A SOFTWARE ARCHITECTURE FOR DISTRIBUTED VOLUME 
RENDERING ON HPC SYSTEMS

Stefan Zellmann  
Chair of Computer Science  
University of Cologne  
Cologne, Germany  
email: zellmans@uni-koeln.de

Ulrich Lang  
Chair of Computer Science  
University of Cologne  
Cologne, Germany  
email: lang@uni-koeln.de

ABSTRACT
We present a scalable software architecture for distributed direct volume rendering on HPC systems. Our approach allows for generically replacing components along the distributed volume rendering pipeline. Renderer components range from highly specialized GPU renderers that implement state of the art features to more versatile remote renderers, that can make use of numerous distributed memory nodes to exploit sort-last parallel rendering, with each node running a generic renderer component itself. Renderer components that are designed to run on CPUs or GPUs respectively make our software architecture most useful for HPC systems. Using zero configuration networking, our system is able to scale at run time by introducing additional resources without having to reset the whole cluster. Generic I/O subsystems allow for various interprocess communication technologies to be used interchangeably, while generalizing the display phase and decoupling it from the rendering and I/O phases can be exploited to hide latency. We integrate our proposed software architecture into the freely available open source direct volume rendering library Virvo.

KEY WORDS
Visualization Tools, High Performance Computing, Distributed Volume Rendering, Zero Configuration Networking

1 Introduction

Direct volume rendering (DVR) with its evergrowing resource demand regarding computational speed as well as memory consumption often constitutes only a subordinate stage for simulations as they are typically conducted on high performance computing (HPC) systems e.g. by meteorologists or physicists. In order to monitor and probably iterate the outcome of these simulations, the scientist usually needs to copy a significant amount of data to his local workstation in order to perform the ensuing visualization task, since HPC systems like large distributed memory machines often lack the necessary graphics hardware needed by state of the art volume rendering algorithms. In such cases, performing remote rendering directly on the HPC system and only transferring images to the local workstation can be beneficial in order to avoid time-consuming data copies. Volume rendering large datasets on CPUs can cause delays due to rendering bottlenecks or network latencies between the distributed memory nodes, which may be acceptable for monitoring purposes but can be circumvent by a decoupled display phase showing only approximate images while no reliable image information is available.

Because volume dataset sizes grow with the 3rd power of their spatial resolution, the volume rendering task itself often demands for distributed memory architectures to be favored over local rendering. In such cases, dedicated render clusters are utilized, which typically consist of distributed memory nodes, which are themselves equipped with state of the art graphics hardware and allow for fast volume rendering algorithms to be applied in a data parallel fashion.

Either way, when performing remote volume rendering on an HPC system, access to that system is typically not granted to one sole person only, so that a need to manage cluster resources arises. Distributed memory nodes dedicated to rendering need to be allocated based on their local utilization and may be assigned tasks only if enough resources are available. On the other hand, a distributed memory node temporarily unavailable due to heavy utilization needs to report back to the resource management system when available for rendering again. Since volume rendering in our case is intended to be an interactive task, traditional resource management systems that act in a batch fashion are out of the question for our purposes, so that our software architecture comprises an interactive resource management subsystem specifically dedicated to visualization tasks.

This paper introduces a novel software engineering approach especially designed to facilitate distributed volume rendering, structuring the problem in a pipeline fashion with loosely coupled stages, that can be generically replaced with custom logic using subclassing. Our primary objective is to devise a software architecture that can effectively hide the complexities of a highly versatile system that can run on most heterogeneous HPC systems. These may range from high end graphics workstations used to power a virtual environment system over dedicated render clusters with Multi-GPU nodes to large distributed memory systems having no graphics accelerators at all.
The paper is structured as follows. Section 2 gives an overview of related publications on architectural considerations for distributed and remote volume rendering. Section 3 outlines our distributed volume rendering architecture by introducing a distributed volume rendering pipeline and detailing, how applications can benefit of a generalization of the respective stages. On top of this, we describe how our software architecture was integrated into the DVR library Virvo. Section 4 briefly discusses the benefits and shortcomings of our approach, while section 5 concludes this paper and provides some pointers for future work.

2 Related Work


Molnar et al. [10] devise a sorting classification for parallel rendering, which distinguishes parallel rendering based upon when the order of image composition is determined in relation to rendering. Their taxonomy is highly adopted throughout the literature, e. g. by Yu et al. [11], who propose 2-3 swap, a generalization of the binary swap compositing scheme [12] to an arbitrary number of processors.

Vollrath et al. [13] propose a GPU-based volume rendering pipeline coarsely dividing GPU volume rendering into proxy geometry generation (geometry processing), actual rendering (fragment processing) and tone mapping (image processing). Their overall system is devided into three layers that are responsible for data handling, application logic and the actual volume rendering task. Their publication is concerned with the communication between these layers and how it affects the GPU volume rendering pipeline.

Meyer-Spradow et al. [14] present a data-flow network driven rapid-prototyping environment called Voreen. The GPU ray casting based application exposes algorithms like proxy geometry generation, transfer function evaluation and light source evaluation through so called processors, that can be combined to data-flow networks for execution. Their framework is specifically designed to render opaque geometry like glyphs or text labels alongside the volume dataset.

Tuvok [15] is a volume rendering library that is integrated into the ImageVis3D application developed at the University of Utah. Fogal and Krüger’s paper primarily focuses on rendering large datasets, on trading dataset size for image quality by using a progressive display method and on hierarchies for level of detail optimizations.

Elvins [16] proposes a parallel splatting implementation using message passing. Howison et al. [17] present a hybrid parallel volume ray casting renderer using message passing together with shared memory parallelism. Peterka et al. [18] introduce the notion of a parallel volume rendering pipeline and discuss the influence of network communication, parallel rendering and compositing to overall rendering time. Sharp et al. [19] introduce a DVR system implementing a Multi-GPU approach for medical applications. Their system is designed to cope with simultaneous requests from remote users connecting to the remote server using thin clients and low-bandwidth network connections. Their evaluation specifically takes the influence of activities of other users competing for system resources into account by simulating multi-user accesses to the remote server.

Zellmann et al. [20] propose an image-based rendering approach for remote volume rendering that is capable of hiding network latencies or delays introduced by e. g. a rendering bottleneck. They decouple the cheap display phase from the rendering phase by displaying and warping approximate images based on 2.5D data while no newly rendered image data is available from the remote servers. This way, they achieve interactive frame rates of approximately 30 Hz necessary for virtual reality applications.

3 System Architecture

3.1 Distributed Volume Rendering Pipeline

Distributed volume rendering may come in a variety of flavors. Parallel volume rendering dedicates multiple resources to one rendering job. Parallelization may be applied in terms of the sorting classification proposed by Molnar et al. [10]. If the dataset consists of multiple time steps, assigning one resource per time step is also an option. We restrict ourselves to sort-last parallel volume rendering, because with that approach data parallelism can be achieved, although our approach could easily be adjusted to accommodate sort-first parallel rendering too. With a rendering system capable of handling multiple jobs at a time, distributed volume rendering also implies assigning distributed resources per job, with parallel rendering applied to jobs with a higher resource demand and having the jobs with lower demand rendered by one resource only. In particular, our definition of distributed volume rendering not just comprises parallel rendering capabilities but also multi-user support.

We propose the pipeline approach depicted in figure 1 to derive a software architecture for distributed volume rendering and explain where benefits can result from making some of the stages replaceable through custom program logic. While the data distribution and display phase usually are assigned to a single processor, the rendering and compositing stage are typically subject to parallelization.
If distributed volume rendering is performed in terms of assigning a single resource to one job, the compositing stage is omitted and execution directly proceeds with the display phase. Although our approach only considers sort-last parallel volume rendering, sort-first parallelism could easily be accommodated, resulting in a generalization of the compositing stage. Our pipeline approach is similar to the parallel volume rendering pipeline from [18], with the main difference that we identify the display phase as an additional pipeline stage, which determines how the rendered image retrieved from the remote server responsible for compositing is displayed on the remote client.

In the remainder of this section, we discuss how specialization of the stages of our distributed volume rendering pipeline can result in a highly versatile system that can run on heterogeneous hardware platforms and targeting various usage scenarios, ranging from remote data exploration using thin clients to virtual reality applications.

### 3.2 Data Distribution

With distributed volume rendering, data distribution needs to be taken care of so that redundancies in communicating data over the network are reduced and only the data is communicated that is needed on each node. This can be achieved in a variety of ways.

With sort-last parallel rendering, one of the nodes needs to act as the master node, which possesses comprehensive knowledge of how the dataset is distributed among the worker nodes. We employ k-d trees [21] as a visibility sorting data structure for the volume dataset (cf. section 3.3), with each worker node being responsible for exactly one leaf node. This suggests a top-down data distribution layout, where volume data is distributed from the master node to the worker nodes. Depending on whether there is a parallel file system attached to the HPC system and whether the whole dataset may be loaded to the main memory of each worker node, our software architecture comprises two alternative ways of distributing the volume data to the worker nodes.

- The master node distributes the axis aligned bounding box (AABB) of each leaf node to the worker node that is responsible for rendering it. Along with that information, the location of the dataset on the parallel file system is communicated, so that each worker node can load its own copy of the volume dataset. This implies that each worker node can store the whole dataset in its main memory and thus abandons data parallelism.
- The AABBs are distributed as before. Each worker node allocates enough main memory to accommodate its share of the volume dataset. The master node sends the whole dataset using IP Multicast. The worker nodes, which registered to the same Multicast group the master node sent to, retrieve each UDP package from the Multicast stream and only store it permanently, if it is contained in the AABB of its leaf node.

Having the worker nodes load the whole dataset from a parallel filesystem will impose the network load upon the connection between the worker nodes and the file server, but may be useful if the master node is used for alpha compositing only and is not meant to permanently store the whole dataset.

IP Multicast implies using UDP, which is unreliable. We ensure that each datagram reached its recipient by communicating additional meta information and fall back to sending the dataset to individual worker nodes using TCP if Multicast failed. Nevertheless, because worker nodes are usually in the same local network as the master node, this happens infrequently.

### 3.3 Renderer Components

That a DVR application which provides facilities for different rendering modalities will benefit from generalization through a subclassing mechanism is obvious. We provide OpenGL renderers that sample the volume using 3D texture slicing as well as a ray casting implementation on the GPU with CUDA [22]. Other renderers implement the shear-warp algorithm on the CPU or the GPU or software ray casting on the CPU (cf. figure 2). In the following, we call these renderers elementary renderers, by this meaning that the sole purpose of this specialized class is rendering a volume dataset without any side effects and that no meaningful further specialization of this type of renderer is possible.

In addition to these more obvious cases, generalization can be helpful for remote rendering in several ways. We use an abstract base class that we call remote client, which is only responsible for establishing a network connection to a remote server. While remote clients actually are renderers, remote servers own exactly one renderer. When the remote client issues an event over the network
connection, that event is passed to the renderer by the remote server, which reacts by e. g. generating a new image, which the remote server sends back over the network connection. Section 3.5 details concrete remote client specializations, which differ in the way the remote rendered image is displayed. Network communication between remote clients and remote servers is also subject to generalization, as outlined in section 3.4. Because remote clients encapsulate the network communication and implement the interface of an elementary renderer, they provide a convenient means to implement sort-last parallel rendering on a fairly high abstraction level.

Before rendering, all our renderers bind a render target. Render targets are a software abstraction that determines if the rendered result is directly output to the screen for display or stored in an offscreen buffer, which is useful if the result is meant to be sent over the network e. g. to a compositing server. With render targets, the accuracy of the compositing calculations for OpenGL based renderers can be influenced as well. By e. g. binding an offscreen buffer with 32 bit floating point precision per color channel, compositing is performed with floating point precision as well, which is impossible with OpenGL visuals that e. g. on consumer hardware typically only provide 8 bit precision per color channel.

We exploit generalization of renderer components for the sort-last parallel case by extending so called brick renderers. Brick renderers are abstract in terms of object orientation and organize the volume into cuboids using a k-d tree, which can be traversed using the visitor design pattern (cf. e. g. [23]). The actual volume data, which is located at the leaf nodes of the k-d tree, may be processed by any kind of renderer. Concrete implementations of brick renderers are the serial brick renderer and the parallel brick renderer. The serial brick renderer processes the leaf nodes one after each other. This is useful if the whole dataset does not fit into the video memory of the GPU. Then the volume is subdivided into convex objects (bricks) that fit into video memory and are transferred to the GPU for rendering sequentially [24]. The parallel brick renderer uses threads to render the bricks in parallel. When all threads finished rendering, the main thread performs sort-last alpha compositing using a visitor that is envoied during k-d tree traversal. This and the fact that brick renderers themselves use an arbitrary renderer to process the leaves yields three reasonable combinations:

- Sort-last parallel Multi-GPU mode, here the parallel brick renderer is combined with any kind of elementary renderer to process the k-d tree leaves. The OpenGL contexts of the rendering threads are configured so that different GPUs are accessed (e. g. by running on different X11 displays, which are themselves assigned to different GPUs through the X server configuration).
- Sort-last parallel distributed memory mode, instead of elementary renderers, the parallel brick renderer maintains a list of remote clients. By using threads, network communication is interleaved as well. The remote clients connect to remote servers that run elementary renderers.
- Combined mode, similar to the sort-last parallel distributed memory mode, but the remote servers themselves run parallel brick renderers. This combination is useful for sort-last parallel volume rendering on clusters of Multi-GPU computers.
3.4 Communication Between Renderers

Listing 1. Pairs of put / get functions must be overwritten that use the underlying network protocol to transfer volume rendering specific data items in order to specialize the I/O subsystem.

Remote clients need to communicate with the renderer of their assigned remote server e.g. over the network using a common inter-process communication (IPC) protocol. Having an extendable network communication protocol can be used to transparently migrating e.g. from commodity network technologies like gigabit ethernet to high-bandwidth, low-latency technologies like InfiniBand [25]. A general I/O subsystem as the one we propose (cf. figure 3) even makes it possible to transparently migrate from a distributed memory system to a shared memory system. The I/O subsystem is implemented in terms of obstacles known from the volume rendering domain. Remote clients request to transfer e.g. volumes, matrices or adjusted transfer functions to render servers and render servers answer appropriately e.g. by sending back rendered images (cf. Listing 1). The I/O subsystem then e.g. serializes the data to send it over a TCP/IP socket connection. With the communication means being completely transparent to the user, an I/O specialization can even be implemented to use a shared memory implementation for communication and thus avoiding latencies due to the network protocol if the remote client and server process are running on the same physical machine.

3.5 Display Clients

Display clients are specializations of the abstract remote client class and determine how remote rendered content is displayed on the screen of the client device. With display clients, we follow a highly versatile approach as well. Compressed image clients retrieve ordinary 2D image data for display using OpenGL. Data is sent from the remote server to the compressed image client either as JPEG images or compressed using run-length encoding (RLE). Our implementation is restricted to JPEG and RLE so far, although in general our system architecture could easily be extended e.g. by using the lossless PNG format as an alternative to JPEG.

The image-based client implements the remote rendering algorithm from [20]. By decoupling the display phase from rendering, with this approach frame rates of 30 Hz can be maintained throughout the rendering session, which makes this display mode especially suitable for virtual reality.

3.6 Resource Management

Zero configuration networking (ZCN) [26] can provide a convenient means to enhance an HPC system with additional resources at run time. We deploy a resource management software instance on a dedicated node of the HPC system to provide controlled access to the render server instances that are started on the nodes of the HPC system. This resource manager serves as a unique gateway to the system and provides the user with cluster resources transparently. Figure 4 depicts how resources and clients connect to the resource manager.

Nodes can register with the resource manager at any time using ZCN. The resource manager maintains available nodes as resources using a queue. ZCN on the one hand makes it easy for the nodes to announce that they are available e.g. after performing a rendering task. On the other hand, the HPC system can be enhanced with new resources at run time without having to restart the resource manager process and thus without incurring any downtime.

When the user issues a rendering request, the resource manager waits until enough resources to accommodate it are available. If the request can be handled by a single
Figure 4. Resources register with the resource manager subsystem using a zero configuration networking protocol. The resource manager establishes connections between remote servers and remote clients by pairing requests and resources to jobs.

resource, the resource manager pairs the request and the resource to form a rendering job, that is assigned to the allocated resource, which then creates a network connection to the remote client. Rendering events are negotiated by means of the network I/O subsystem described in section 3.4. Resource assignment is performed in a first-come, first-served fashion.

If the request requires to be handled by more than one resource, the communication protocol implemented on the remote client side will stay exactly the same. The only difference is, that the remote server resource will not create an elementary renderer, but rather a parallel brick renderer that is set up to perform sort-last parallel Multi-GPU rendering or distributed memory rendering as described in section 3.3.

Resources that finished processing a rendering request register with the resource manager again using ZCN. The system is tolerant to resources becoming unavailable due to hardware or software failure. During a rendering job, if the socket connection breaks, the client is informed that the resource became unavailable and can react appropriately. Because of the loose coupling of resource manager and resources, the resource management subsystem will not be affected by the deficient resource, which will simply not register with the resource manager until it is functioning again. When an idle resource becomes unavailable, this information is distributed over the ZCN protocol to the resource manager and the respective resource is removed from the resource list.

3.7 Integration into the DVR Library Virvo

The Virvo DVR library (cf. e. g. [27]) is part of the open source volume visualization software DeskVOX. Virvo is accompanied by a set of tools that are loosely coupled to the library. We extend this set of tools with an additional one called voxserver. Depending on configuration, voxserver may either serve as the resource management subsystem, or as a remote server that the Virvo library can connect to. Voxservers being configured as mere remote servers typically connect to a central instance of voxserver running in resource management mode. For a higher degree of flexibility, the Virvo library can also connect to a mere remote server, if resource management is unnecessary. While voxserver is responsible for the data distribution stage of the distributed volume rendering pipeline, the rendering, compositing and display stages are integrated into the Virvo library itself. Thus, if a visualization software uses Virvo for volume rendering, it can directly access the complete distributed volume rendering pipeline.

For the ZCN facilities provided by the resource management subsystem, we use Apple’s Bonjour protocol [28] on Microsoft Windows and Apple Mac OS X. On Linux, we can use the same protocol transparently by installing the software Avahi [29].

4 Discussion

An important trait of the Virvo library is its support for various elementary renderer types. This is reasonable because that way it is possible to support legacy GPUs that lack 3D texture support on the one hand, while on the other hand providing access to e. g. CUDA accelerated rendering, which makes it easy to implement ray casting as a more natural approach to sampling the volume rendering integral than 3D texture based slicing algorithms can be. Software ray casting leads to even higher versatility, so that Virvo, in addition to platforms with high-end and legacy GPUs supports hardware with no accelerated graphics at all. Because of this heterogeneity it seemed out of the question to prepare only one of the elementary renderers to support sort-last parallelism. We thus seeked for a software design that was able to naturally scale all the existing renderers as well as renderers potentially added to the library in the future to multiple processors without having to alter the way they function internally. By superimposing a renderer maintaining the k-d tree search structure as well as thread handling for parallelism on top of the elementary renderers, this task is clearly separated from the mere rendering task that the elementary renderers perform. Versatility regarding parallel rendering could even be improved by exposing the visibility sorting data structure to be replaced if desired. We do not follow this approach because we believe that the choice the visibility sorting data structure won’t affect rendering speed that much as it would e. g. with ray tracing, where millions of graphics primitives are stored in such data structures in order to accelerate ray traversal [30].
Extending the distributed volume rendering pipeline with a dedicated stage for the display phase provides a means to adapt to various data exploration scenarios. With virtual reality applications, interactive rendering is a necessity because interceptions, especially if the virtual reality system supports head tracking, can cause fatigue and nausea. If DVR is deployed to monitor intermediate results of a time-consuming simulation on an HPC system, the researcher conducting the simulation can probably cope with only a few new frames per second. Because of this heterogeneity, we find it important to dedicate a separate stage of our distributed volume rendering pipeline to the display phase.

Volume rendering is an interactive task by nature, so that a traditional resource management system designed to function in a batch fashion would be tedious to adapt to our needs. Designing a dedicated resource management system made it easy to incorporate the design choices best suiting our demands. Having a handcrafted resource management system allows for the implementation of choices based on features specifically devoted to the visualization domain. A suitable node for a specific rendering request in that case can e.g. be selected based on the amount of available and used video memory or based on its CUDA compute capability, quantities that probably would not be exposed by a traditional resource management system. On the other hand, this lack of generality results in a major shortcoming of our approach, since our resource management system can only accommodate direct volume rendering and would have to be adapted to support even an alternative visualization method.

Zero configuration networking is convenient because it can help to significantly decrease system downtime. With no need to reconfigure and restart the system when removing or adding new resources, users do not even have to resend their rendering requests when the system is altered. On the other hand, ZCN can provide a convenient load balancing means, since resources can be dynamically added to the HPC system if there are more render requests in the queue than the system can handle at once. We know of no software solution for direct volume rendering that uses ZCN to such an extent.

5 Conclusion and Future Work

We presented a new pipeline approach to distributed volume rendering. Our distributed software architecture is highly versatile by supporting distributed remote rendering access in a multi-user context to parallel distributed memory and Multi-GPU resources. Our system scales to heterogeneous hardware platforms by supporting various kinds of HPC systems ranging from render clusters with Multi-GPU support to large distributed memory systems with no graphics accelerators at all. We adopt an existing pipeline approach to distributed volume rendering and extend it with multi-user support and a dedicated stage to accommodate the display phase. An interactive resource management subsystem for interactive use enables us to make use of zero configuration networking to dynamically allocate new resources. The system does not need to be stopped and users do not even have to resend their rendering requests.

In the future, we would like to extend our system to not just support direct volume rendering exclusively. Especially the resource management subsystem can be decoupled from the rendering system with not too much effort. We so far use ZCN only to reallocate resources at run time to accommodate new rendering requests. A powerful extension to our approach would be to dynamically reallocate resources to existing rendering jobs at run time, which could even be exploited to implement dynamic load balancing schemes.

References


