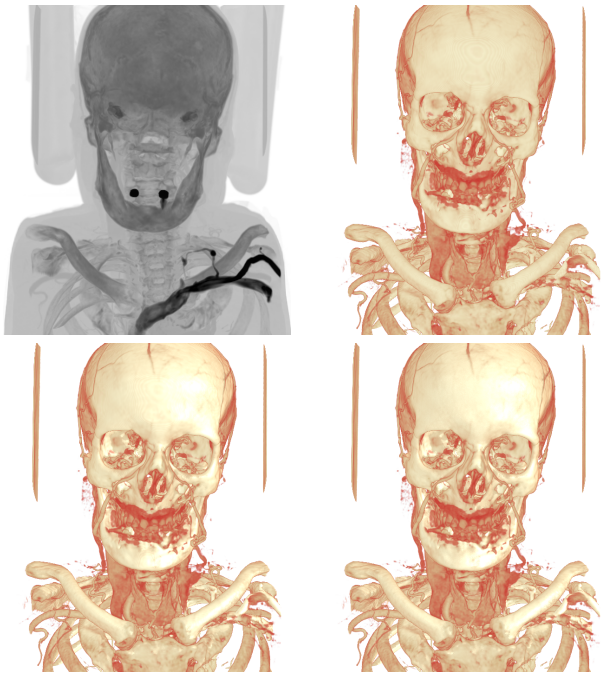

Figure 2: *The effect of various rendering options. Top-left: MIP integration (see [PB07]); top-right: alpha blending integration; bottom-left: with shading added; bottom-right: jittering the ray start positions. See text for details*

$\lambda_{near})/\lambda_{step}$). (We always set $\lambda_{step} = 1$.) We then step along the ray at these positions, sampling the texture at each position as $V(\mathbf{r}(\lambda))$ and integrating as appropriate.

We support multiple integration modes, as required for clinical medical use [PB07]. For *MIP* (Maximum Intensity Projection), the result of integration $I(x,y) = \max_j\{V(\mathbf{r}(\lambda_j))\}$ is simply the maximum of the sampled scalar values along the ray (see figure 2). *Minimum* and *Average* values are similarly computed.

For the *Alpha Blending* integration mode, during integration, an associated [] (or pre-multiplied) ARGB colour vector is updated along each ray as $\mathbf{c} = \mathbf{c} + (1 - c_\alpha)src_j$, where src_j represents the amount of light and opacity emitted or reflected along the viewing direction at position $\mathbf{r}(\lambda_j)$.

The client supplies a transfer function as a set of keypoints $\{(x,\alpha,r,g,b)_j\}$ such that $\forall j : x_j, \alpha_j \in [0,1]$ and $r_j, g_j, b_j \leq \alpha_j$ are pre-multiplied colours. We can then define a sampling function $\mathbf{T}(x) : [0,1] \longrightarrow [0,1]^4$ which maps scalar volume values to colours and opacities. In this case, $src_j = \mathbf{T}(V(\mathbf{r}(\lambda_j)))$.

We also use pre-integration of the transfer function [EKE01] to avoid unnecessary aliasing where the transfer function may contain high frequencies. In this case, $src_j = \mathbf{T}_{2D}(V(\mathbf{r}(\lambda_{j-1})), V(\mathbf{r}(\lambda_j)))$, where $\mathbf{T}_{2D} : \mathbb{R}^2 \longrightarrow [0,1]^4$ is a pre-computed texture such that

$$\mathbf{T}_{2D}(x,y) = \begin{cases} \int_x^y \mathbf{T}(\tau)\,d\tau/(y-x) & (x \neq y) \\ \mathbf{T}(x) & (x = y) \end{cases}$$

We estimate gradients during integration using 6-point finite differences in order to perform shading:

$$\nabla V(x,y,z) \approx \frac{1}{2\delta} \begin{bmatrix} V(x+\delta,y,z) - V(x-\delta,y,z) \\ V(x,y+\delta,z) - V(x,y-\delta,z) \\ V(x,y,z+\delta) - V(x,y,z-\delta) \end{bmatrix}.$$

To mitigate wood-grain artefacts, we expose an option to jitter the ray start position by up to one voxel in the direction of the ray. In this case, $\lambda_j = \lambda_{near} + (j + J(i,j))\lambda_{step}$, where $J(i,j) : \mathbb{R}^2 \longrightarrow [0,1]$ is a pre-computed jitter texture.

To improve performance, we implemented early ray termination [EWRS*06], so that if c_α becomes sufficiently close to 1 during integration, the ray computation may be considered finished.

4. Empty space skipping

The most important optimization for volume rendering is empty space skipping. We tried several approaches to empirically determine the most appropriate strategy for our context.

4.1. Depth-First Tree Traversal

One strategy that briefly appeared promising was depth-first traversal of a BSP tree. This strategy is typically used for CPU rendering. We constructed a binary tree where the root node corresponds to the entire volume. The tree is constructed by repeatedly bisecting the volume along the dimension with longest length. At each node we stored both the minimum and maximum scalar values within the corresponding sub-volume. Thereafter it is relatively straightforward to perform the ray casting with reference to the BSP tree.

However, there are two significant problems with this approach in our context. First is the issue of thread divergence. Within the CUDA architecture, threads in a thread group (called a warp) must all execute the same instruction simultaneously to achieve good performance. If different threads take different code paths this can be disastrous for performance. The thread divergence in this strategy makes it quite unsuitable for GPUs.

We tried to avoid the problem of thread divergence by simply using the tree to determine the ray start position, λ_{near}. However, we found that this scheme was still not efficient since it was necessary to store the stack or queue of visited nodes for each thread, which is required for the depth-first traversal. This per-thread storage requirement was too great for either registers or shared memory, and therefore

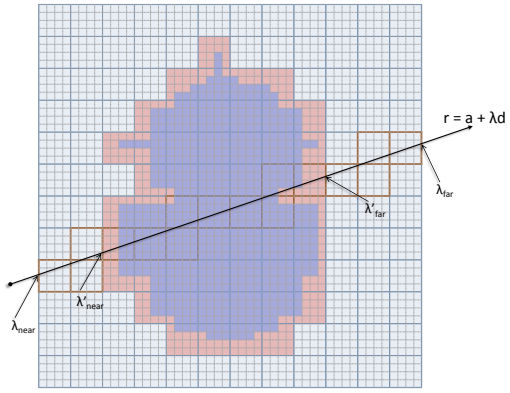

Figure 3: *Empty space skipping represented in 2D. The smallest squares represent the underlying volume data; the larger squares represent the bricks whose minimum and maximum values are stored in a low-resolution proxy volume. We traverse all the bricks that intersect a ray (shown in brown outline), determining whether each bricks is visible (pink in diagram). The ray bounds λ_{near} and λ_{far} are tightened accordingly.*

the cost of accessing and updating this stack dominated the render time.

4.2. Our Proposed Method

Previous methods for GPU space skipping rasterize precomputed bounding geometry to determine the start position for each ray. This has typically been performed on a vertex shader [LMK03, EWRS*06], although [KGB*09] implements the strategy using CUDA.

Instead, we propose a new multi-resolution method for empty space skipping. We create a pyramid of low-resolution proxy volumes where each voxel in a low-resolution volume stores only the minimum and maximum values within the corresponding brick at the high resolution. For each ray, after clipping the ray to bounding planes, we interesect the ray with the lowest resolution proxy volume, and visit each intersected brick in the range $[\lambda_{near}, \lambda_{far}]$. We walk through the low resolution voxels, from front to back in a Bressenham-style 3D line-drawing routine (see Figure 3).

For each intersected brick, we compute the ray parameters λ_f, λ_b at the front-most and back-most intersection points of the ray with the brick. At the first non-transparent brick, we set $\lambda_{near} := \lambda_f$ and $\lambda_{far} := \lambda_b$. At subsequent non-transparent bricks, $\lambda_{far} := \lambda_b$. When an opaque brick is encountered, the ray may terminate.

With pre-integrated transfer functions, it is simple to determine whether a brick is transparent from the minimum and maximum scalar volume values. The pre-integrated transfer function already stores the average opacity of the transfer function between each pair of scalar values. Therefore a brick can be considered transparent precisely when the pre-integrated opacity between minimum and maximum values is below a threshold close to zero, i.e. when $\mathbf{T}_{2D}(v_{min}, v_{max}) < \theta$. We use $\theta = 0.002$. A similar computation discovers fully opaque bricks.

After stepping along the rays through the lowest resolution proxy volume, we have updated values of λ_{near} and λ_{far} for each ray, which are tighter to the visible volume data than the original bounds. We then proceed to the next finest resolution and repeat, beginning each ray at the new λ_{near} position. In this way we progressively update the near and far clipping positions for each ray with efficient parallelism.

Finally, we use the tightest bounds for each ray in the integration phase, which uses the original volume data. We implement each level of ray clipping in a separate CUDA kernel invocation.

For the scalar integration modes, e.g. MIP, there are similar rules for adjusting $\lambda_{near}, \lambda_{far}$ based on whether each brick may contain values that affect the integration result.

For volumes up to 512^3 in size, we found that the best results were produced with two proxy volumes, using scale factors of 8 and 64. In this case, overall rendering time was reduced by between 35% and 78% (depending on the transfer function) compared to 15% reported in [KGB*09]. The two low-resolution passes accounted for about 25% of the total time. See the results in figure 4.

We think of our space skipping algorithm as being akin to a *breadth-first* traversal of a BSP tree, as opposed to the depth-first traversal which is better suited to CPUs.

5. Architecture

The design of our system is motivated by a few key requirements. In a typical hospital environment, CT scans are performed more or less continually, with 3D data sets stored in a database on a server within the hospital's network. We aim to provide interactive diagnostic-quality volume rendering of these data sets for simultaneous remote users, over low-bandwidth connections and on thin clients, while minimizing hardware cost. These remote users are not only radiologists that typically see the patient scans, but other clinicians such as surgeons, oncologists and general practitioners who may currently only see a textual description provided by the radiologist. The patients themselves may also perhaps wish to view their data, as it has been shown that a visual depiction incentivizes patients to a greater degree of involvement in their own health [Vis].

To achieve these goals, we architect our system for all rendering to be performed on server GPUs. We submit that this choice gives us a number of benefits over both client rendering and CPU rendering.

Transfer function	No skipping (ms)	Our skipping (ms)	Speed-up
Internal	38.8	25.4	35%
Muscle	45.2	20.3	55%
Native	58.0	12.7	78%
Bones	34.9	16.7	52%

Figure 4: *Results for our proposed empty space skipping technique, for 4 different transfer functions, shown above. Transfer functions are ordered clockwise, from top-left to bottom-left.*

5.1. Server vs Client Rendering

In our context of hospital installation, performing rendering on server machines provides several advantages over client rendering. First, it should be noted that the 3D data sets, generated by MRI/CT scanners, are stored in a back-end database. In smaller configurations, this storage server may be the same machine as the rendering server. In larger configurations, the rendering and storage servers will be nearby on a high bandwidth connection, and the rendering server will cache the data sets it retrieves from storage. Therefore the data sets, which can be around 1GB in size, already exist on the server. Transferring these data sets across the wire to client machines over potentially low bandwidth networks would cause an unacceptable delay in interaction of perhaps several minutes, in an environment where every second of visualization counts.

Second, we know that volume rendering is highly compute and memory intensive, and we do not wish to impose cumbersome restrictions on client machines in terms of their CPUs and GPUs. By performing the rendering remotely, we enable the surgeon at home with her netbook to see the same visualizations as the radiologist in his lab with a workstation.

Third, by concentrating the required compute power in the data centre, we may leverage efficiencies and economies of scale, reducing installation costs. By performing load balancing among multiple processors we achieve the scalability that allows us to serve many different client requests simultaneously. Thus, rather than needing to equip each client machine with the relevant hardware for rendering, we can equip our servers according to price or performance constraints, without restricting the clients that may request rendered images.

5.2. GPU vs CPU Rendering

The benefits of server rendering for commercial medical environments have long been understood by, for example, Fovia [Fov]. But whereas Fovia's products perform rendering on CPUs, we have designed for server rendering on GPUs. In the literature, GPUs have been preferred for volume rendering in the last decade due to their much greater parallelism, their buit-in trilinear texture sampling, and their superior memory bandwidth. In the last few years, the gap between CPU and GPU compute power has continued to grow, both in real terms and in computational value (flops per dollar).

5.3. Target Hardware

With the introduction of the CUDA platform, nVidia enabled general purpose programming on their GPUs without the need to program a graphics API. But with the introduction of the Tesla hardware, it became feasible to add powerful GPU compute ability to clusters within data centres. The Tesla S1070 for example comprises four C1060 GPUs, each with 4GB of memory and 240 cores, with a theoretical total peak performance of 4 Tflops in a 1U rack-mount case [nvi]. The S1070 connects via cables to the PCIe bus of a host machine. Two S1070s may be connected to each server for a total of 8 GPUs on each rendering machine. In 2008, the Tokyo Institute of Technology famously boosted the compute power of their TSUBAME supercomputer by adding 170 Tesla S1070s.

To our knowledge, we are the first to present a system for multiple-client volume rendering which is designed to leverage this server GPU environment. We designed our system for a cluster with one or more rendering servers running Windows Server 2003 or later, each with one or more nVidia GPUs attached.

The recently announced Tesla S2070 will replace the S1070 with more than twice as many cores and 6 times as much GDDR5 memory.

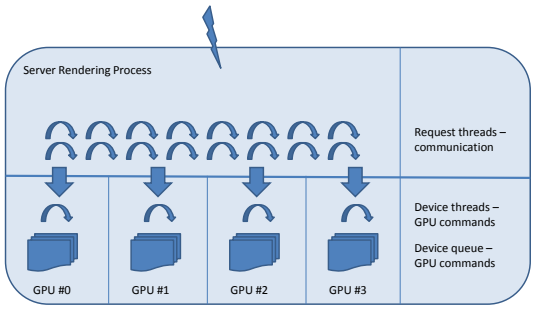

Figure 5: *Threading model of server process. Diagram shows a server with four GPUs attached.*

5.4. Scalability

A client session begins by requesting a volume. This request is handled by a master node, which selects a particular rendering server to handle the client's rendering requests for this volume. Subsequent requests from the client to render the volume are sent directly to the negotiated rendering server. We have found that an effective method for selecting the server is simply to look for the GPU with the most free memory, and return its host's address.

The volume data is then loaded as a 3D texture to the selected GPU attached to the rendering server. Thus each viewport's rendering requests are served by exactly one GPU.

Each rendering server has multiple GPUs attached and each GPU serves multiple clients. It is important for the CUDA runtime that calls which use GPU resources should occur in the same thread that allocates and frees those resources. Therefore it is essential to get the process and threading model right.

On each rendering server we run a process as a Windows service which receives client requests, performs rendering and replies with data as appropriate. Within this process, we start one *device thread* for each GPU attached to the server. As client requests arrive at the server, simultaneously on different *calling threads*, we determine which GPU must serve each request, and serialize the request data to the appropriate device thread (figure 5). Each time a device thread receives a render request, it wakes up if necessary and processes its render request queue in order. When the device thread has finished processing a render request, it signals an event to notify the calling thread. The calling thread then returns the data to the client.

5.5. Programming

In the first instance we have programmed our transport and request layers using COM/DCOM. For the GPU layer we have used the CUDA Toolkit 2.3.

We support various rendering options (see section 3). We program these different options using C++ templates directly in the CUDA code. Each different integration method, data access method, shading algorithm or transfer function type is a function object [Str97]. This ensures that only the appropriate code is compiled in for each set of options.

This method, known as compile-time polymorphism or static polymorphism, has long been popular in computer graphics scenarios. Its disadvantage is that combinations of different options can cause exponential growth in compiled code size. Generally speaking, runtime-generated code is preferable, but often difficult to produce. In the present case, there seems to be no obvious way to compile CUDA code at run-time.

6. Performance

6.1. Client-Side Visualization Software

In concert with our GPU-based volume rendering server, we have developed client-side software for interactive visualization of medical volume datasets. The software, which is designed to run under Windows, can be configured to communicate with our COM-based rendering software running either on the local machine or (via DCOM) on a remote server. The software user interface provides most of the functionality typically used by clinicians to study medical volume datasets, including the facility to load volume datasets from filesystem storage, to choose the rendering mode, interactively to manipulate the viewpoint and transfer function, and to define and interactively manipulate clipping planes. We have used this visualization software to render a wide variety of volumes (including all the rendering examples shown in this paper).

6.2. Rendering Performance

Most authors quote rendering performance statistics obtained using typical volume datasets and viewport sizes. However, because the efficiency improvements obtained by implementing empty space skipping depend critically on the geometry of the non-empty region of the volume (which is a function of the applied transfer function), it is difficult to make meaningful performance comparisons with other systems. With this problem in mind, we have tried to select medical volumes similar in size to those used by other authors but, in addition, we have made all of our test data (including volume datasets, projection matrices, transfer functions) publicly available[†] so that others can more easily obtain comparable results in future. Our volume datasets, which we believe to be broadly representative of the more demanding datasets likely to be encountered in modern hospitals, are available both as standard DICOM series, and in

[†] See http://www.XXXXXXXXXX.com

Figure 6: *Representative* 512×512 *renderings of the datasets used for performance evaluation (transfer functions in brackets): (i) Head (Internal), (ii) Legs (Muscle) (iii, iv) Body (Internal and Bones).*

Time (ms)	Transfer Function			
Volume	Internal	Muscle	Bones	Native
Head	13	13	10	7
Torso	20	18	10	10
Legs	20	18	5	8

Figure 7: *Mean rendering times (ms) for our test volumes and transfer functions.*

a binary format that can be read into memory easily with a few lines of code (supplied).

Figure 6 illustrates our 3 test datasets. These were visualized from viewpoints arranged in a circle about the centroid of the volume with a viewport of size measuring 512×512. So as to make our results comparable with those of other researchers, we conducted these tests using a GTX285 graphics card, which has a retail price of around €500. Rendering speeds are reported in Table 7. This rendering performance appears to be comparable with speed obtained by the fastest published systems [KGB*09].

6.3. Scalability Testing

In the context of our multi-user, server-based rendering application, one limitation of the raw rendering performance measures obtained in the previous subsection is that they don't take account of other activities likely to increase the computational load on the server, such as retrieving volume datasets from filesystem storage. Another is that, because interactive visualization of real datasets does not generally require frames to be rendered at constant rate, it is difficult to extrapolate from such figures more meaningful statistics, such as the number of users that could be supported simultaneously under typical or peak loads.

To evaluate the likely performance of our system under realistic operating conditions, we designed a test framework to simulate server load representative of that likely to be generated in a real hospital. We achieved this by adding command logging and playback capabilities to our volume rendering software.

The command logging capability was designed to record all the requests made of the volume rendering service, including requests to load volume data into memory and to modify transfer functions, clipping planes, and projection matrices.

The playback capability allowed us to replay prerecorded command sequences to the server with timing closely representative of the original sequence. Our playback software can play back many prerecorded sequences simultaneously in multiple threads - allowing simulation of multiple users simultaneously using the same server.

During playback, we reproduce the timing associated with an original sequence of requests as closely as possible. After a request has been serviced, each playback thread waits until the start time of the next request. For rendering requests, we measure the latency associated with the call since this is directly representative of the latency and frame rate experienced by the user during interactive usage. Render request latencies were scored in three categories:

- GOLD means a latency of < 67 ms
- SILVER means a latency of < 100ms.

To determine how our software would be used, we observed clinicians interacting with medical volume datasets. Using this information, we recorded some test sequences (of around five minutes' duration) that closely represented the clinicians' behaviour. These sequences comprised the following components:

- Load a 512^3 volume (c. 20 s)
- Load several transfer functions (c. 15 s)
- Interactive adjustment of the transfer function (c. 10 s)
- Adjust the viewpoint interactively at 15 fps (c. 150 s)
- Define some clipping planes and adjust them interactively at 15 fps (c. 40 s)
- Passively inspect rendered frames (c. 10 s)

We explore the performance impact of varying the number of simulatenous clients by replaying sequences in multiple threads and measuring the proportion of rendering requests that were serviced with GOLD and SILVER latency. So as to avoid any artificial correlation between N playback threads, we offset start times for successive threads by a time interval

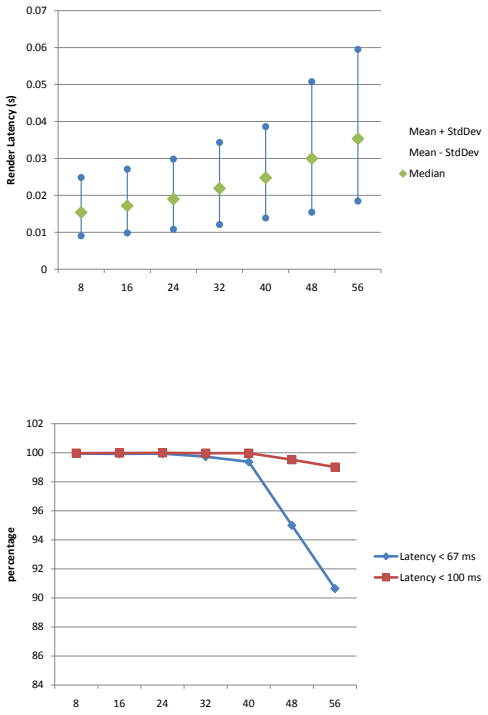

Figure 8: *Results of scalability testing under representative loading. Top: Distribution of render request latencies versus number of simultaneous clients. Bottom: Percentage of requests serviced with GOLD (< 67 ms) and SILVER (<100 ms) latency vs. number of simultaneous clients.*

D/N where D is the sequence duration. So as to avoid making performance measurements in the start up period, each sequence is replayed three times but performance statistics are obtained only for the middle repetition when the maximum number of playback threads is always active.

We performed these tests on a Dell T5500 workstation with 8 Intel Xeon cores, 16GB RAM and one attached nVidia Tesla S1070 (four GPUs). Figure 8 shows how rendering performance scales with the number of simultaneous interactive clients under representative loading. Each GPU has 4GB of memory and so we were only able to support 15 simultaneous clients per GPU before running out of memory. As the load nears the maximum that can be supported, we see that performance decreases approximately linearly.

Our design target was for 90% of rendering requests to be serviced within a 67 ms (GOLD) latency so that we could achieve good responsiveness at interactive frame rates of at least 15 fps. From the graph, we see that our system can support approximately 52 simultaneous interactive users at this frame rate.

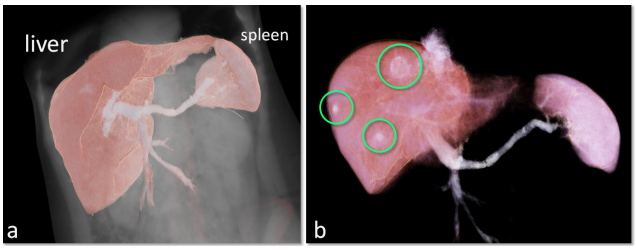

Figure 9: *Volumetric rendering for the diagnosis of liver hemangioma.*

The current cost for the Tesla hardware which supports these 50-60 simultaneous interactive clients is approximately €4K-6K, equating to a hardware cost of under €100 per simultaneous 3D user.

7. Applications

In this section we show applications of our remote rendering system to radiology diagnosis. Figure 9 shows an example. In a patient's CT scan the liver and spleen have been segmented via the interactive technique in [CSB08]. In fig. 9a the foreground objects are visualized as opaque and the background with a highly translucent transfer function. In our visualization system, each segmented layer has an associated colour transfer function and thus the visual characteristics of each layer can be individually modified. In fig. 9b the background has been removed completely and the transparency of the liver increased so as to show structures within. This highlights the anomalous hemangiomas (shown in green circles) in this patient.

In fig. 10a we show a mandible and clavicles from a CT scan. The visualization highlights both the dense bony structures and the bone marrow inside. Simultaneous visualizations of these diverse (and occluding) tissues is of fundamental importance for the diagnosis of osteoporosis or cancerous conditions. Figure 10b shows some selected vertebrae and the bone marrow within.

Figure 11 we show 3D renderings of a CT scan of a patient's abdomen. In fig 11b the segmented aorta is highlighted and the background removed. In this view the presence of two aneurysms (enlargements of blood vessels) becomes very clear, together with numerous calcifications along the vessels walls. Figure 11c highlights the presence of a thrombus (blood clot) within the bigger aneurysm.

Figure 12 shows integration of our volumetric rendering with a recent automatic organ detection technique [CSB09]. Running an automatic organ detection algorithm in the background allows new, single-click visual navigation. On the

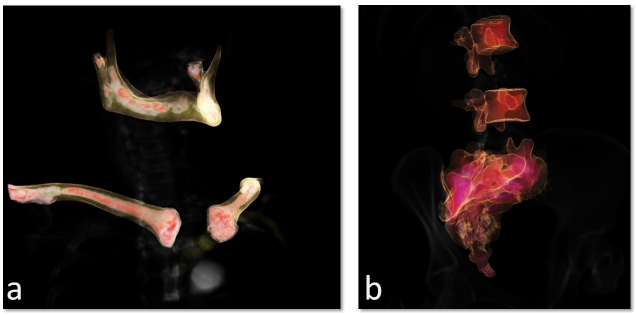

Figure 10: *Volumetric rendering for the analysis of bony structures and the bone marrow within.*

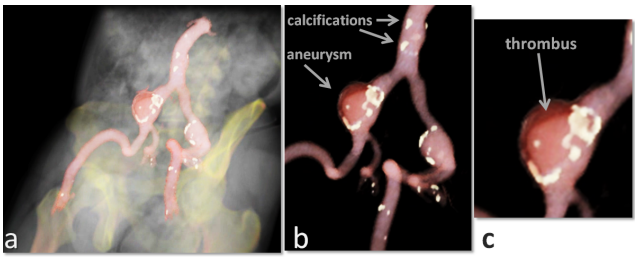

Figure 11: *Volumetric rendering for the diagnosis of aneurysms and thrombus in blood vessels.*

right hand side of the viewer a list of detected structures is automatically populated. Now the user can simply click on the structure of interest to perform the following operations: i) the virtual camera is re-positioned to best view the selected structure; ii) clipping planes are automatically activated; iii) the colour transfer function is modified to best view the selected structure. This novel one-click navigation technique greatly improves the efficiency of typical radiology workflows, which is of paramount importance to these practitioners.

The diagnostic value of our visualizations has been confirmed by the professional radiologists to whom we have informally shown our system at the RSNA09 meeting in Chicago.

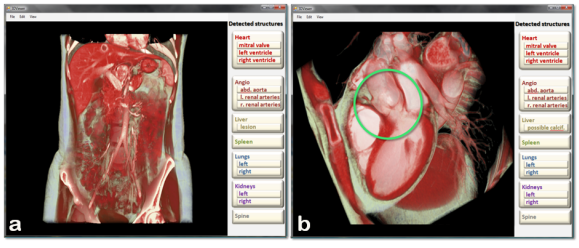

Figure 12: *Visualizing automatically detected organs to improve clinical workflow efficiency. (a) a clipped view of a patient's abdomen. (b) shows the effect of selecting the* heart / mitral valve *option.*

8. Conclusions

By exploiting the power of general purpose GPU hardware and a novel GPU-based algorithm for empty space skipping, we have developed a volume rendering implementation that gives state-of-the-art performance. Our test data have been made publicly available in the hope that other researchers will more easily be able to compare their own rendering performance results.

Based on this implementation, we have developed a scalable server-based architecture for enterprise-scale volume rendering that is capable of supporting a large number of simulataneous users. To validate our system, we have conducted scalability testing by carefully simultating likely real-world loads. We show that a system based on an 8-core Intel Xeon server equipped with four Tesla GPUs can support up to 60 clients at interactive frame rates. We believe the price performance ratio is unprecedented.

References

[BA01] BULLITT E., AYLWARD S.: Volume rendering of segmented tubular objects. In *MICCAI* (2001). 2

[BCFT06] BORLAND D., CLARKE J., FIELDING J., TAYLOR R. M.: Volumetric depth peeling for medical image display. *Visualization and Data Analysis* (2006). 1, 2

[Bey09] BEYER J.: *GPU-based Multi-Volume Rendering of Complex Data in Neuroscience and Neurosurgery*. PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Oct. 2009. 2

[Bru06] BRUCKNER S.: Exploded views for volume data. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (2006), 1077–1084. 2

[CSB08] CRIMINISI A., SHARP T., BLAKE A.: Geos: Geodesic image segmentation. In *European Conference on Computer Vision (ECCV)* (2008). 8

[CSB09] CRIMINISI A., SHOTTON J., BUCCIARELLI S.: Decision forests with long-range spatial context for organ localization in ct volumes. In *MICCAI workshop on Probabilistic Models for Medical Image Analysis (MICCAI-PMMIA)* (2009). 8

[EEH*00] ENGEL K., ERTL T., HASTREITER P., TOMANDL B., EBERHARDT K.: Combining local and remote visualization techniques for interactive volume rendering in medical applications. In *VIS '00: Proceedings of the conference on Visualization '00* (2000), pp. 449–452. 2

[EKE01] ENGEL K., KRAUS M., ERTL T.: High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (2001), pp. 9–16. 1, 3

[ESE00] ENGEL K., SOMMER O., ERTL T.: A framework for interactive hardware accelerated remote 3d-visualization. In *In Proc. TCVG Symp. on Vis. (VisSym* (2000), pp. 167–177. 2

[EWRS*06] ENGEL K., WEISKOPF D., REZK-SALAMA C., HADWIGER M., KNISS J.: *Real-time Volume Graphics*. A K Peters Ltd, 2006. 1, 2, 3, 4

[Fov] http://www.fovia.com/. 5

[GWGS02] GUTHE S., WAND M., GONSER J., STRASSER W.: Interactive rendering of large volume data sets. pp. 53–60. 2

[HBT*05] HARDENBERGH J., BUCHBINDER B. R., THURSTON S. A. M., LOMBARDI J. W., HARRIS G.: Integrated 3d visualization of fmri and dti tractography. In *Visualization* (2005). 2

[HHN*02] HUMPHREYS G., HOUSTON M., NG R., FRANK R., AHERN S., KIRCHNER P., KLOSOWSKI J. T.: Chromium: A stream-processing framework for interactive rendering on clusters. *ACM Trans. on Graphics (Proc. of SIGGRAPH) 21*, 3 (2002). 2

[HRS*99] HASTREITER P., , REZK-SALAMA C., TOMANDL B., EBERHARDT K., ERTL T.: Interactive direct volume rendering of the inner ear for the planning of neurosurgery. *BVM* (1999). 2

[KGB*09] KAINZ B., GRABNER M., BORNIK A., HAUSWIESNER S., MUEHL J., SCHMALSTIEG D.: Ray casting of multiple volumetric datasets with polyhedral boundaries on manycore gpus. In *ACM Trans. on Graphics, (SIGGRAPH ASIA)* (2009). 2, 4, 7

[KKH02] KNISS J., KINDLMANN G., HANSEN C.: Multidimensional transfer functions for interactive volume rendering. *IEEE Transactions on Visualization and Computer Graphics 8*, 3 (2002). 2

[KW03] KRUGER J., WESTERMANN R.: Acceleration techniques for gpu-based volume rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)* (2003), IEEE Computer Society. 2

[Lac95] LACROUTE P.: Real-time volume rendering on shared memory multiprocessors using the shear-warp factorization. In *Parallel Rendering Symposium* (1995). 2

[Lev88] LEVOY M.: Display of surfaces from volume data. *IEEE Comput. Graph. Appl. 8*, 3 (1988). 2

[Lev90] LEVOY M.: Volume rendering by adaptive refinement. *The Visual Computer 6*, 1 (1990). 1, 2

[Lju06] LJUNG P.: *Efficient Methods for Direct Volume Rendering of Large Data Sets*. PhD thesis, DEPARTMENT OF SCIENCE AND TECHNOLOGY, LINKOPING UNIVERSITY, SE-601 74 NORRKOPING, SWEDEN, 2006. 2

[LL94] LACROUTE P., LEVOY M.: Fast volume rendering using a shear-warp factorization of the viewing transformation. *ACM Trans. on Graphics (Proc. of SIGGRAPH)* (1994). 1, 2

[LMC01] LUM E. B., MA K. L., CLYNE J.: Texture hardware assisted rendering of time-varying volume data. In *VIS '01: Proceedings of the conference on Visualization '01* (2001), IEEE Computer Society. 2

[LMK03] LI W., MUELLER K., KAUFMAN A.: Empty space skipping and occlusion clipping for texture-based volume rendering. *IEEE Visualization* (2003). 2, 4

[MPHK93] MA K.-L., PAINTER J., HANSEN C., KROGH M.: A data distributed, parallel algorithm for ray-traced volume rendering. In *Parallel Rendering Symposium* (1993). 2

[Neu92] NEUMANN U.: Interactive volume rendering on a multicomputer. In *SI3D '92: Proceedings of the 1992 symposium on Interactive 3D graphics* (1992). 1, 2

[nvi] http://www.nvidia.com/. 5

[PB07] PREIM B., BARTZ D.: *Visualization in Medicine*. Morgan Kaufmann, 2007. 3

[PDSB05] PERSSON A., DAHLSTROM N., SMEDBY O., BRISMAR T.: Volume renderinf of three dimensional drip infusion ct cholangiography in patients with suspected obstructive biliary disease: a retrospective study. *The British Journal of Radiology 78* (2005). 2

[PN01] PATIDAR S., NARAYANAN P.: Ray casting deformable models on the gpu. In *Proc. Computer Vision, Graphics and Image Processing (ICVGIP)* (2001). 2

[RBE08] RÖSSLER F., BOTCHEN R. P., ERTL T.: Dynamic Shader Generation for Flexible Multi-Volume Visualization. In *IEEE Pacific Visualization Symposium 2008 (PacificVis '08)* (2008), pp. 17–24. 2

[RBG07] RAUTEK P., BRUCKNER S., GRÖLLER M. E.: Semantic layers for illustrative volume rendering. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (2007). 1, 2

[RSEB*00] REZK-SALAMA C., ENGEL K., BAUER M., GREINER G., ERTL T.: Interactive volume on standard pc graphics hardware using multi-textures and multi-stage rasterization. In *HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (2000), pp. 109–118. 2

[RTF*06] RÖSSLER F., TEJADA E., FANGMEIER T., ERTL T., KNAUFF M.: Gpu-based multi-volume rendering for the visualization of functional brain images. In *In Proceedings of SimVis 2006* (2006). 1, 2

[SCCB] SILVA C., COMBA J., CALLAHAN S., BERNANDON F.: A survey of gpu-based volume rendering on unstructured grids. *Brazilian Journal of Theoretic and Applied Computing*. 2

[SGL94] SINGH J. P., GUPTA A., LEVOY M.: Parallel visualization algorithms: Performance and architectural implications. *Computer 27*, 7 (1994), 45–55. 2

[SHC*09] SMELYANSKIY M., HOLMES D., CHHUGANI J., LARSON A., CARMEAN D. M., HANSON D., DUBEY P., AUGUSTINE K., KIM D., KYKER A., LEE V. W., NGUYEN A. D., SEILER L., ROBB R.: Mapping high-fidelity volume rendering for medical imaging to cpu, gpu and many-core architectures. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1563–1570. 1, 2

[SMW*05] STRENGERT M., MAGALLÏ£¡N M., WEISKOPF D., GUTHE S., ERTL T.: Large Volume Visualization of Compressed Time-Dependent Datasets on GPU Clusters. *Parallel Computing 31*, 2 (2005), 205–219. 1, 2

[Str97] STROUSTRUP B.: *The C++ Programming Language*. Addison Wesley, 1997. 6

[SWB*00] SATO Y., WESTIN C.-F., BHALERAO A., NAKAJIMA S., SHIRAGA N., TAMURA S., KIKINIS R.: Tissue classification based on 3d local intensity structures for volume rendering. *IEEE Transactions on Visualization and Computer Graphics 6*, 2 (2000). 2

[THFC97] TAM R. C., HEALEY C. G., FLAK B., CAHOON P.: volume rendering of abdominal aortic aneurysms. In *Visualization* (1997). 2

[THRS*01] TOMANDL B. F., HASTREITER P., REZK-SALAMA C., ENGEL K., ERTL T., HUK W., GANSLANDT O., NIMSKY C., EBERHARDT K.: Local and remote visualization techniques for interactive direct volume rendering in neuroradiology. *Radiographics* (2001). 2

[TL93] TOTSUKA T., LEVOY M.: Frequency domain volume rendering. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques* (1993), pp. 271–278. 2

[TSD08] TATARCHUK N., SHOPF J., DECORO C.: Advanced interactive medical visualization on the GPU. *Journal of Parallel and Distributed Computing 68*, 10 (Oct. 2008), 1319–1328. 1, 2

[Vis] http://www.thevisualmd.com/. 4

[VKG04] VIOLA I., KANITSAR A., GRÖLLER M. E.: Gpu-based frequency domain volume rendering. In *Proceedings of SCCG'04* (2004), pp. 49–58. 2

[Waa08] WAAGE J.: State of the art parallel computing in visualization usinf cuda and opencl. In *INF358 Seminar in Visualization* (2008). 2

[WE98] WESTERMANN R., ERTL T.: Efficiently using graphics hardware in volume rendering applications. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (1998). 2

[WFM*05] WALD I., FRIEDRICH H., MARMITT G., SLUSALLEK P., SEIDEL H.-P.: Faster isosurface ray tracing using implicit kd-trees. In *IEEE Transactions on Visualization and Computer Graphics (IEEE VIS)* (2005). 2

[Yag] YAGEL R.: Volume viewing algorithms: Survey. In *In International Spring School on Visualization*. 2