Web based Out-of-Core Volume Visualization in Client-Server Architectures

Neslisah Torosdagli*, Sumanta Pattanaik
CS Division, Dept. of EECS, UCF
Orlando, USA
*neslisah@knights.ucf.edu

Curtis Lisle
KnowledgeVis, LLC, Maitland, FL
Yanling Liu
Frederick National Lab for Cancer Research
Frederick, Maryland

Abstract— **High quality, interactive volume visualization is increasingly important as advancements in volumetric capture devices, increased GPU power, and greater interest from the biological/medical community are all currently happening. Researchers and clinicians want to access their data anywhere using a wide range of devices, from smart phones up to powerful multicore machines. Web based visualization applications are increasingly popular now that browsers are supported on nearly all computing platforms, and the new HTML5 standards enable graphical interaction without the need for proprietary plugins or additional APIs. In this paper, we present a novel, browser-based, out-of-core volume rendering tool that enables interactive rendering of large volume data on portable devices. The tool uses a client-server approach, where the rendering task is carried out on a server, and the interaction and viewing occurs through a client-system web browser.**

Index Terms—**Volumetric data visualization, web-technologies, GPU computation, out of core, client-server application.**

I. Introduction

Many applications in a wide range of domains such as medicine, biology, physics, and engineering benefit from visualization of volumetric data at interactive rates. The performance of GPUs has been increasing rapidly in recent years and their parallel processing power can now be harnessed in visualization applications using OpenCL/Cuda or OpenGL/DirectX APIs [1]. However, volume rendering is still challenging for larger volumes that exceed the resources available on the rendering hardware. This is particularly true for mobile clients. Although there are some large-scale volumetric visualization tools available such as Vaa3D-TeraFly[8], none of them fulfill requirements especially for mobile clients. Hence comes the need for a volume visualization architecture that can scale in performance by using server-based computing, yet deliver the volumes to small or mobile devices.

In this paper, we present a browser based adaptive out-of-core volume ray-casting system for interactive visualization of volumetric big data on devices from smart phones and tablets to high end workstations. In spite of many attempts by researchers to develop portable interactive visualization systems using proprietary or specialized plugins and APIs, the portability and wide adaptation of those systems has been limited because the actual software ports to non-native platforms are not trivial. We have addressed this problem by making our system web based. Web browsers are supported in almost all computing platforms. HTML5, the new mark-up language for web content with native interaction support, introduction of new elements such as *canvas,* and integration of *scalable vector graphics* (SVG) content, allows us to perform interactive volume rendering in our web-based system without having to resort to proprietary plugins. Thus our system is portable to a wide range of devices and platforms.

To be interactive requires rendering the volume at interactive rates (generally more than ten frames per second). The computation cost associated with a volume renderer, makes it nearly impossible to guarantee interactive frame rates independent of the computing power of the user's device. To address this problem, we have developed a client-server framework; where the server is responsible for compute/storage/memory intensive tasks, such as volume processing and rendering the view. The client is responsible for interaction, transfer function manipulation and display of the image frames rendered by the server. The appropriate choice of the server hardware configuration and a decent network connectivity will deliver interactive frame-rates. Any browser-based device, serving as a client, will allow the user to interact with the volume and interactively view the image rendered by the server.

To our knowledge, ours is the first web-based, out-of-core volume visualization system.

II. Background

In this section we will provide a quick background to the algorithms directly relevant to this paper. For a detailed discussion on volume rendering, we recommend interested readers to the book [2] and for scalable GPU rendering to the recent survey articles[3].

A. Front-to-Back Ray Casting Algorithm

Ray casting based volume rendering techniques assume that the volume is composed of a grid of emitting and absorbing volume elements (voxels). The ray-casting step accumulates light from voxels encountered along the path of the ray after taking into account the volumetric attenuation of light from the point of origin to front of the volume. The front-to-back ray casting method has an advantage over back-to-front ray casting, in that it accumulates opacity (the degree to which light is not allowed to travel through) along ray and terminates the accumulation process when the opacity is close to 100% -- thus reducing rendering cost.

The emission and absorption property of the voxel is assigned using a transfer function. The transfer functions play an important role in volume rendering and designing them is, in itself, an entire area of research, which is out of the scope of this paper. In our work, we have assumed that the transfer function is interactively assigned by the user.

B. Out-of-core Volume Rendering

Like most ray tracing algorithms, ray casting based volume rendering is easily parallelizable and makes effective utilization of the GPU processing capabilities. However, the available GPU memory (and even CPU memory) is often inadequate for the size of modern day biological/medical volumes. Out-of-core techniques have been proposed to handle this problem. Such techniques often use a standard memory-paging scheme, in which large volumes are broken into smaller chunks (called *bricks*) such that multiple bricks can easily fit into the GPU memory. The bricks are retrieved and processed in a view-specific order, for example: starting with the bricks nearest to the view, followed by the bricks further away in the view frustum. This technique works best when a small part of the volume is visible and requires a relatively small number of brick transfers. However, when the whole volume or a relatively larger portion of the volume is in the view, the cost of streaming full-resolution volume bricks can significantly reduce the frame rate. Level-of-detail based techniques, similar to mip-map construction in traditional texture are used to reduce the streaming cost. A hierarchy with progressively lower resolution bricks is constructed from the original bricks. Using a top-down traversal technique, the bricks at the appropriate resolution for their location are identified and streamed. This scheme significantly reduces the cost of brick transfers and improves the frame rate. As an added bonus, the view dependent resolution doubles as a *minification* filter and hence improves the rendered image quality by reducing aliasing. Such out-of-core techniques have been well researched in the literature. We have followed a technique similar to [4][5].

C. Client-Server Architecture

Using a client-server based system, the compute/storage/memory intensive component is executed on a server equipped with high-performance CPU and/or GPU hardware. The output is then delivered to one or more client devices (often less powerful) through a connected network. The additional effort lies on building a communication layer, using some existing client-server protocol. We have used a web-based, thin-client system that handles the image frame display, interaction with the volume, and the interactive transfer function creation.

III. Our System

A. Adaptive Out-of-Core Volume Ray Casting

In our adaptive out-of-core volume ray-casting algorithm the volume dataset is partitioned into an octree hierarchy of bricks, and as needed, before rendering every frame, a view dependent candidate set of the bricks from this hierarchy is computed and transferred to GPU along with an index texture describing mapping of the available bricks to the GPU memory.

The volume is first partitioned into bricks. The brick size is set to be some power of 2-cube (say a cube of size $32\times32\times32$).

Processing of volume data, which includes partitioning of volume data into the octree hierarchy and pre-computing gradients, is a time-consuming task and hence is, carried out in the GPU in a preprocessing step. The results are stored for later use.

In the preprocessing stage, in addition to volume data and octree structure, histograms and lookup tables are also computed and stored. Since an octree inherently requires a power of 2 cube, processing any rectangular, non power of 2 volume, requires us to create nonexisting (hereto called "ghost") bricks, that are not physically stored anywhere, but are required for octree handling. We use a lookup table to keep track of such ghost bricks. Depending on the data size, the preprocessing step can be very time consuming. For example: preprocessing Visible Human Project female dataset takes around 12 hours to preprocess data on Intel Core i7-4770K-GeForce GTX 780 with 15.6GB Memory.

In our system, every brick in the octree hierarchy is assigned a unique id consisting of a two-component tuple (*level#*, *index*), where index is the flattened 3D index of the brick in the volume. Depending on the system memory capacity, bricks are stored in physical or virtual CPU memory.

For every new rendered frame (triggered by interactive viewing and/or transfer function update) we traverse through the octree hierarchy in top-down order to compute a candidate brick set (see Fig. 1. a). The resolution of the bricks in the candidate set are chosen such that the number of voxels on the face of the brick closely matches with the number of pixels in the foot-print of the view-dependent projection of the brick bounding volume on the display window.

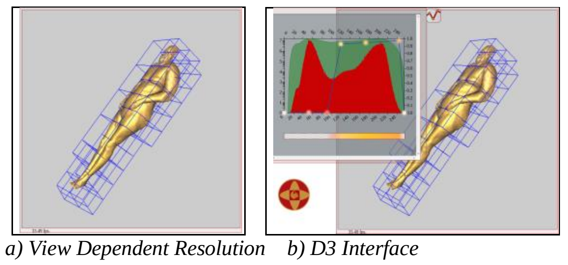


Fig. 1. Octree-hierarchy and D3 Interface

Required bricks' resolutions are identified according to their locations by traversing octree in breadth-first order. The allocated GPU memory is organized as a cubic volume whose size is an exact multiple of brick size. Thus each brick in the candidate set can be assigned a unique location in the GPU resident volume.

Once the candidate brick set computation is completed, an index table is created to map voxels encountered at render time to the appropriate brick in the GPU cubic volume. The size of the index table is set to be equal to the number of the bricks at the leaf level. The candidate set is processed one brick at a time in the order in which they appear, and a tuple, composed of the octree level of the candidate brick and its location in GPU memory, is assigned to the cells of the table that correspond to all the descendant leaf bricks of the candidate brick. The index table is stored as a two-channel 2D integer texture. Once the index table construction is completed, view dependent candidate bricks and the index texture are sent to the GPU, and the front-to-back ray-casting algorithm is executed.

During the ray traversal in front-to-back ray casting, for every point along the ray in the volume, its corresponding CPU brick id is computed, and the index texture pixel for this brick id is read to find the mapped GPU brick id and its octree resolution. If the voxel maps to a brick, then the offset for the voxel data in the corresponding GPU volume memory is computed, and the voxel's properties (emission, its opacity and gradient) are read. If the voxel does not map to any brick, computation continues to the next voxel until either the ray exits the bounding volume or early ray termination occurs. Early ray termination is applied when opacity of the voxel reaches 95 percent or above.

All server side implementations are carried out in C++ using the Boost library. To speed-up brick lookup, the octree of image bricks is stored in a memory-mapped file. GPU programming for octree construction and gradient computation is carried out using OpenCL. If the server is running locally then OpenGL implementation of volume ray casting is used otherwise OpenCL implementation is used. For the former (i.e. locally run server) the user has an option to run in OpenCL also. However, our OpenGL implementation runs faster. Front-to-back ray casting is implemented using a two-pass rendering, where the first pass computes the ray exit points in the volume, and the second pass executes the actual front-to-back ray casting and rendering. The voxel gradients of the volume are used as the normal for shade computation using a Phong lighting model.

B. Web-based Client-Server architecture (see Fig. 2)

We implement a web-based client to make our system accessible from a wide variety of platforms without any setup requirement. The client and server connect over WebSockets [6]. Websockets are chosen due to their full-duplex capabilities on a single socket connection. Although in our implementation, clients trigger most of the actions in the server, full-duplex communication is required as for instance when preprocessing is completed, server needs to send a message to the appropriate client. Our client handles display and interaction with the volume and interactive transfer function creation.

Requests initiated by the client include load new file request, transfer function update, and camera update, among others. The server renders the volume and sends back the rendered frame to the client through the WebSocket connection[6].

WebSocket server side is developed using Boost Library. The server runs n+1 threads running in parallel, where n is the number of simultaneous client interfaces connected to the server. The first thread is used for listening for socket requests and creating a new thread for each connection request. Each successive thread created, handles the rendering requests of the client on the other end of the connection, and transmits the rendered frame to the client.

Client requests are encoded in JSON format. Every render request has an action tag; so the server knows what to perform when it parses the request. There is a queue shared between the listening thread and each client thread. When the listening thread receives the request in JSON format, it parses the request, and puts the appropriate action request in the appropriate shared queue. Client threads, on the other hand, polls the shared queue, and when there is an action request pushed to the queue, the client thread performs a rendering request and transmits back the rendered frame to the appropriate client through the open connection. The rendered frames are transmitted to the clients as binary images.

D3 is used for implementing an interactive Transfer Function editor on the web page (Fig. 1b). The Transfer Function is composed of an array of anchor points, where each anchor point has x, y, r, g, and b float values. x represents gray scale voxel value, y represents transparency of the anchor point, and r, g, and b refer to red, green and blue color components of the mapped color value. The user can interactively add/delete/modify the opacity and the color of an anchor point. The user is allowed to save and retrieve the transfer function.

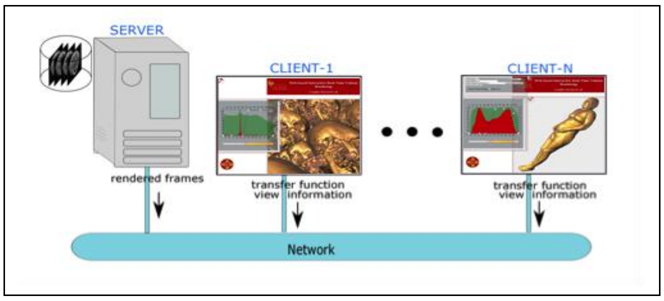
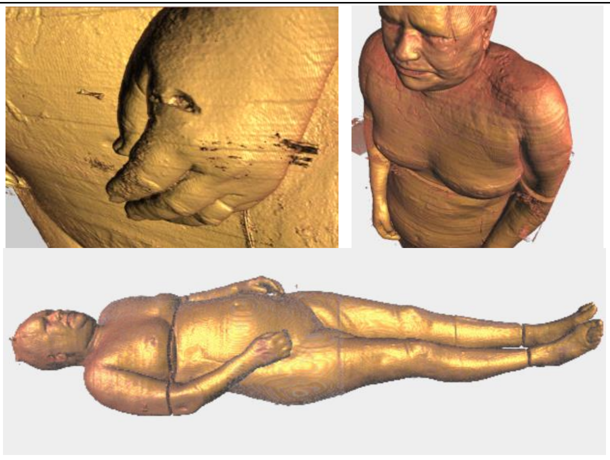


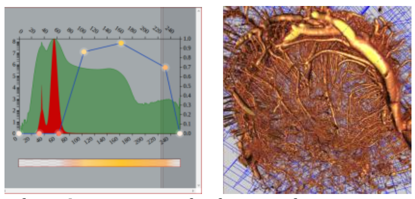
Fig. 2. Client-Server Architecture

IV. Results and Discussion

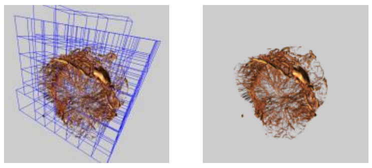
This system is developed as a proof of concept for an interactive and portable client-server architecture and an out-of-core volume ray casting application. Performance is given for a system with Intel Core i7-4770K- GeForce GTX 780 with 15.6GB Memory. The server is implemented in C++. We have experimented with a variety of volume data including Visible Human Project's Female Dataset. Figure 3 shows some renderings of the Visible Human Project's Female Dataset [7], whose size is 2048×1216×5185 and High Resolution Brain CT data (courtesy NCI), whose size is 1588×1588×1173.Visible Human Project's Female dataset required slice-by-slice cleaning before applying our regular preprocessing.



a,b,c) Rendering of Visible Human Project Female Dataset[7]



d) Client interface showing Transfer function for Brain Data set (L) and rendering (R).



e,f) Renderings from Brain Data Set: with (L) and without (R) hierarchical brick outline.

Fig. 3. Client Screenshots

In our current implementation, volume data for both the volumes hold 8 bits per voxel. When client makes a “load a new volume” request, if the volume is not already preprocessed, then the preprocessing is automatically triggered. When preprocessing is completed, server sends message to the requesting client about completion of preprocessing stage. The bricks are stored sequentially in a file. At the time of rendering, when a brick at a certain level is required, its data is fetched from the appropriate position. Since spatially closer bricks are more likely to have similar projected area and belong to the same octree level, they appear adjacent to each other, and hence fetching them using Memory Mapped File (available in Boost library) is fast. We will explore maintaining a cache of bricks in memory as an improvement option for our system when we port the system to a decent server.

We obtain acceptable visual quality, which is highly dependent on the transfer function, at an average frame rate of 10Hz. Although our prototype system is missing many possible optimizations, the overall performance is promising. We are planning additional optimizations for future work.

V. Conclusion and Future Work

We have presented an out-of-core volume ray-casting algorithm with a web-based interface and a client-server architecture. Our system can render large volumes and display through a browser to allow users with mobile devices.

Compute intensive volume processing and out-of-core volume ray casting are performed on a remote server, while the interaction, transfer function design and display of the image rendered by the server is done through a browser running on a client device. The initial implementation results are promising, and interactive visualization is achieved on client devices.

Management of bricks on the GPU is a crucial part of our algorithm. Efficient mapping of the available brick slots in the GPU for the adaptively-computed, view-dependent brick candidates is crucial to the efficiency of the out-of-core algorithm. This can be posed as a classic divisible Knapsack Problem, where the weights of the items (bricks) are the same, but there is high benefit to having all the front bricks at the required resolution. Furthermore, we hope to take advantage of frame-to-frame coherence, and reduce the CPU to GPU transfers required for each frame.

Acknowledgment

This work is fully supported by the Frederick National Laboratory for Cancer Research.

References

[1] J. Beyer, M. Hadwiger, and H. Pfister. "A survey of GPU-based large-scale volume visualization," Proceedings of Eurographics Conference on Visualization, 2014.

[2] K. Engel, M. Hadwiger, J. Kniss, C. Rezek-Salama, and D. Weiskopf, Real-time Volume Graphics, A K Peters, 2006.

[3] T. Fogal, A Schiewe, and J. Kruger, "An analysis of scalable GPU-based ray-guided volume rendering," Proceedings of IEEE Large-Scale Data Analysis and Visualization (LDAV), 2013.

[4] E. Gobbetti, F. Marton, and I. Guiti, “A single-Pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets,” The Visual Computer, vol. 24(7). pp. 787–806, 2008.

[5] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann, “GigaVoxels: ray-guided streaming for efficient and detailed voxel rendering,” Proceedings of Interactive 3D graphics and games (ACM I3D '09), pp 15-22, 2009, New York, NY, USA.

[6] "What Is WebSocket?" WebSocket.org. Web. 8 May 2015.

[7] "The National Library of Medicine's Visible Human Project." U.S National Library of Medicine. U.S. National Library of Medicine, n.d. Web. 01 Sept. 2015. <http://www.nlm.nih.gov/research/visible/visible_human.html>.

[8] "Vaa3D." Vaa3D. N.p., n.d. Web. 01 Sept. 2015. <http://home.penglab.com/proj/vaa3d/home/index.html>