

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

Jerry Chen¹, E. Wes Bethel², Ilmi Yoon¹

¹San Francisco State University

²Lawrence Berkeley National Laboratory

ABSTRACT

In this paper, we explore leveraging industry-standard media formats to effectively deliver interactive, 3D scientific visualization to a remote viewer. Our work is motivated by the need for remote visualization of time-varying, 3D data produced by scientific simulations or experiments while taking several practical factors into account, including: maximizing ease of use from the user's perspective, maximizing reuse of image frames, and taking advantage of existing software infrastructure wherever possible. Visualization or graphics applications first generate images at some number of view orientations for 3D scenes and temporal locations for time-varying scenes. We then encode the resulting imagery into one of two industry-standard formats: QuickTime VR Object Movies or a combination of HTML and JavaScript code implementing the client-side navigator. Using an industry-standard QuickTime player or web browser, remote users may freely navigate through the pre-rendered images of time-varying, 3D visualization output. Since the only inputs consist of image data, a viewpoint and time stamps, our approach is generally applicable to all visualization and graphics rendering applications capable of generating image files in an ordered fashion. Our design is a form of latency-tolerant remote visualization infrastructure where processing time for visualization, rendering and content delivery is effectively decoupled from interactive exploration. Our approach trades off increased interactivity, reduced load and effective reuse of coherent frames between multiple users (from the server's perspective) at the expense of unconstrained exploration. This paper presents the system architecture along with an analysis and discussion of its strengths and limitations.

CR Categories: I.3.2 [Computer Graphics]: Graphics Systems, Remote systems, Distributed/network graphics.

Keywords: remote and distributed visualization, latency tolerant visualization, streaming media, interactive techniques.

1. INTRODUCTION

The scientific community faces a well-recognized challenge - the need to perform data analysis and visualization on data that is located "somewhere else." This situation occurs when large simulation results are placed on large storage caches at computing centers, yet the scientist or analyst is located elsewhere [1,2]. In the general subject area of remote and distributed visualization, there are several different approaches aimed at solving the fundamental problem of facilitating interactive exploration of large, complex, multidimensional datasets from a remote location.

Figure 1 shows three possible ways to partition the visualization pipeline. The portions of each pipeline configuration in blue are local to the user, while those in red are the remote from the user. In the top row, all of the simulation or experiment data are sent to the remote client for analysis, visualization and rendering. In this configuration, pipeline performance will be dominated by the cost of sending data across the network link. While this approach provides the best performance in terms of local desktop interactivity, it presumes (1) that the data will fit on the local workstation, and (2) the wait time for data to be moved to the workstation is not excessive. A fact of life we face is that scientific simulations and experiments routinely generate multi-terabyte datasets, and neither of these presumptions applies.

Another pipeline partitioning is shown in the middle row in which data is stored at, read from and visualized on the remote resource, and visualization results (predominantly geometry) are sent to the local client where they are rendered and displayed. This approach has the advantage of providing excellent desktop interactivity once the visualization results have moved across the network link. Additionally, this approach permits use of scalable visualization resources

on the remote site to accelerate processing, to amortize the cost of I/O across many processors, and so forth. It also presumes (1) that all the visualization results will fit onto the the desktop and (2) visualization results are mostly geometry/surfaces because non-geometry data (e.g., volumes of data) requires network bandwidth proportional to the original data size along with intensive computation for rendering. In many instances, these assumptions are not true.

In the pipeline configuration shown in the bottom row, data is stored at, read from, visualized and rendered on the remote resource. The resulting images are sent to the viewer. Transmission of rendered images can become a desirable approach if the delivery of rendered images can support what appears to be n -dimensional navigation, where the n dimensions include time, space or any other parameter that controls either the content or presentation of the visualization.

The trend in the sciences is that data sizes grow at a rate proportional to increases in processing power and storage capacity. This trend is expected to continue for the foreseeable future. The anticipated data growth rate exceeds the projected ability to move “all the data” from the central computing center to a remote location for interactive analysis. We look to the third pipeline configuration – send images – as a way to decouple network bandwidth requirements for remote visualization from the increasing size of data to be visualized. In addition, such an approach allows us to leverage parallel computing, visualization an I/O infrastructure often present at central computing facilities. Using images as the fundamental exchange medium in remote visualization settings offers several distinct advantages over other pipeline configurations: (1) network bandwidth requirements are independent of data size, but realizable compression ratios for the images will be a function of image size and content; (2) support for a broad range of client-side platforms, ranging from a high performance desktop to a PDA for field scientists; (3) reuse of image frames is maximized, which reduces computational load on the back-end visualization infrastructure; (4) maximum compatibility with visualization applications, since nearly all produce images as output.

An image-only remote visualization architecture presumes (1) it is feasible to deliver precomputed images in a fashion that gives the illusion of 3D/4D navigation, and (2) that adequate scientific insight may be obtained through the constrained navigation that will result by sampling a finite number of viewpoints. To answer (1), we present in this paper techniques for implementing a two approaches to assembling structured images using industry-standard digital media formats that achieve the objective of 3D/4D navigation over the Internet. The question raised in (2) is one of human cognitive abilities: we observe that in many instances, users typically use only a subset of all possible visualization functionality to understand 3D shape and depth relationships or to browse time-varying data or other visualization attributes.

In Section Two, we review related work in remote visualization implementations. In Section Three, we describe our approach, which includes an overall system architecture, image preprocessing, media encoding and image access/delivery mechanism. In Section Four, we present the results of a performance analysis. Finally, we conclude with discussion about our approach, including limitations, along with suggestions for future work

2. RELATED WORK

In the “move data to the user” pipeline configuration, Beck et al. have focused on data staging and efficient use of networking to efficiently move bulk data from one location to another [3]. In the present day of growing data sizes,

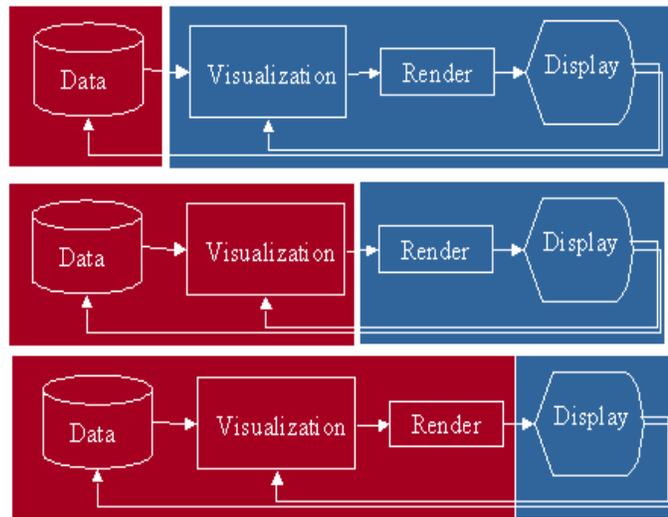


Figure 1. Three possible partitionings of the visualization pipeline. The portions of the figure in blue are local to the user, while those in red are remote from the user.

there is a downside to this approach: multiple copies of an already large dataset are being made at intermediate staging and remote caching depots. This approach is not tractable as datasets begin to range into the 100s of TB. The problem gets amplified as more and more users on collaborative teams try to make additional copies of large datasets – there will be additional and arguably unnecessary load placed onto storage and network systems.

Over the years, several different approaches have been used in the second pipeline configuration, where visualization results – rather than bulk data or final images – are sent between the remote back-end server and the client. CEI's Ensign Gold, when run in "Server of Servers" mode, uses a similar parallel and pipelined decomposition to perform visualization on a parallel machine, then sends results (geometry) to a remotely located client [4]. Web-based remote visualization often generates VRML that is then sent to remote clients [5]. Visapult [6] uses a hybrid approach using parallel-pipelined architecture to implement scalable, remote and distributed volume rendering. The Visapult approach is "hybrid" because a combination of pre-rendered images and a "geometry lattice" are transmitted between the back-end and client. The pre-rendered images are direct volume rendering for data subsets; they are not "final images." These images are used as textures that the client viewer then texture-maps onto the geometry lattice. Generally speaking, the amount of data moved between the Visapult back-end and the client viewer is logarithmic with respect to size of the source data. All these examples aim to maximize client-side rendering interactivity by sending visualization results for retained-mode rendering. Once the initial data transfer occurs, desktop interactivity is completely divorced from network performance characteristics. However, all these approaches will impose substantial demands on client-side resources, including use of GPUs to accelerate rendering along with system memory to hold the geometry data produced by visualization results. In many instances, the size of visualization output can be proportional to or larger than the source data, such as when generating an isosurface of noisy data. Depending upon the specific client and the specific network, such an approach may simply not be feasible due to the sheer amount of data produced by the visualization algorithm.

Using images as the remote delivery medium has also been well explored in a number of different ways over the years. An early effort here relied on remote Xlib capabilities, where a client application was run at the central facility but then sent commands to the user's local X Server. More recent work has focused on overcoming the limitations of remote Xlib by providing access to remote desktops. VNC [7] uses a custom client viewer to intercept events on a client workstation and send them to a VNC server, which is sometimes implemented as an X extension. VNC overcomes most problems of remote Xlib, but does not support delivery of hardware-accelerated rendering: in direct rendering configurations, OpenGL commands bypass the X server (and consequently VNC). SGI's OpenGL Vizserver [8] provides the ability for pixels produced by an Xlib or OpenGL application run on a server to be sent to a custom client viewer. Both the VNC and Vizserver approaches intercept local input events on the client and send them to the remotely running application. Both use image compression algorithms to accelerate delivery of results. To further enhance the effect of image compression, other approaches utilize simple geometry created on the server side, transmitted and rendered on the client side while the full data set and simple geometry both rendered on the server side and only the image difference (usually high frequency portion) is transmitted to enhance the delivery of rendered images [9,10]. These approaches aim to reduce the bandwidth by utilizing desktop's computing power. These approaches, however, increase overhead at server side to compute both the high quality and low quality image and their differences.

An alternative approach not depicted in Figure 1 is one where image contents are reused; none of the above approaches reuse images – each will regenerate a completely new image when needed. Image based rendering (IBR) refers to a set of technologies that render a new frame from existing frames rather than from source data (geometry). Because the input is an image or set of images, IBR's computational complexity for incremental frames is independent from the complexity or size of data sets and the quality of rendered images. IBR approaches are useful for the remote scientific visualization because the total size of images that are required for rendering is much smaller than the size of the scientific data. 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 server sends only the difference between the frames. The approach maintains high image quality and achieves high compression ratio, but server needs to be synchronized with client all the time and algorithm works only with iso-surface rendering [11].

Apple's QuickTime VR [12] provides navigation or data exploration using pre-rendered images or photos. There are two main kinds of QTVR: panorama and object movies. QTVR panorama movies permit a user to "pan the viewpoint" through 360 degrees of rotation to look at the environment from a fixed point, and support a zoom-in and zoom-out

capability. A QTVR panorama movie consists of a single panoramic image, or image sequence in the case of time-varying movies. They may also contain hyperlinks to other QTVR movies. QTVR object movies provide the ability to look at an object from different angles, along with the ability to zoom-in and zoom-out. Unlike the panorama movie, a single time step of a QTVR object movie consists of many images that represent the view of the scene from different viewpoints. Time varying QTVR object movies contain a time sequence corresponding to each view. When a user changes their viewpoint or plays the animation, the QuickTime player will find the image corresponding to the current view and display it.

The work we present here leverages the QTVR Object Movie concept to implement “simulated” 3D, time varying navigation of scientific visualization results. Images are first generated by a visualization application from a number of prescribed viewpoints. These images are then encoded into QTVR Object Movie format, or into a structured image collection contained by reference in a web page. In the case of the QTVR encoding, a standard QuickTime viewer serves as the viewer on the client side. In the case of the structured image collection, the web browser downloads the web page containing URLs to the structured image collection along with a small amount of JavaScript code that serves to implement the interactive image navigation capability. This approach effectively decouples the cost of scientific visualization rendering from the act of interactive exploration, but at the expense of unconstrained interaction. It provides maximum reusability of frames as one single set of images – the QTVR object movie or the structured image collection – is sent to all clients who request it. Likewise, due to image data being the source, the approach is widely applicable to any visualization or rendering application that can generate image files in a prescribed manner.

3. IMPLEMENTATION

In the work we present here, we’re interested in alternatives to the traditional pipeline decompositions to better support interactive, remote, 3D visualization. Specifically, we want to be able to support fully interactive, 3D visualization of temporally varying data or visualization attributes using industry standard client-side viewers that are readily available to anyone. We observe that in many instances, users require only a subset of all possible visualization functionality: to understand 3D shape and depth relationships or to effectively browse temporally varying attributes. The approach we take is to explore ways to deliver visualization and rendering results, i.e., images, in a way that provides the experience of 3D, interactive exploration. We desire a delivery mechanism that strikes a balance between cost and functionality. Sending images rather than data is a better approach in terms of easy usability (little or no client-side installation or configuration is required) and scalability in terms of accommodating visualization results from ever-large data sources. The experience of 3D interaction with a dynamic scene is achieved by displaying precomputed images from different

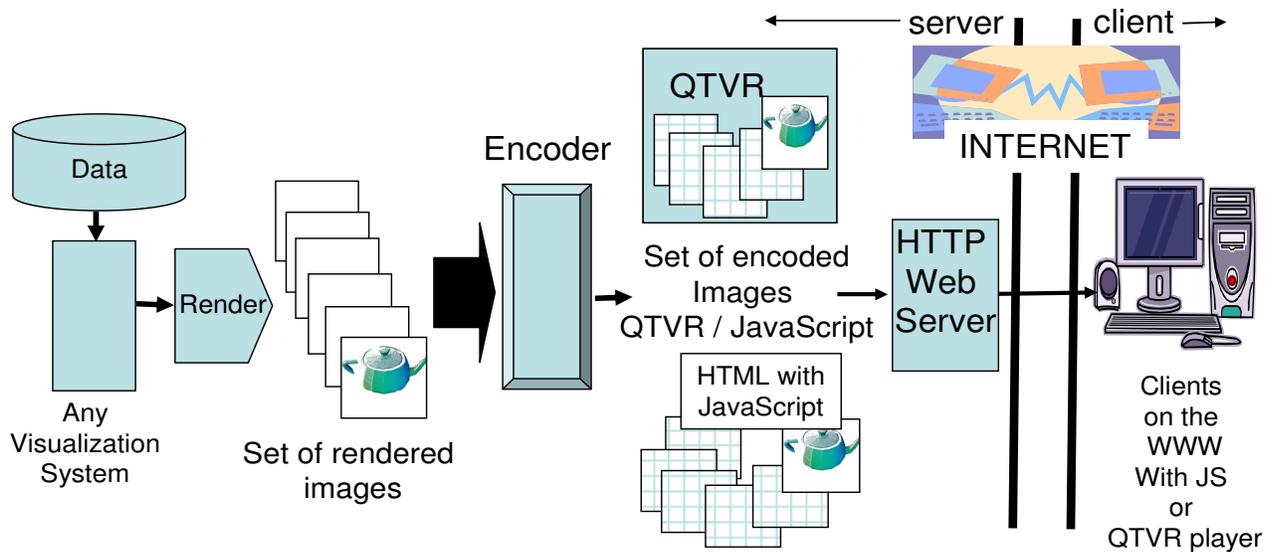


Figure 2. MBender Architecture: Data flow and system components.

viewpoints and over time. These images may be generated by any visualization or rendering system, thus this approach is widely applicable to a large number of potential uses. During interaction on the client side, the client viewer application, whether it is a QTVR player or our custom JavaScript code, allows the user to navigate through space or time represented by the structured image collection.

As shown in Figure Two, any visualization system can be used to generate the image sets that are then used as input to an encoder step. The encoder produces either a QTVR Object Movie or a web page containing URLs to images on the web server along with a small amount of JavaScript code that implements the client-side navigation capability. These QTVR movies and JS media are requested by the client then delivered by a web server; no special server-side configuration is required.

The QTVR Object Movie format is essentially a “solid file format.” it is a block of data that contains all the images that comprise the entire movie. It can be thought of as a three-dimensional array of images, where rows correspond to views from a different angles of azimuth, columns correspond to views from a different elevations, and the depth dimension contains time varying images from a azimuth, elevation pair. Whereas QTVR panorama movies correspond to many views from a given viewpoint, the QTVR Object Movie uses a reverse viewing concept. It contains multiple views of an object fixed in space such that the multiple views correspond roughly to “orbiting about the object.” Both the QTVR Object Movie and Panorama Movie player support a zoom-in/zoom-out capability on the client side. To create the QTVR digital media, our encoder application will encode images into QTVR media format with proper headers and metadata. Currently, QTVR does not leverage the “on-demand data transmission” feature – data is transmitted only when demanded. As consequence, users must download the entire QTVR Object Movie before starting to navigate through the scene. We found the QTVR Object Movie player presents a usable interface for navigation through 3D, time varying scenes. There are some limitations with this approach, however. First, the entire QTVR Object Movie will be loaded into memory. This approach is fine for small movies, but is intractable for large ones. Second, the QTVR player doesn’t offer client-configurable controls to optimize use of resources like memory.

In contrast to the QTVR Object Movie approach, we implemented an alternative navigation and image cataloguing and delivery mechanism. The “encoding” step here is to create an HTML page containing some metadata and JavaScript code – a very compact representation since images are not downloaded until needed. During navigation, URLs for images are dynamically constructed by the JavaScript code depending upon virtual viewpoint, temporal location and so forth. Note that the client’s web browser – the execution environment for the JavaScript navigation code – will request images when needed. Since the image URL is constructed programmatically, the HTML “media” our encoder produces is quite compact. As viewpoints are updated with mouse movement, the JavaScript interface “feels like” (behaves like) a virtual trackball. The JavaScript code transforms from mouse movement to select an image from a structured image collection that corresponds to a particular viewpoint.

Since JavaScript offers a programmable environment for interacting with the browser, we use it to our advantage to implement an image cache to limit relative resource consumption for a given session. Our implementation contains an internal, tunable image cache containing JavaScript image objects. Since JavaScript is not able to directly measure memory consumption, our tunable image cache limits the number of images that may be resident in JavaScript memory at once. When images are small, the cache has a small memory footprint; when the images are large, the memory footprint for the same number of images may become quite large. The JavaScript cache management code tracks whether or not an image is in memory. If the needed image is not in cache memory, the JavaScript cache manager will request it from the remote web server via the web browser. This approach has the added benefit of being able to take advantage of the web browser’s own cache management infrastructure: before making a new HTTP request for an image, the browser will first check its local disk cache to determine if the image needs to be requested from the remote web server. Overall performance of the JavaScript approach is highly dependent upon web browser, which controls memory management and the size of the available memory.

4. EVALUATION

To evaluate the performance of our approach, we conducted a number of different experiments that measure various aspects of runtime performance. Overall performance is a function of many different factors, including: network bandwidth and latency, degree of image compression (that is in turn dependent upon image content), the size of the

tunable JavaScript image cache, the pixel dimensions of the images being transferred, user interaction behavior, ambient system load, and so forth. We limit our scope of study to focus on memory utilization and image download time to test the premise that the tunable JavaScript image cache achieves good balance between interactive image display performance while maintaining a modest memory footprint. We compare the memory footprint of the JavaScript approach with that required by a QTVR Object Movie player when presented with identical image sequences. Our testing environment is a Pentium 4, 2.4G Hz with 1.5 GB DDR memory. The wide-area network connection in our experiment is residential broadband connection with 451KB/s of throughput and 15.5 milliseconds latency between the client system and remote web server. To measure the memory consumption of QTVR Object Movie player, we generate several movie files with diverse sizes and then monitor the memory consumed by the player when displaying each movie.

Table 1 shows the result of our experiment. We see that the QTVR player itself consumes about 19,964KB of memory. After loading QTVR Object Movies of sizes 4,151KB, 49,540KB and 92,820KB, the QTVR player’s memory footprint is approximately 24,792KB, 69,248KB and 111488KB respectively. On the client, the memory footprint size required by the QTVR player when displaying the object movie is approximately equal to the size of the object movie. When we are interacting with the QTVR player to navigate through the Object Movie, memory consumption remains about the same as when not navigating. This result implies that the QTVR player stores all images comprising the QTVR Object Movie in memory in their original compressed format. During playback, the QTVR player decompresses images on the fly for every single view and time step during navigation. With current processor speeds, on-the-fly decompression is performed at a rate suitable for interactive navigation. Figure 3a and 3b shows the memory consumption pattern of QTVR in comparison of JavaScript media along with effective frame rate.

	QuickTime Player only	stime.mov	vrtime.mov	isotime.mov
Digital Media Size	0	4151	49540	92820
Number of Frames		37x11 = 407	10x20x11 = 2200	20x37x11=8140
Initial Loading Time	1~2 sec	10.470 sec	122.187 sec	222.503 sec
FPS after all images are loaded (400x300)		About* 20 fps	About* 20 fps	About* 20 fps
FPS after all images are loaded (1600x1200)		About* 4~5 fps	About* 4~5 fps	About* 4~5 fps
Memory Consumption	19964	24792	69248	111488

Table 1. Performance of QTVR Object Movie in its downloading, loading into memory and memory consumption (in KB) by using QuickTime Player 6.5.2. (**note-** About* indicates the fps is measured by the number of frames divided by navigation time observed by user after loading is done.)

In terms of usability and navigability, the QTVR player offers the ability to zoom in or out of a particular view. This type of interaction allows a user to examine a small region of the image at higher resolution for closer inspection of detail. One drawback with the QTVR media encoding approach is that the zooming operation uses a fixed image resolution. The result is that zoomed-in views are achieved by scaling an image. If the original image size is relatively small, the zoomed-in views offer no greater detail than the original and visual quality drops. Figures 4a and 4b illustrate the difference between zooming in on a low-resolution image and a high-resolution image. Using very high resolution images help alleviate the visual fidelity problem during zoom-in operations, but doing so requires more time for downloads as well as a substantially greater memory footprint at runtime. This behavior limits the capability for QTVR Object Movies to deliver high resolution and high quantity images set.

Early in our development of the JavaScript implementation, we followed the same overall resource use pattern as QTVR Object Movies. We preloaded all the images into browser’s memory when loading the page. The result was not optimal since memory consumption would grow unbounded. When physical memory was exhausted and the system began to use virtual memory, system performance was seriously degraded due to memory paging. We found that for a

given number of images, the QTVR player has a smaller memory footprint than a web browser. The reason is that web browsers maintain images in memory in an uncompressed format (Figure 3a). Table 2 shows the relationship between number of images and memory consumption when using IE 6.0.2900, Mozilla 1.7.6, and FireFox 1.0.6. We observe that these web browsers consume a substantial amount of memory, especially when caching higher resolution images.

#of cached image	1	64	128	256	512	1024	2048
IE 400x300	22192	64592	87360	132448	222142	405476	769984
IE 1600x1200	34028	394792	755916	1465312	n/a	n/a	n/a
Mozilla 400x300	23544	43808	69460	114612	207116	389308	753084
Mozilla 1600x1200	23772	379064	740380	1446740	n/a	n/a	n/a
FireFox 400x300	23776	46548	68668	114748	206064	388644	752240
FireFox 1600x1200	23852	379196	740452	1452728	n/a	n/a	n/a
Initial Loading Time	<1~2 sec						
First pass fps (400x300)	About* 5~10 fps						
First pass fps (1600x1200)	About*2~4 fps						
After first pass fps	About* >20 fps	About* >20 fps	About* >20 fps	About* >20 fps	About* >20 fps	About* >20 fps	About* >20 fps

Table 2. Performance of JavaScript version - this table shows the memory consumption for caching different numbers of images versus different browsers with different image resolution. The versions of the browsers are IE 6.0.2900, Mozilla 1.7.6, FireFox 1.0.6. Memory consumption for better resolution is huge. All three browsers behave similarly on the ratio of cached image to memory consumption. During first pass, JS downloads images and then loads images into memory, so it is lower. From the second pass, JS uses decompressed frames directly from memory, so fps is as high as QTVR or higher (due to decompressed format while QTVR uses compressed format in memory). (**note-** About* indicates the fps is measured by the number of frames divided by navigation time observed by user after loading is done, and “<” or “>” are used to abbreviate “less than” and “more than” respectively.)

These memory use patterns motivated us to implement a mechanism to limit a browser’s memory requirement when navigating through high resolution, time varying 3D visualization results. Our solution is a finite- and tunable-sized image cache implemented in JavaScript. It limits the number of images in memory, and avoids the memory oversubscription problem by downloading images only when needed for interactive navigation. We implement our JavaScript image cache as an image queue, and cache all the incoming image data into the array until it is full. When new images are downloaded, they replace the last recently cached image. Figure 3a shows a case where the image cache is set to 2048 images. Our Table 2 shows the amount of cached images versus required memory for certain resolution of images. Each cell in the table indicates memory consumption for each of the different browsers we tested. Each cell in the table reports browser memory use when the cache is full. When our JavaScript image cache is set to limit the number of images in memory, we limit the amount of memory the browser consumes and thus avoid the poor performance that results from memory paging. Note that our JavaScript image cache limits the number of images in memory, but not their absolute size in bytes: JavaScript has no mechanism to measure and report an object’s memory size. Using profiling tools external to the browser to observe memory consumption, we note that an image object of resolution 400x300 occupies about 355KB memory, and an image resolution of 1600x1200 occupies about 5642KB of memory. The ratio between number of pixels per image and between memory consumptions are almost the same, $(400 \times 300) / (1600 \times 1200) = 0.0625$ which is similar to $355 / 5642 = 0.0629$. Therefore we can use this observation to estimate the actual cache size in bytes, and still be able to control the actual cache size in some degree.

Another performance factor we like to compare between QTVR and JS version is that fps or the accumulated number of frames over time. QTVR waits long to show the first frame, but once downloaded, the fps (frame per second) is pretty high. JS version starts fist frame very quickly, but fps stays low while downloading images and then boosts up higher

than QTVR since it used decompressed images (Figure 3b). Performance of both approaches is sensitive to connection. We just measure the downloading time for the two resolution of our test image set. For 400x300 JPEG image, it takes about 0.103 second (5 ~ 10 fps is possible) and for 1600x1200 JPEG image, it takes about 0.25 second (2 ~ 4 fps is possible) on our cable connection. The number doesn't seem to affect the interaction experience that much, however, a 1600x1200 image takes much longer to be decompressed and the overall interaction rate becomes not that smooth. Thus using higher resolution may improve the visual quality but also decrease the interaction rate. Combine with the observation discussed in the previous paragraph, we need to be careful to support higher resolution image using both approaches because increasing resolution will result in longer delay and huge memory consumption.

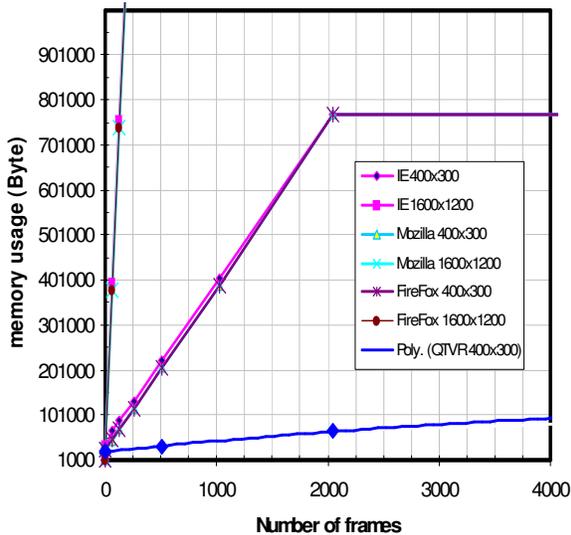


Figure 3a. Comparison of QTVR and JS media on memory consumption. JavaScript uses decompressed images while QTVR uses compressed images showing differences in memory consumption. In this example, the maximum of image cache is set as 2048, which can be set lower to reduce memory consumption in the expense of fps. JS shows higher fps due to usage of decompressed image in the memory.

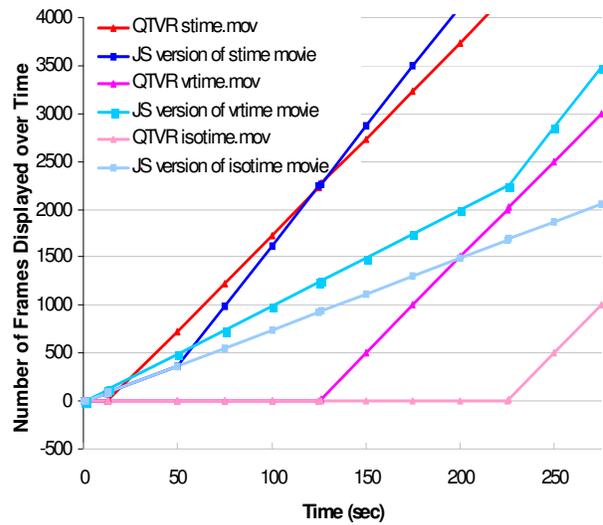


Figure 3b. Comparison of accumulated number of frames displayed over time for three different sets of interactive movie. QTVR starts with long downloading time (especially for bigger ones) and fps is also slightly lower. In case of JavaScript media, the first pass is slower due to downloading and loading into memory, but from the second pass, the fps goes higher than QTVR. Graph was drawn using data from the Table 1 and 2.

5. CONCLUSION AND FUTURE WORK

Delivering interactive 3D, time-varying visualization by using structured images decouples the cost of scientific visualization rendering from the act of interactive exploration. This approach offers a number of distinct advantages when compared to other decompositions of the visualization pipeline. The data transfer cost is bounded by number of views and resolution of rendered images that comprise the interactive movie. This characteristic is increasingly important in light of growing data size and complexity. In the case of our JavaScript implementation, the combination of internal memory and web browser cache offers opportunity for images to be reused on the client. The approach we describe offers the ability to perform 4D interaction/exploration (three spatial dimensions for the viewpoint, and an additional temporal dimension for time varying data or other visualization parameters) using “standard desktop software,” like a standard Web browser, or the QuickTime player plug-in from Apple Computer. The approach is generally applicable in that images may be produced by any visualization or rendering application. No special server-side setup or configuration is required – all our experiments were conducted using an unmodified Apache web server. Images, which may be expensive to compute, may be reused across multiple visualization sessions. A central premise is

that a user may successfully obtain an understanding of 3D structure and depth relationships from a finite number of precomputed views.

However, there are some noteworthy limitations of the approach we present in this paper. For QTVR, the entire Object Movie file must be completely downloaded onto the local system before users may begin full navigation. Another issue is memory consumption: in QTVR, all images must be loaded into memory before users may navigate through the scene. A QTVR Object Movie containing scientific visualization may become quite large and may not be playable depending upon the amount of available system memory. Third, when zooming in for close-up views, the resolution of the displayed image will remain the constant. The resulting pixel zoom during display results in degraded visual quality. Although our JavaScript implementation provides on-demand data downloading and has more flexibility in terms of its image caching strategy, it also suffers from the same fixed-resolution image display artifacts as QTVR. In addition, the



Figure 4a. Visual artifacts result when using a fixed image resolution for zoom-in operations.

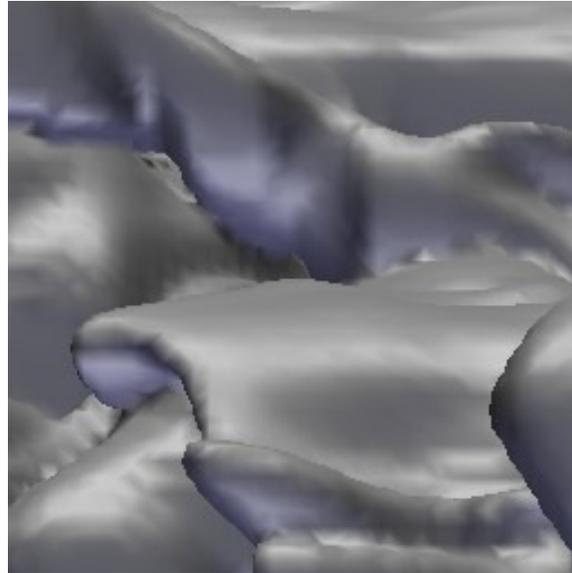


Figure 4b. A zoomed-in view of a high resolution image is free of artifacts, but consumes substantially more resources for a given QTVR Object Movie.

web browser controls overall memory performance, and different browsers have different memory management characteristics. Unlike QTVR, the JavaScript image object model ends up storing image data in memory in an uncompressed format. Another limitation is that our approach does not offer the possibility of unconstrained navigation. If views are generated on 10-degree intervals of azimuth, users do not have the possibility for a view at a 5-degree interval.

Future work should focus on techniques to overcome the limitations we have identified in this paper. A multiresolution approach for image delivery and display would be useful to overcome the visual artifacts associated with fixed-resolution image zooming. More intelligent prefetching would make more effective use of network bandwidth as well as available system memory. A more thorough study of the impact of varying multiresolution parameters and prefetching strategies would with understanding their relationship upon system performance in common use scenarios.

ACKNOWLEDGEMENTS

This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, Mathematical, Information and Computational Sciences Division of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231

REFERENCES

- [1] K. Brodlie, J. Brooke, M Chen, D. Chisnall, A. Fewings, C. Hughes, N. W. John, M. W. Jones, M Riding, and N Road, "Visual Supercomputing – Technologies, Applications and Challenges," Eurographics State of The Art Reports 2, Grenoble, France August 30-Sept 3, 2004.
- [2] R. Mount, "Office of Science Data-Management Workshops," Report from the DOE Office of Science Data-Management Workshops, Stanford, May, 2004. Available at <http://www-user.slac.stanford.edu/rmount/dm-workshop-04/Final-report.pdf>
- [3] Beck, M., T. Moore, and J.S. Plank. An End-to-end Approach to Globally Scalable Network Storage. in ACM Sigcomm 2002. Pittsburgh, PA: Association of Computing Machinery.
- [4] Ensign Gold, CEI International. <http://www.ceintl.com/products/ensightgold.html>
- [5] Trapp, J.C., and Pagendarm, H.G., "A prototype for a WWW-based Visualization Service", Eurographics Workshop, Visualization in Scientific Computing '97, pp 21 – 30. (1997)
- [6] Bethel, W., et al., Using High-Speed WANs and Network Data Caches to Enable Remote and Distributed Visualization (LBNL-45365). in Proceedings of SC2000. 2000: Dallas, TX.
- [7] VNC Documentation. AT&T Laboratories, Cambridge. <http://www.uk.research.att.com/archive/vnc/howitworks.html>
- [8] OpenGL Vizserver, Silicon Graphics, Inc. <http://www.sgi.com/products/software/vizserver/>.
- [9] Marc Levoy, "Polygon-Assisted JPEG and MPEG Compression of Synthetic Images," SIGGRAPH '95, <http://graphics.stanford.edu/papers/poly/>.
- [10] D. Cohen-Or and E. Zadicario, "Visibility Streaming for Network-based Walkthroughs" Graphics Interface'98, 1--7, June 1998.
- [11] Ilmi Yoon and Ulrich Neumann, "Web-based Remote Rendering with IBRAC (Image-based rendering acceleration and compression)", Eurographics 2000, Vol 19(3), pp. C321 ~ 330.
- [12] QuickTime VR, Apple Computer, Inc. <http://www.apple.com/quicktime/qtvr/>.