Interactive, Internet Delivery of Visualization via Structured, Prerendered Multiresolution Imagery

Jerry Chen, Ilmi Yoon, *Member, IEEE* and E. Wes Bethel, *Member, IEEE*

Abstract—
We present a novel approach for latency-tolerant delivery of visualization and rendering results where client-side frame rate display performance is independent of source dataset size, image size, visualization technique or rendering complexity. Our approach delivers pre-rendered, multiresolution images to a remote user as they navigate through different viewpoints, visualization parameters or rendering parameters. We employ demand-driven tiled, multiresolution image streaming and prefetching to efficiently utilize available bandwidth while providing the maximum resolution a user can perceive from a given viewpoint. Since image data is the only input to our system, our approach is generally applicable to all visualization and graphics rendering applications capable of generating image files in an ordered fashion. In our implementation, a normal web server provides on-demand images to a remote custom client application, which uses client-pull to obtain and cache only those images required to fulfill the interaction needs. The main contributions of this work are: (1) an architecture for latency-tolerant, remote delivery of precomputed imagery suitable for use with any visualization or rendering application capable of producing images in an ordered fashion; (2) a performance study showing the impact of diverse network environments and different tunable system parameters on end-to-end system performance in terms of deliverable frames per second.

Index Terms— **remote visualization, remote rendering, internet media delivery, multiresolution digital media**

I. Introduction

It is well accepted that interactive visual data exploration is an effective means to facilitate data understanding [27]. Typically, output from visualization and graphics applications consists of a set of images that result from changes in visualization and rendering parameters. Generally speaking, we can characterize the resulting image set as the result of an exploration of visualization or rendering parameter space.

The work in this paper describes a novel approach for delivering the interactive exploration experience to users. This new approach is appropriate for many, but not all, visualization and graphics use modalities. In particular, it provides what appears to the consumer to be an interactive data exploration experience in the form of semi-constrained navigation through visualization or rendering parameter space but without the cost associated with learning how to execute or use a potentially complex application on a remote computing platform. It is a general purpose solution that is widely applicable to any application that produces visual output. It offers capabilities that overcome limitations of previous similar approaches making it suitable for use in delivering an interactive visual data exploration experience to a potentially large body of consumers.

Manuscript received December 2006, revised April 2007.

J. Chen is with Yahoo Inc., Sunnyvale CA, jerryc@yahoo-inc.com
Yoon is with San Francisco State University, yoon@cs.sfsu.edu
Bethel is with Lawrence Berkeley National Laboratory, ewbethel@lbl.gov

A. Background and Motivation

In [5], Bergeron describes three broad user-centric visualization use modalities. “Presentation visualization” is where you know what is there and want to show it to someone else. “Analytical visualization” is where you know what you are looking for. “Discovery visualization” occurs when you have no idea what you’re looking for. Discovery visualization is characterized as an “undirected search,” or “unconstrained navigation” through visualization or rendering parameter space.

Perhaps the most practical way to implement discovery visualization is to have the consumer/user actually execute the application and interact with it to perform unconstrained navigation through an $n-$dimensional visualization and rendering parameter space. It would be a nearly intractable problem to precompute and store images that span these n dimensions for later exploration.

The other two use modes – analytical and presentation visualization – do not require full, unconstrained navigation through an $n-$dimensional parameter space. In these modes, and depending of course on the ultimate application, the size of n will probably be small and the sampling of each of these dimensions will be limited to a meaningful range of values. In these modes, it becomes feasible to compute and store the images produced by the visualization and rendering process for later exploration by a set of consumers.

There are a number of benefits associated with this approach. First, consumers are relieved of the burden of learning to launch and interact with potentially complex applications. Second, the potentially high cost of visualization and rendering is amortized across a potentially large number of consumers. Third, we can facilitate trivial access to the resulting images so that a consumer need not ever be faced with a command-line prompt, nor have to obtain a login at a central computing facility. Fourth, it is possible through a unique implementation, such as the one we describe here, to provide the experience of interactive data exploration through a multidimensional parameter space. Fifth, it offers an exciting new set of possibilities for presentation visualization, which typically takes the form of static images, one-dimensional movies (MPEG) or interactive movies with significant restrictions (QuickTime). In our experience working with scientific users [17], we have found such benefits to be of high value and interest to scientific users; discovery visualization plays a key, but not dominant, role in the scientific process, and ease-of-use issues are a significant barrier to use of visualization technology. After an initial phase where discovery visualization plays a key role, scientific users tend to spend much more time in analytical and presentation visualization.

B. Our Approach and Contribution

The work we present here describes a methodology for preparing and for efficiently and effectively delivering multidimensional visualization and rendering results – sets of images – to a remote consumer. The primary motivation and context for this work stems from the desire to overcome a number of difficulties characteristic of remote and distributed visualization while providing the benefits associated with interactive exploration of multidimensional visual results. Our aim is to provide the opportunity for such benefits without the burdens typically associated with forcing users to run remote and distributed applications to perform interactive visual data exploration.

The fundamental idea behind our approach is as follows. First, a rendering application generates a set of images in a structured and ordered fashion, perhaps by varying the view position, one or more visualization parameters, temporal slice of data, and so forth. A collection of structured images might contain views of a 3D object where the viewpoint is moved along regular lines of latitude and longitude about the object. Second, a preprocessing step we refer to as "encoding" prepares the images for transmission to and consumption by the client. Third, a client application requests and displays the precomputed images at the user's pleasure. For example, the user may navigate from view to view to inspect a 3D object from any precomputed view. Since input to our system consists only of images, this method of encoding, delivery and user interaction with content is generally applicable to any interactive visual domain. This type of approach is not new – it is the basis of several different types of interactive digital media, including MPEG, QuickTime and QuickTime VR [7] movies. Figure 1 shows a complete, end-to-end view of the system.

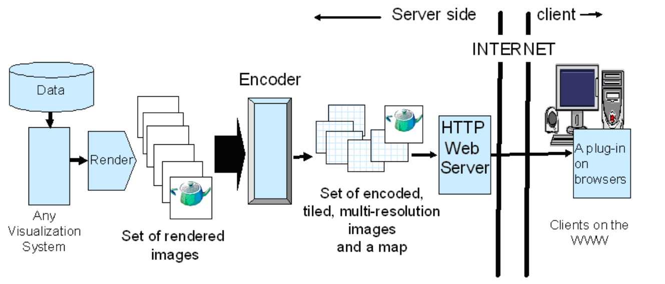


Fig. 1. An end-to-end view of our image delivery implementation.

The new contributions of our work are as follows. First, we employ multiresolution imagery and client-side view-dependent resolution selection to overcome the fixed image resolution typically associated with digital media formats and user experiences. Second, our approach offers the ability to perform $n-$dimensional attribute browsing; in contrast, MPEG-1 and MPEG-2 offer one-dimensional browsing while QuickTime VR offers navigation through only three dimensions. Third, our implementation has a relatively small, fixed-size memory footprint making it suitable for use on a wide range of platforms. Fourth, we use a prefetching algorithm to minimize display latency and overlap I/O with display operations. Fifth, we present performance studies: (1) evaluating the impact of tunable system parameters on storage requirements and end-to-end performance; and (2) reporting the performance improvement that result from prefetching, which effectively overlaps I/O and display processing.

We begin in Section II with a discussion of background topics germane to our work. Next, we present in Section III the architecture of our approach including all requisite preprocessing steps, client application design and implementation, and tunable system parameters like image tile size, prefetching and caching. In Section IV we present a performance study that describes server-side storage and client-side performance characteristics under a variety of network and tunable parameter configurations. We conclude with discussion and suggestions for future work in Section V.

II. Background and Previous Work

A. Remote and Distributed Visualization

The term “remote and distributed visualization” refers to a mapping of visualization pipeline components onto distributed resources. Remote visualization is motivated by the reality that users need to perform analysis of data that is too large to move to their local workstation or cluster, or that exceeds the capacity of their local resources to process.

From a high level, there are three fundamental types of bulk payload data that move between components of the visualization pipeline: “scientific data,” visualization results (geometry and renderable objects), and image data ([4], [25]). In some instances and applications, the portion of the pipeline that moves raw data between components is further resolved to distinguish between “raw” and “filtered data” ([29], [19], [18]). For simplicity and without loss of generality, we refer to these three partitioning strategies as “send data,” “send geometry” and “send images.”

The “send data” partitioning aims to move data from server to client for visualization and rendering. Optimizing this portion of the pipeline can take several different forms. One is to optimize use of distributed network data caches and replicas so a request for data goes to a “nearby” rather than “distant” source, as well as to leverage high performance protocols more suitable for bulk data transfers than TCP [2]. Other optimizations leverage data subsetting, filtering and progressive transmission from remote sources to reduce the level of data payload crossing the network ([18], [28]).

In a “send geometry” partitioning, data I/O and visualization algorithms run on a server with resulting “renderable geometry” sent to a client for rendering and display. One way to optimize this path is to send only those geometric primitives that lie within a view frustum; such optimizations have proven useful in network-based walkthroughs of large and complex data [8]. Frustum culling, when combined with occlusion culling and level-of-detail selection at the geometric model level, can result in reduced transmission payload [23].

In a “send images” partitioning, all processing needed to compute a final image is performed on a server, then the resulting image data is transmitted to a client. Over the years, there have been several different approaches to implement this partitioning strategy. Virtual Network Computing (VNC) [30] uses a client-server model: the custom client viewer on the local machine intercepts events and sends them to the VNC server running at the remote location; the server detects screen updates, packages and sends them to the client for display. OpenGL Vizserver [33] and VirtualGL [10] use a client-server model for remote delivery of hardware-accelerated rendering, but without access to the entire remote desktop. Some visualization applications, e.g., VisIt [24] and ParaView [22], support a “send images” mode of operation where a scalable visualization and rendering back-end provides images to a remote client. In these approaches, a remote client requests a new frame; the server-side application responds by executing the local part of the visualization pipeline to produce and transmit a new image.

Several works in the “send images” theme are more closely aligned with our work here. Engel’s system provides an image-streaming codec suitable for use with OpenInventor applications that communicate using a multicast model to a group of clients that may have vastly different connection characteristics [12]. GVid [15] is the video streaming module of the Grid Visualization Kernel [20]. As middleware, it enables encoding, transmission and display of on-demand images produced by GLUT applications, including GLUT-based visualization applications. More recently, Ellsworth [11] generates MPEG-encoded movies directly from running climate simulations for immediate consumption and sharing by climate scientists.

Our approach offers distinct advantages over these previous works. First is the notion of generality: any application that produces visual output can provide source images for delivery to remote consumers. Second, our delivery mechanism allows for multiresolution image browsing, something not supported by existing digital media formats. Third, once the visual output has been produced, the results can be reused across a potentially large audience (this benefit is shared with some previous works, e.g., [11]). Related, consumers need not be experts in setting up and running potentially complex distributed applications.

B. Image-Based Rendering for Remote Visualization and Interaction

Image-based rendering (IBR) refers to the process of rendering a new frame from existing frames rather than from scene content. IBR rendering speed for incremental frames is independent of the complexity or size of the scene and the quality of rendered images. IBR approaches appear useful for remote scientific visualization: scientific datasets currently are commonly in the range of 10s of TB and growing. In contrast, the storage requirements for collections of presentation and analytical visualization images is likely to be much smaller and will incur a much smaller cost to deliver to a large user community.

The notion of accelerating remote visualization via IBR techniques is not new. Image-based rendering acceleration and compression (IBRAC) extracts temporal coherence between frames and the server sends only the difference between the frames [35]. Visapult [3], Semotus Visum [25] and an optimized GVK [23] all employ IBR or IBR-like methods to accelerate remote visualization.

QuickTime VR [7] provides interactive navigation through pre-rendered images. The two types of QTVR – panorama and object movies – each support a different type of user navigation. QTVR object movies, which are more closely related to our work, consist of up to three dimensions, or arrays, of images. Whereas panorama movie images typically represent views from a fixed viewpoint to different azimuthal angles, object movie images typically represent viewpoints from different "latitude and longitude" positions, but with a common "look at" point, as well as zoom-in and zoom-out. The QTVR player responds to user input by selecting the correct image to display from up to three dimensions of images. While not strictly IBR, QTVR deals only in image data and presents new image data in response to user-induced viewpoint changes.

An earlier version of the work we present here leverages the QTVR object movie concept to implement 3D, time varying navigation of scientific visualization results [6]. Here, frames were generated by a visualization application using prescribed viewpoints, then encoded as a QTVR object movie. A user obtains the movie using a web browser, then interacts with the movie using a "garden variety" QTVR player.

Our present work overcomes two fundamental limitations of our previous approach. The first is a resource consumption problem: QTVR object movies can become quite large and players will download the entire movie into main memory. In some cases, no navigation is possible until the entire movie has been downloaded. The second limitation is one of fixed image resolution. When the client zooms in for a closer view, the zoom is accomplished by image scaling. The result is either degraded visual quality if the movie is comprised of low-resolution images, or inefficient use of resources if the movie consists of high-resolution images.

C. Remote Visualization Using Image Tiling and Multiresolution Streaming

Tiling, streaming and multiresolution levels are common remediation strategies for the resource utilization problems that can result when using fixed, high-resolution imagery. The basic premise is that tiling and multiresolution delivery is an approach that effectively balances bandwidth with visual perception requirements. Since high-resolution images are not distinguishable from low-resolution ones when viewed from a distance, it is a waste of resources to send and display a high-resolution image when a low-resolution one will suffice. Here, we want to use available bandwidth as efficiently as possible to deliver the maximum resolution that a user can perceive from a given viewpoint. For zoomed-in views, high-resolution images are subdivided into tiles so only viewable tiles are downloaded as needed. Several commercial products utilize these ideas to provide online browsing capability for very high-resolution 2D images, but without the cost of downloading the entire, full-resolution image to the client. Zoomify [36] and Google Map [14] use these concepts to support 2D image navigation. In contrast, our implementation provides the ability to navigate through n dimensions of multiresolution image data to provide a rich user navigation experience better suited for many types of remote scientific visualization.

NASA's World Wind [26] and Google Earth [13] are remote visualization tools that implement 3D earth browsing. Google Earth shows impressive 3D navigation through streamed satellite images demonstrating the effectiveness of simulated 3D exploration with 2D image sets. It combines many different kinds of observed and geoinformational data on a 3D sphere, including satellite images, political borders, city names, animation of some earth events, ground geometry, and so on. The Google Earth client will dynamically download data according to the client's view parameters and event selection criteria. While these projects focus exclusively on earth browsing and related visualization on top of the geographic maps, the concepts are generally applicable to image delivery applications.

III. Architecture

Our system takes as input prerendered images or photographs that represent different viewpoints, resolutions and time steps from any kind of application that produces visual output, including image-capture devices. The images are then preprocessed by an encoder application that generates a new set of multiresolution image tiles that will be streamed and pre-fetched for efficient bandwidth utilization. In addition to images, the encoder creates two metadata files called "map file" and "catalog file" that contain encoding and streaming information. The catalog file contains the information necessary to support the map file. Contents for both types of files are described more fully in Section III-B. Source images are retiled as requested, and all tiles and metadata files are then placed into a publicly accessible directory on a web server. The custom client for our system, described in Section III-C, begins by downloading the map file. Later, as the user begins navigation through an $n-$dimensional space, the client requests new images from the server to fulfill the user's desired viewpoint. The client uses image prefetching, which is described in Section III-D, to accelerate performance by hiding latency.

A. Definitions

We use the terms "view" and "pan" synonymously to refer to an image, or set of tiles, associated with a particular point in $n-$dimensional browsing space. Our implementation and the definitions that follow reflect a bias towards reproducing the familiar QTVR object movie functionality. A sequence of "horizontal pans" defines an orbit about an object across a set of azimuthal angles at a constant latitude with the view pointing at the center of the scene. Multiple horizontal orbits at different "latitudes" are referred to as "vertical pans." There may be more than one time step per view, but there must be the same number of time steps for all views. While our use of the term "pan" is divergent from the traditional cinematographic definition, it is consistent with the definition present in the QTVR documentation, and we adopt its use here.

While QTVR movies are limited to three dimensions of source imagery, in our system, a user may navigate, or browse views, in an $n-$dimensional "view space." Two of these dimensions typically correspond to changes in horizontal and vertical pan positions. A third corresponds to multiresolution image level and is associated with changes in zoom level. A fourth is associated with either temporally varying images or changes in a visualization parameter. In principle, there is no limit to the number of browsable dimensions. In practice, storage requirements grow exponentially with n, so n will likely be small in most instances.

A "tile" is a subdivided source image. As part of preparing images for delivery to the client, we decompose large, high-resolution images into tiles. The tile size is a tunable system parameter that can have a dramatic effect on storage and performance requirements. In Section IV-B, we present experimental results that characterize the relationship between tile system and performance.

Views may be comprised of a set of multiresolution tiles. All views must have the same depth of multiresolution. In Figure 2, the red box corresponds to the subset of four image tiles at a particular view and at some multiresolution level that is currently being displayed to a user via the client. The "visible area" inside the red box is comprised of subsets of four larger image tiles. We refer to the complete set of tiles for this view as the "available area."

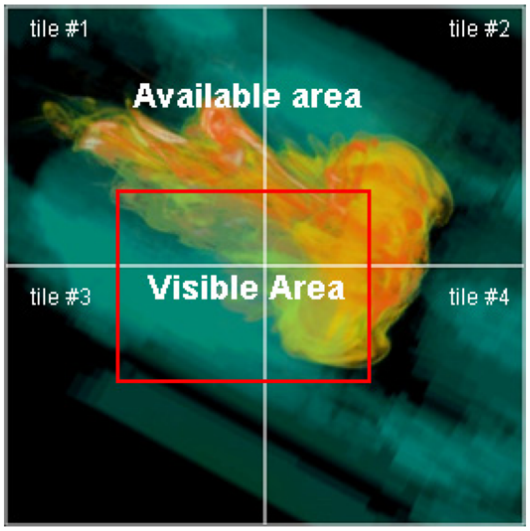


Fig. 2. The "visible area" (red box) is the portion of a tiled scene view that is being displayed to the user at the client. The visible area is drawn from a potentially larger set of images, the "available area," which comprises all image tiles for a given scene at some point in n-dimensional display space.

B. Preprocessing

Preprocessing consists of two stages. The first is to create the multiresolution frames and the second is to create tiles from the multiresolution frames and metadata files describing tile layout, location and so forth. Any application that can generate high-resolution image files corresponding to prescribed views is a suitable source of images. When multiresolution imagery is needed, the source rendering application must produce images at different resolutions. During this first stage of preprocessing, each individual image corresponds to a single point in the $n-$dimensional view space. By "each individual image" we also mean the multiple resolutions of a single source image corresponding to some position in an $n-$dimensional view space.

The second preprocessing step consists of two substeps; first, create image tiles from individual source images and then second, generate metadata for the complete collection of images. We refer to this second preprocessing step as "encoding." Our encoder first subdivides an individual view image into a set of view tiles. The tile size is a tunable parameter. Its value has an impact on server-side storage requirements (see Section IV-B) and download speed (see Section IV-C). The encoder generates an XML "map file" that contains metadata describing the set of tiles. The client later uses the map file to determine the location of the image tiles. The encoder also creates a "catalogue file" that contains URLs of map files. The catalogue file is a useful mechanism for organizing collections of related movies. At the end of the preprocessing step, the map file and image tiles are placed into a location accessible to the web server for later distribution to remote consumers.

In our performance experiment, we did not collect data measuring the runtime of the encoding step. Conceptually, the cost of this one-time operation is amortized over many uses of the resulting collection of images. Conceptually, the cost of this step is low: the source images are read into memory, then written back out to disk in tiled form. The runtime and storage complexity of this step is linear with respect to the number of source images.

C. Client

The client application parses the XML map file to set up a GUI, then fetches and displays images in response to user input events. User input events that correspond to navigation through an $n-$dimensional view space cause the client to download and display new images from the server. This client-pull model results in a great deal of implementation flexibility as any web server can be used to deliver images. Additionally, a single web server can service multiple simultaneous clients. No server-side communication or back-end processing code is required for this implementation.

The client supports a multithreaded pull model so that multiple tiles may be requested at once. Each worker thread opens a connection back to the server for requesting images. These connections remain open over the lifetime of the worker thread so that they can be reused for subsequent image requests and thus avoid the expense of opening and closing remote network connections. Later in Section IV-C, we show results that indicate that the multithreaded I/O approach results in substantial performance gains when used on high-latency network connections.

1) Client-side Memory Management.: When the client connects, it requests and downloads the catalogue file, then constructs a tree of movies available at the server side. After the user selects one of the movies, the client will download the corresponding map file and use its metadata to create a map structure in memory.

The client's map structure contains an array of pan information; each pan has an array of tile information. From the client's view, individual tiles have one of four memory-resident classifications: "at server," "downloading," "in memory cache," or "in disk cache." In order to access the "closest copy" of the tiles, the client must track the current location of each tile image, and record the filename and point to the address of the tile in the client's memory cache. When a tile needs to be loaded for display, the "closest" tile is selected: "in memory cache" is closer than "at server" or "in disk cache."

The client-downloaded tiles are cached into memory and ordered in a priority queue. The queue is ordered using a least-recently used (LRU) strategy: tiles recently viewed have higher priority in the queue, while tiles not recently viewed have lower priority. When the user "navigates away from" the currently displayed tile, the client will alter the tile's relative priority in the queue as other tiles are more recently viewed. When the size of the queue reaches its capacity, the least recently used tiles will be removed from the priority queue to make room for new, incoming tiles. For multiresolution image sets, low-resolution images are loaded first, then refined with higher-resolution images when needed to satisfy a given viewpoint: low-resolution images have higher priority than higher-resolution images. The memory cache size – queue length – is a tunable client parameter that trades client-side memory consumption against processing speed.

In addition to a memory cache, the client has an optional disk cache where tiles are cached to the local file system. This secondary cache is similar to the familiar disk cache used by web browsers. It serves to improve performance by eliminating the need to download previously obtained images from the remote server in the current or subsequent client sessions.

2) Client Prefetching.: By utilizing prefetching, a client pulls tiles from the server according to the current and predicted view position, time step, or other browsable attribute, and then caches them into local memory. Such prefetching can overlap with other application operations, like rendering. However, prefetching takes a lower priority than an image download needed to satisfy the current viewpoint. Prefetching can have a substantial positive impact on client-side performance. Prefetching design is described in Section III-D, and its impact on performance is presented in Section IV-E.

3) Client Implementation.: The present client implementation is in C++ and makes use of JNI (Java Native Interface) and JNLP (Java Network Launching Protocol) technologies to launch the program remotely by clicking a web link. A "jnlp" file contains the information about our package for Java Web Start to download and launch on any platform. At present, we have built and tested the client on Windows platforms. The client uses OpenGL to display image data using texture mapping, and uses texture coordinate transformations to implement zoom and translation operations.

D. Prefetching Design

Prefetching is known to improve performance and usability in many application areas. Prefetching in sequential, streaming audio and video applications is straightforward since little, if any, prediction is required to determine the next block of data the application will need. In our application, where a user may navigate freely through an $n-$dimensional space, the navigation direction is not known in advance, but must be predicted based upon usage patterns and heuristics.

In our implementation, there are five degrees of freedom (DOFs) for user navigation: four view transformations and one for "data browsing." The four view transformations are: left/right/up/down translation, zoom in/out, azimuthal rotation (horizontal pan) and polar angle rotation (vertical pan). We use the "data browsing" degree of freedom for either temporally varying image data, as in the case of a time-evolving dataset, or for a visualization parameter change, such as isocontour level.

Figure 3 is a graph representation of the client's internal state engine. During execution, the client transitions between one of three possible states in response to user input and completion of work tasks. Nodes in the state graph represent a specific client state; edges represent a state change. The edges in Figure 3 are labeled with the event that induces a state change. The client enters the prefetch state only when "idle."

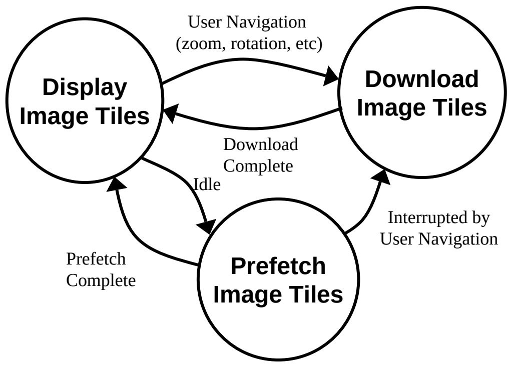
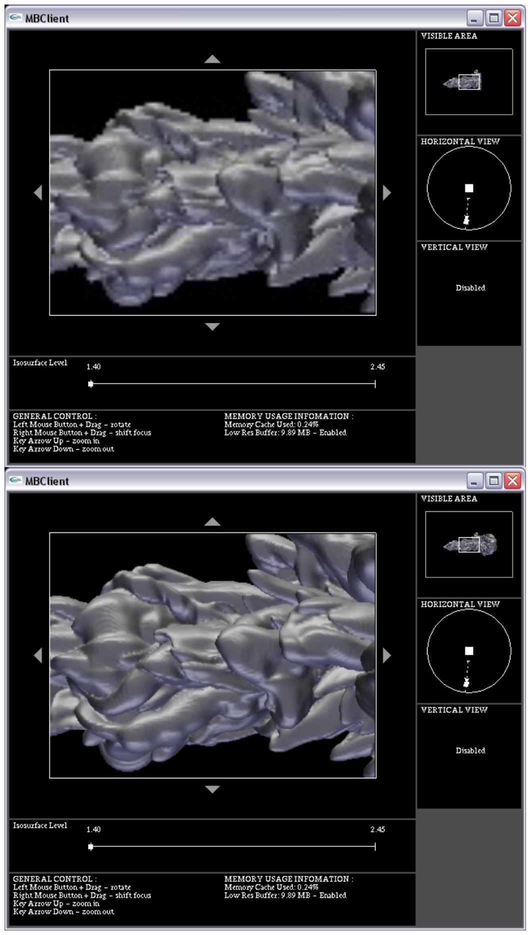


Fig. 3. The Client's internal state diagram. Nodes represent client state, while edges show the conditions producing a change of state. Prescribed image tile downloads – those needed to satisfy a known viewpoint – have priority over prefetching in this arrangement.

Some view change operations, namely zoom and translation, do not always require a new set of images. In cases where new images are not required, the client implements the change in view via texture coordinate transformations. Translating across a tile boundary, or zooming in past a critical resolution threshold, will trigger an image prefetch operation.

Our current client implementation switches between image resolution levels in a discrete fashion, which can result in visual artifacts as shown in Figure 4. That is, we do not attempt to "blend" between source images of different resolution as the user changes zoom levels. Performer uses pixel-wise blending of renderings of models from different levels-of-detail (LOD) to visually smooth such transitions [31]. Adding such capability to our client would make for interesting future work.



GENERAL CONTROL :
Left Mouse Button + Drag – rotate
Right Mouse Button + Drag – shift focus
Key Arrow Up – zoom in
Key Arrow Down – zoom out

MEMORY USAGE INFOMATION :
Memory Cache Used: 0.24%
Low Res Buffer: 9.89 MB – Enabled

Fig. 4. These two images are screen shots of the client displaying isosurface visualization image sequences at different zoom levels. In the top image, we have zoomed in up to the threshold where lower-resolution images are used as the source. The visual artifacts resulting from zooming in on low-resolution images are clearly visible. In the bottom image, we have zoomed in one more step – just past the threshold where the next higher-resolution images are chosen by the client for display. This dynamic approach to multiresolution image selection and display approach is applicable to a broad range of potential application areas.

Other DOFs – rotation and changes in time or visualization parameter – will always require a completely new set of images. To minimize display latency that would result from blocking while downloading images, the client uses a more aggressive prefetching strategy for rotation. It will prefetch images in each of the four possible directions of rotation: plus/minus azimuth and plus/minus polar angle. The client initiates such prefetching after the current view is loaded. The most recent direction of rotation is set to be the priority in prefetching order: the tiles in the same direction as the most recent rotation direction will be downloaded first.

Our prefetching algorithm has two tunable parameters:

prefetch depth (how many "views ahead") and prefetch ratio (how many tiles for each view). The number of image tiles a client will prefetch is a function of both of these parameters. For example, when performing rotation operations, if the prefetch ratio is set to 50%, then the client will prefetch half the tiles needed to display the current view. If the prefetch depth is set to a value of three in this case, the client will prefetch three view's worth of images for each of the four possible directions of rotation. In Section IV-E, we measure the impact of prefetching on client performance.

IV. Performance Characterization

Our application's performance on both the server and client side is a combination of several different factors and processing stages, some of which are tunable. In this section, we discuss the impact on server-side storage and client-side performance of these tunable system parameters.

Server-side work consists of encoding images as a preprocessing step and then delivering images to the client. We described the processing steps earlier in Section III-B. Since preprocessing is performed once per collection of images and, in our implementation, is not associated with interactive delivery of images to the client, we do not consider its impact on client-side performance. Another aspect of server-side performance is responding to client requests for images. Our implementation uses a standard web server for this purpose: the client asks for images using HTTP GET requests. The web server responds by sending the client-requested image. This request-respond conversation occurs over a TCP connection. The bandwidth limitations of TCP as a communication protocol are well understood. Previous efforts like GridFTP [1] overcome single-stream TCP performance through connection striping. Our multithreaded client implements a form of TCP striping (see Sections III-C and IV-C) to improve performance.

We report client-side performance in terms of "frame rate," or frames per second. In this context, client-side frames/second reflects the sum of: (1) the time needed for a client to decide which image it needs to satisfy a given view; (2) the time needed to obtain the image – this time will be less if the image is in the client's cache and will be higher if the client must request the image from the server and await its arrival; (3) the time required to display the image. Client-side runtime performance is dominated by (2), whereas the cost of (1) and (3) is negligible.

The remainder of this section is organized as follows. We begin with an enumeration of experiment resources in Section IV-A. Tile size impact on server-side storage requirements appears in Section IV-B, on single- and multi-threaded client performance in Section IV-C. Section IV-D presents performance data reflecting the interplay between user navigation and multiresolution image levels. Image prefetching's positive impact on client-side performance is reported in Section IV-E.

	Throughput (Mb/s)	Latency (ms)
100BT LAN	93.75	0.1
Limited bandwidth fiber	5.86	22.0
Yahoo DSL	1.25	16.0

TABLE I
NETWORKS AND PERFORMANCE CHARACTERISTICS.

A. Experiment Resources

In all the performance experiments that follow, the server and client computer systems remain constant while we vary different tunable parameters and networks. For the server, we use a desktop class machine consisting of a single AMD Athlon 1400 CPU with 512MB of memory running Apache 2.0.49 under SuSE Linux 9.2. For the client, we use a laptop consisting of a 2.66GHz P4 CPU, 448 MB RAM and RADEON IGP 345M display adapter that shares 64 MB system memory running Windows XP. Both server and client machines are equipped with 100BT Ethernet adapters.

For the source images, we created several different image sequences from scenes having content representative of many typical applications. One sequence consists of a ray-traced scene containing two popular mesh models (a Buddha and a dragon). The others are commonly used forms of scientific visualization: direct volume rendering, isosurface rendering and a ball-and-stick molecular rendering. For the scientific visualizations, we generated images at resolutions of 400x300, 1600x1200 and 4000x3000. For the ray-traced scenes, we used Pixie (pixie.sf.net) – an open source photorealistic renderer with a Renderman-like interface – to create images at 2000x2000 resolution and used image downsampling to construct lower-resolution images. For the scientific visualization renderings, we used OpenRM Scene Graph [9], which includes both visualization and parallel rendering capabilities. All images are stored in JPEG format with a relatively high level of quality (low level of loss). As seen in Figure 5, the size of a JPEG-compressed file is a function of image content; we include data from several different source applications for the sake of completeness.

Our performance experiments use three different networks representative of those available to most users. Throughput and latency measurements, obtained using Netperf [34], are shown in Table I.

B. Tile Size Impact on Server-Side Storage Requirements

The tile size, which is a tuneable parameter set during the image encoding phase, has a potentially significant impact on performance and storage resource requirements. Our experiments show evidence of the classic trade-off between speed and storage. Looking at Figure 2, we see that the portions of the image tiles contained within the Available Area, but lying out the Visible Area, are effectively "wasted" in the sense that those pixels must be transmitted to the client, yet don't contribute to the visible part of the scene. We would expect that smaller tile sizes would result in better overall frame rate at the client: the amount of "wasted pixels" decreases as tile sizes grow smaller. A smaller tile size results in an increase in the number of tiles for each frame.

As shown in Figure 5, the space and transmission gains one realizes from JPEG compression drop as the tile size decreases due to two factors. First, each JPEG file has a header – increasing the number of tiles per frame results in more JPEG header information being transmitted over the network than would be the case with larger tile sizes. While the size of the JPEG header is variable, in the case of these test images, the header average size is about 226 bytes across all tile sizes and image sequences. Second, JPEG compression likely becomes less effective with decreasing tile size. In other words, we would not necessarily expect the size of a JPEG-compressed image to be the same as the sum of sizes of JPEG-compressed tiles of the same image. An additional factor influencing client-side performance with smaller tile sizes is a larger number of image requests from the client. While our implementation uses persistent TCP connections (to avoid the cost of setting up and tearing down a socket for each image request), it is cheaper to request a single image of N bytes than to request M images of N/M bytes due to increased work at the server (e.g., `fopen()`). One of the objectives of our performance experiment is to capture the net result of these trade-offs with a single, client-side frames-per-second number.

C. Tile Size and Threading Impact on Download Speed

Generally speaking, it is well accepted that TCP-based communication performance can be improved through a combination of protocol tuning [21] and "connection parallelism" ([32], [16]). We ran a set of tests to understand the performance gain that would result when varying two primary parameters – connection parallelism and tile size – over three different but typical categories of networks. We expect an increase in connection parallelism to produce a greater throughput over a TCP-based network

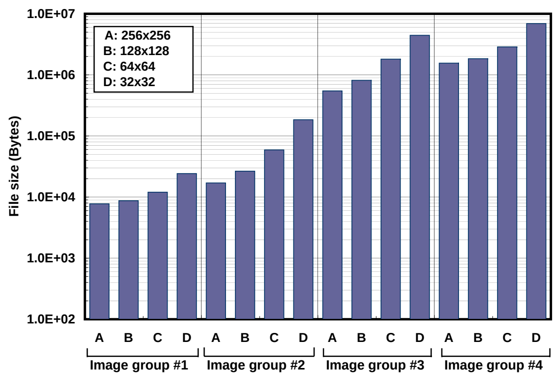


Fig. 5. This graph shows the relationship between tile size and storage requirements at the server over several different image sequences. In all cases, decreasing the tile size results in an increase in server-side storage requirements. In some cases, the difference in storage requirements between 256x256 and 32x32 tiles may be as much as about one order of magnitude.

link; such an improvement translates into a better user experience as measured by client-side frame rate.

We tested several different configurations of tile size and connection parallelism over three different networks. The results, shown in Figure 6, indicate the relation between number of threads, tile size and performance on different types of connection. For all of the tests in this battery, we had the client cycle through approximately 100 viewpoint rotation steps in an ordered, sequential-step fashion using a medium-resolution isosurface image set as the data source. With this approach, the client frame requests are deterministic and repeatable regardless of the number of client-side threads.

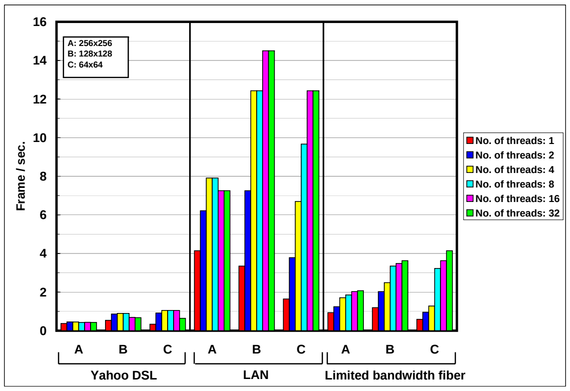


Fig. 6. This chart shows client performance – reported as frames/second – while varying two independent variables over the three test networks. One independent variable is tile size and the other is the number of client image-download threads. From these results, we observe: (1) a tile size of 128x128 gives the best overall performance across the three test networks; (2) the client performance improvement resulting from increasing TCP parallelism is more pronounced on higher-bandwidth networks.

As we begin to increase the number of client-side threads, we see a performance increase in nearly all cases up to a limit of about sixteen threads. These results appear to be consistent with prior work aimed at improving TCP throughput via striped connections [16]. Our testing methodology did not include the option of exploiting parallel architectures to determine the degree to which asymptotic network performance is correlated to use of uniprocessor machines. Figure 6 shows that 128x128 appears to be the optimal tile size for the test conditions in this study.

D. Impact of User Interaction and Multiresolution Image Levels on Performance

There is a complex interplay between potential client-side navigation patterns through an $n-$dimensional view space and the ultimate impact on system performance. A common navigation pattern is the context/focus model, where where a user begins with a zoomed out view to establish context, then zooms in to focus on detail.

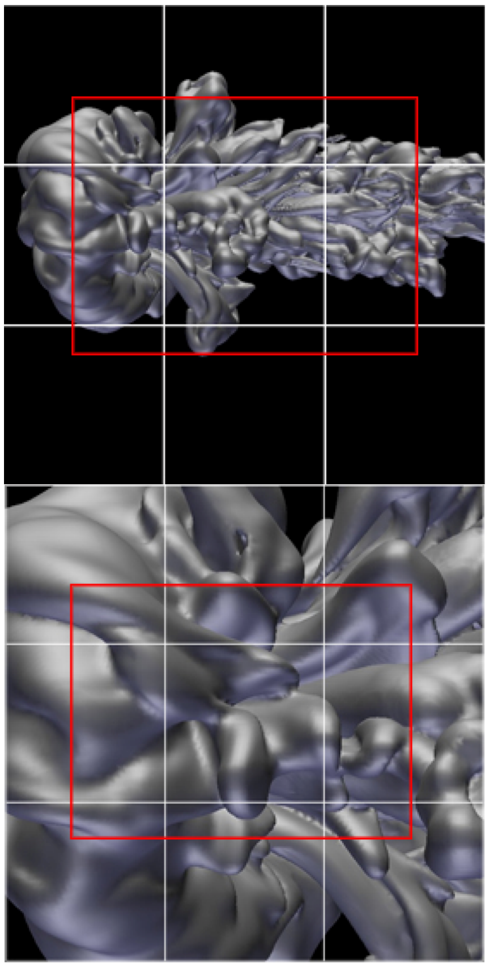


Fig. 7. Outline view (top) and detail view (bottom). The visible area, inside the red box, is visible on-screen. Outside, the pixels are "wasted." In the top view, the cost of downloading "wasted pixels" is negligible since they compress well. In contrast, it is more expensive to download the "wasted pixels" in the bottom view as they don't compress as well.

In Figure 7, the top image shows an overview of a scene using low-resolution tiles. The visible area – shown as a red outline box – is the portion that is displayed at this particular level of zoom. The remainder of the Available Area – the area outside the red box – is not displayed at the client. The pixels in the Available Area outside the red box are in effect "wasted" in this particular view. They would be useful if the user were to zoom out or translate the viewpoint. The bottom image shows a focus view from the same scene. As the user zooms in, the client requests higher-resolution tiles (if they are available) from the server and uses them to supplant the lower-resolution tiles for the zoomed in view. It is important to note that the same number of tiles and pixels are being displayed in both context and focus views. The difference is that the context view uses low-resolution tiles while the focus view uses high-resolution tiles.

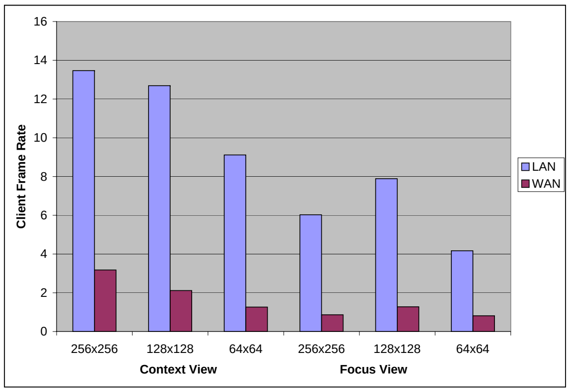


Fig. 8. Performance in frames-per-second for the context view shown in Figure 7. Here, we measure performance while varying tile size and run the application over several different network configurations, both with and without striped connections.

Figure 8 shows client-side frame rates for both the context and focus views of Figure 7. We ran these tests over the networks shown in Table I. To simplify the presentation in Figure 8, we averaged the client-side frame rates from the two WAN configurations to produce a single WAN performance number. We had the client perform about 100 rotation steps and measured client-side frame rate.

There are two main messages from the test results in Figure 8. First, client-side frame rate is dependent upon image network throughput, which is a function of network speed, latency and image compression characteristics. For this particular image sequence, tiles in the context view compress better than those in the focus view. Second, the degree of adverse performance impact caused by downloading "wasted pixels" similarly varies as a function of network throughput and image characteristics. Scenes such as the one we show here have a high variance in compression between low and high-resolution image tiles. That variance is reflected in client-side performance. We would expect that scenes having less variance in compression would correspondingly show less client-side performance variability when switching resolution levels.

E. Prefetching Design and Impact on Performance

As discussed in Section III-D, the basic idea behind prefetching is to predict which frames the user may wish to see "next" or "soon" and request them from the server ahead of time. The desired objective is an overall improvement in frame rate as the latency caused by requesting, waiting for, processing and displaying a given image would be "hidden" from the user.

Since prefetching involves predictions in an $n-$dimensional space, our performance measurement methodology uses two types of tests. The first is a scripted, orderly "user navigation" through a single dimension of the $n-$dimensional parameter space. We conducted one such test for each of zooming, rotation and viewpoint translation. The second test consists of unconstrained user navigation through the $n-$dimensional parameter space. We extended our client so that user interactions may be input via a scripting interface. In this way, the testing conditions are consistent and reproducible. To generate the scripts, we modified the client to journal the user actions it sees. We then created journals for each of four different tests. Three of these consist of navigation through a single dimension. One consists of random navigation through a six-dimensional space (azimuth, polar angle, viewpoint translation, time and isocontour level).

These tests were conducted using the same server and client hardware described in Section IV-A, but over a different set of networks. These networks consist of three DSL-type connections and one broadband connection – all are typical residential services. One of the three DSL connections was to Taiwan, where the latency between client and server was about 175ms, compared to an average 15ms latency over domestic DSL services. The broadband connection has higher bandwidth than DSL – approximately 3.5Mb/s compared to 1.3Mb/s – but has higher latency as well – approximately 43ms compared to 15ms.

We used a fixed set of prefetch parameters: prefetch depth of three and prefetch ratio of 50%. We did not measure performance using a different set of prefetch depth and prefetch ratio parameters. Such expanded performance characterization would form the basis for future work.

Rather than measure and report absolute frame rate at the client, we instead accumulate the amount of "delay time" a client spends between deciding it needs to display "the next" image and when display is complete. Delay time will be less if the image is in the client's cache, and greater if at the server.

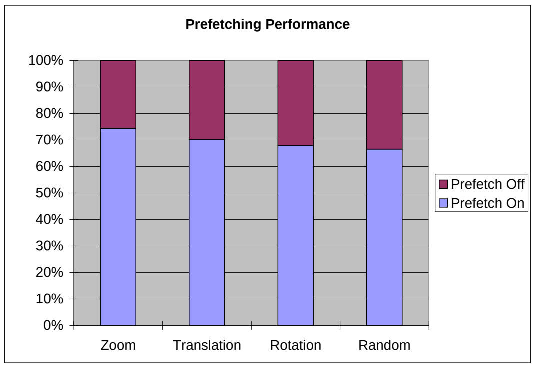


Fig. 9. Image prefetching can substantially reduce the amount of time the client spends waiting for new image data, even in cases of random, unconstrained navigation. This graph compares the relative amount of time the client spends waiting on new image data to arrive with and without prefetching in four different user navigation modes. In the case of Translation, the client with prefetching enabled spends about $3t$ units of time waiting for images to arrive, while the client without prefetching waits about $7t$ units of time. While the absolute amount of speedup varies depending upon various factors, including interaction mode, we see consistent performance improvement across all test cases.

In Figure 9, we report the relative amount of time the client spends waiting for image data with and without prefetching in several different navigation modes. We averaged the "wait time" across all four test networks to simplify the presentation in Figure 9. Looking at the second column labeled "Translation," the client without prefetching spends about $7t$ units of time waiting for the next image to be ready for display while the client with prefetching will wait only about $3t$ units of time – this example shows an effective speedup of about 233% in client-side frame rate when prefetching is enabled. We are displaying relative time here rather than absolute time as the relative improvement with and without prefetching is of primary interest.

As expected, prefetching helps more when there is an "orderly" navigation through a single dimension; it helps the least when there is a random navigation through many different dimensions. The effect of prefetching helps to minimize the interframe latency and variability, which in turn has a positive impact on usability.

F. Discussion

According to our test results, we see tiles sizes of 256x256 and 128x128 result in the best overall performance. A smaller tile size may result in slightly better performance in some cases, but also will result in larger memory consumption due to the larger map structure in the memory. Threaded requests make better use of TCP-based network connections. The optimal number of threaded connections is system and connection dependent. Increasing the size of the client memory cache can result in higher "hit rate" of tile requests and improve overall performance. A local disk cache can augment the memory cache, and gives performance rates comparable those of the memory cache. Image prefetching helps to hide the latency associated with image downloads. Our test results show that client-side performance increases by an amount ranging between about 180% to 300%.

Our tests, which focus on measuring the performance impact of individual parameters in the system, do not reflect a multi-user population simultaneously accessing a collection of images. In such a configuration, client-side performance will still be dominated by time required to obtain the source images from the server. Since we are using a standard web server to service requests for images, those who wish to deploy a production-capable version of our approach will benefit from a vast body of existing knowledge describing optimization of large web server operations in production environments.

V. Conclusion and Future Work

This work presents a novel approach to remote delivery of interactive scientific visualization results. Inspired by QuickTime VR object movies, we have shown how to overcome limitations of fixed image resolution and unbounded client-side memory consumption. This work has several distinct, positive characteristics making it attractive as a vehicle for delivering visualization and rendering results. First, client-side image display rates are independent of source dataset size, image size, visualization technique or rendering complexity. Second, "expensive" images can be computed once, perhaps offline and on large computing systems, and then served to a large audience of consumers thereby amortizing the cost of creating images through image reuse. Third, the client speaks to a "garden variety" web server using an industry standard protocol (HTTP) to request images – no special back-end machinery is needed.

During this study, we identified several areas for future work. One is to explore mechanisms for predictive image prefetching, which would help to further hide latency and improve client-side performance. Related is the notion of "auto-tuning" the tunable prefetch parameters as well as an exhaustive study of how tunable prefetch parameter settings impact performance. As our approach does not overcome a fundamental design limitation of QTVR – namely, discrete samples through a continuous space (constrained vs. unconstrained navigation) – interesting future work would explore using IBR techniques in the client for creating in-between images for viewpoint changes as well as LOD-blending to smooth transitions between different levels of multiresolution image data. The notion of coupling our approach with a live-running visualization application, particularly leveraging the prefetching algorithm to have the application generate frames "ahead of the user" might have the effect of reducing latency.

Acknowledgment

This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098. The authors wish to thank Scott Bishop of SFSU who used Pixie to generate the photorealistic image sequences of the Buddha and dragon models that we used in some of our performance experiments. The authors also wish to thank John Shalf of LBNL, who made significant contributions to the early architectural design of the system, along with anonyomous reviewers for their helpful suggestions for improving the presentation.

References

[1] William Allcock, John Bresnahan, Rajkumar Kettimuthu, and Michael Link. The Globus Striped GridFTP Framework and Server. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, 2005. IEEE Computer Society.

[2] Micah Beck, Terry Moore, and James S. Plank. An End-to-end Approach to Globally Scalable Network Storage. In *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 339–346, New York, NY, USA, 2002. ACM Press.

[3] Wes Bethel, Brian Tierney, Jason Lee, Dan Gunter, and Stephen Lau. Using High-Speed WANs and Network Data Caches to Enable Remote and Distributed Visualization. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Washington, DC, USA, 2000. IEEE Computer Society.

[4] Ian Bowman, John Shalf, Kwan-Liu Ma, and E. Wes Bethel. Performance Modeling for 3D Visualization in a Heterogenous Computing Environment. Technical Report LBNL-56977, Lawrence Berkeley National Laboratory, Visualization Group, Berkeley, CA, USA, 2004.

[5] David M. Butler, James C. Almond, R. Daniel Bergeron, Ken W. Brodlie, and Robert B. Haber. Visualization Reference Models. In *VIS '93: Proceedings of the 4th conference on Visualization '93*, pages 337–342, 1993.

[6] Jerry Chen, E. Wes Bethel, and Ilmi Yoon. Interactive, Internet Delivery of Scientific Visualization via Structured, Prerendered Imagery. In *Proceedings of 2006 SPIE/IS&T Conference on Electronic Imaging, Volume 6061, A 1-10*, 2006.

[7] Shenchang Eric Chen. QuickTime VR – An Image-Based Approach to Virtual Environment Navigation. *Computer Graphics*, 29(Annual Conference Series):29–38, 1995.

[8] Daniel Cohen-Or and Eyal Zadicario. Visibility Streaming for Network-based Walkthroughs. In *Graphics Interface*, pages 1–7, 1998.

[9] R3vis Corporation. OpenRM Scene Graph. http://www.openrm.org, 1999-2006.
[10] D. R. Commander. VirtualGL. `http://www.virtualgl.org`.
[11] David Ellsworth, Chris Henze, Bryan Green, Patrick Moran, and Timothy Sandstrom. Concurrent Visualization in a Production Supercomputer Environment. *IEEE Transactions on Visualization and Computer Graphics – Proceedings Visualization 2006*, 12(5):997–1004, Sept.–Oct. 2006.
[12] Klaus Engel, Ove Sommer, Christian Ernst, and Thomas Ertl. Remote 3D Visualization Using Image-Streaming Techniques. In *Advances in Intelligent Computing and Multimedia Systems (ISIMADE '99)*, pages 91–96, 1999.
[13] Google Inc. Google Earth. `http://earth.google.com/`.
[14] Google Inc. Google Map. `http://maps.google.com/`.
[15] GUP Linz Institute of Graphics and Parallel Processing. GVid Project Page. `http://www.gup.uni-linz.ac.at/gvid/`.
[16] T. Hacker, B. Noble, and B. Athey. Improving Throughput and Maintaining Fairness Using Parallel TCP, 2004.
[17] Bernd Hamann, E. Wes Bethel, Horst Simon, and Juan Meza. Visualization Greenbook: Future Visualization Needs of the DOE Computational Science Community Hosted at NERSC. *International Journal of High Performance Computing Applications*, 17(2):97–124, 2003.
[18] H. Hege, A. Hutanu, R. Kähler, A. Merzky, T. Radke, E. Seidel, and B. Ullmer. Progressive Retrieval and Hierarchical Visualization of Large Remote Data. In *Proceedings of the Workshop on Adaptive Grid Middleware*, Sept 2003.
[19] Hans-Christian Hege, André Merzky, and Stefan Zachow. Distributed Visualizaton with OpenGL VizServer: Practical Experiences. ZIB Preprint 00-31, 2001.
[20] Paul Heinzlreiter and Dieter Kranzlmüller. Visualization Services on the Grid: The Grid Visualization Kernel. *Parallel Processing Letters*, 13(2):135–148, 2003.
[21] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance – RFC 1323, May 1993.
[22] Kitware, Inc. and Jim Ahrens. ParaView: Parallel Visualization Application. `http://www.paraview.org/`.
[23] Dieter Kranzlmüller, Gerhard Kurka, Paul Heinzlreiter, and Jens Volkert. Optimizations in the Grid Visualization Kernel. In *IEEE Parallel and Distributed Processing Symposium (CDROM)*, pages 129–135, 2002.
[24] Lawrence Livermore National Laboratory. VisIt: Visualize It Parallel Visualization Application. `http://www.llnl.gov/visit/`.
[25] Eric J. Luke and Charles D. Hansen. Semotus Visum: A Flexible Remote Visualization Framework. In *VIS '02: Proceedings of the Conference on Visualization '02*, pages 61–68, Washington, DC, USA, 2002. IEEE Computer Society.
[26] Chris Maxwell, Randy Kim, Tom Gaskins, Frank Kuehnel, and Patrick Hogan. NASA's World Wind. `http://worldwind.arc.nasa.gov/`.
[27] Bruce McCormick, Thomas DeFanti, and Maxine Brown. Visualization in Scientific Computing. 21(6), November 1987.
[28] Valerio Pascucci and Randall J. Frank. Global Static Indexing for Real-time Exploration of Very Large Regular Grids. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, New York, NY, USA, 2001. ACM Press.
[29] Steffen Prohaska, Andrei Hutanu, Ralf Kahler, and Hans-Christian Hege. Interactive Exploration of Large Remote Micro-CT Scans. In *VIS '04: Proceedings of the Conference on Visualization '04*, pages 345–352, Washington, DC, USA, 2004. IEEE Computer Society.
[30] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1):33–38, 1998.
[31] John Rohlf and James Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for Real-time 3D Graphics. In *SIGGRAPH*, pages 381–394, 1994.
[32] Douglas C. Schmidt and Tatsuya Suda. Transport System Architecture Services for High-Performance Communications Systems. *IEEE Journal on Selected Areas in Communications*, 11(4):489–506, 1993.
[33] Silicon Graphics Inc. OpenGL Vizserver. `http://www.sgi.com/products/software/vizserver/`.
[34] IND Networking Performance Team. NetPerf. `http://www.netperf.org/netperf/NetperfPage.html`.
[35] Ilmi Yoon and Ulrich Neumann. IBRAC: Image-Based Rendering Acceleration and Compression. In *Eurographics 2000*, volume 19, pages 321–330, 2000.
[36] Zoomify Inc. Zoomifyer. `http://www.zoomify.com/`.

Jerry Chen earned his BS in Computer Science from the University of California, Davis in 2001 and is working towards his MS degree at San Francisco State University. He is a Software Engineer at Yahoo! Inc. in its Small Business department where he helped revamp an ordering service with Web2.0 technologies and improved the system's scalability for dynamic marketing strategies. His research interests include internet application design, user interface/interaction design, visualization, computer graphics.

Dr. Ilmi Yoon is an Assistant Professor in the Computer Science Department of San Francisco State University. She earned both her MS and Ph.D. degrees in Computer Science at the University of Southern California at Los Angeles in 1996 and 2000. Her focus was on "Web-based Remote Rendering using Image-Based Rendering Techniques." Her recent research focuses on 3D network visualizations on the WWW, 3D Visualizations related to Life Science, and Serious Games for Nursing. Her research relates to interactive media, Web3D, and Information Visualization.

E. Wes Bethel is a Staff Scientist at Lawrence Berkeley National Laboratory. His research interests include high performance remote and distributed visualization algorithms and architectures. He earned his MS in Computer Science in 1986 from the University of Tulsa and is a member of ACM and IEEE.