Ubiquitous Interactive Visualization of Large-Scale Simulations in Geosciences Over a Java-based Web-Portal


(1.)Minnesota Supercomputing Institute, University of Minnesota, Minneapolis, MN 55455, U.S.A.
(2.)Computer Science Institute, AGH University, Krakow, Poland
(3.) Laboratory of Computational Science and Engineering, University of Minnesota, Minneapolis, MN, 55455, U.S.A.

submitted to Concurrency: Practice and Experience, edited by G.C. Fox

Abstract

We have designed a web-based software system for real-time interactive visualization of results taken directly from large-scale simulations of 3-D mantle convection and other large-scale modeling efforts. This approach allows for intensive visualization sessions for a couple of hours as opposed to storing massive amounts of data in a storage system. Our data sets consist of 3-D data with over 10 million unknowns at each time-step to be used for volume rendering. Large scale interactive visualization on a display wall holding around 15 million pixels has already been accomplished. With the advent of the age of mobile web, we have now extended this tactic to hand-held devices, such as the OQO and Nokia N800, recently net-book computers, the iPHONE and the Android-powered system. We are developing web-based software in Java to extend the ubiquitous use of this system across long distances. The software is aimed at creating an interactive and functional application capable of running on multiple browsers by taking advantage of two AJAX-enabled web frameworks: Echo2 (http://echo.nextapp.com/site/echo2) and Google Web Toolkit (http://code.google.com/webtoolkit/). The software runs in two modes allowing for a user to control an interactive session or observe a session controlled by another user. Modular building of the system allows for components to be swapped out for new components so that other forms of visualization could be accommodated, such as molecular dynamics in mineral physics or 2-D data sets from regional convection models.

1. Introduction

The relentless drive in pushing larger and faster massively parallel computers into the market, coupled with the introduction of GPUs, are stressing the dire need to visualize and analyze the enormous amount of numerical data being generated. The recent announcement of Blue Waters IBM system by the supercomputer center NCSA at Illinois has presaged the dawning of the age of petascale computing in just 3 years' time. The coming challenges are indeed quite daunting with the petascale machine Blue Water's RAM having around 1 petabytes and disk space over 20 Petabytes. We can easily expect to see the tsunami of output coming from solving partial differential equations, which may range between one billion and even one trillion grid points. With this gargantuan amount of numbers one cannot hope to proceed in the traditional way of post-processing. There is definitely a need for a new strategy for visualizing such large data sets or else we will be drowned under the torrential data being generated and not
be productive at all. For this reason we have been engaged in a new approach in visualization, coined interactive visualization [1][2][3][4] in the past two years. A possible solution may be to interrogate interactively and visualize the solution in a collaborative manner, while it is running on a massively parallel system. In this way we can save a lot of post-processing time and decrease significantly the computational time because we do not need to write the massive data stream continuously to the disks.

Interactive visualization for small data sets is by no means a new idea but one that has been kicked around for awhile since the early 1990's . [5][6] and [7] have presented work on remote visualization using mobile devices and this trend has become more popular with the surge of Internet usage and increase in the bandwidth. In the past year researchers at the Laboratory of Computational Science and Engineering (LCSE ) laboratory [1] at the University of Minnesota working under an interactive visualization mode have now the capability of analyzing and manipulating in real time 3-D mantle convection simulations with 10 million degrees of freedom [2][3] and astrophysical simulations with close to 100 million degrees of freedom[1]. In the previous works [1][2][3] we have shown that the availability of hundreds of processors coupled with high-speed networks at 10 Gbits/sec and advanced volume rendering systems, combined with appropriate software lends to fast analysis of numerical data and increased levels of research productivity by means of a 15 million pixel power-wall ( see Figure 3 ).

In section 2 we present works similar to our own that have already been implemented. In section 3 we discuss the hardware that makes up the backbone of our visualization system. In section 4 we discuss the Java based modular components that sit in between the renderers and the users, who may use a phone device or other small display devices. In section 5 we show some preliminary benchmarks and examples of our web application used on current technology. In section 6 we discuss future applications, ideas, and developments related to our web-client-server paradigm, especially for mobile phones and net-books.

2. Related Works

Visualization in the volumetric format presented shares many attributes found in the works of others who have come before us.

OpenVisaar [8] presents a collaborative system for 3D visualization. It is built upon a visualization framework whose purpose is to promote collaborative 3D visualization through a client-server system, using a client written in Java to promote cross-browser compatibility. This system utilizes the GPU client-side through the use of OpenSG (Open Source Scene Graph) for all the rendering needs of the program. The server is used to pass off the information and facilitate the collaboration. The capabilities of the systems depend mainly on the hardware used, both CPU and GPU, but a moderate system can easily make use of this system (interactive frame rates rendering 1,000,000 particles or 64,000 streamlines on a 2.8 GHz Intel processor with an NVIDIA GeForce 6600 GT graphics card). It can be modified so that an entire cluster functions as the rendering engine.

COVE [9] , also known as Collaborative Object-oriented Visualization Environment aims at using collaboration through distributed and parallel computing. It is built upon the Distributed Object-Oriented Virtual computing Environment (DOVE), which is a virtual environment of interacting objects aimed and distributed and collaborative computing. Based on this environment, a wide array of different visualization applications can be created and plugged into
the system. The developers of COVE have created an application that plugs into COVE which visualizes through the use of ray casting. The system allows for either a client-server setup or a peer-to-peer option for collaboration.

In [10] another Java based system is created for visualization, with a primary focus in the medical field. The server uses C++ and OpenInventor to take data obtained from CT and MR scans and render images for either local or remote use. This system is similar to our own system in that it can do all of the rendering server side. However, it can also do some rendering client side. This would seem to be an optimal approach where the user only needs to access the necessary resources to do any visualization i.e. local rendering for simple tasks and server side rendering for more complex visualizations.

ScanView [11] is a remote visualization application built with focus on security from 3D reconstruction attacks. The system involves a client-server architecture in which both sides render images. The client is made up of OpenGL and wxWindows and handles low resolution versions of the visualization so as to give the user a smooth experience. Whenever the user stops manipulating the visualization, the client requests a high-resolution image from the server based on the current view of the client. The server also uses OpenGL but with a backend of a GeForce4 video card to handle requests.

RAVE [12], which stands for Resource-Aware Visualization Environment, is a system that has both a fat and thin client. The fat client, referred to in the paper as "active", receives a copy of the data set and renders locally. This client stays in communication with the server to receive any changes to the data set and for collaboration purposes with other users. The thin client is for use with any system that does not have the hardware to render locally, such as a PDA as is used in their demonstration. The thin client only receives images based on the input from the user. The PDA used in this research had to have a custom thin client built in C++ as opposed to the idea of using Java as was used for the fat client.

3. Glossary
   In order to assist the reader, we present below a glossary of terms used in this paper.

   **BOB** - Brick of bytes. The data output format that we use for volume rendering. In this format a single data voxel corresponds to a single byte within the "brick of bytes." The format was developed by Ken Chin-Purcell at Minnesota's Army High Performance Center in the early 1990's [13].

   **QDR** - Quad data rate infiniband. Quadrupled bandwidth of 4X SDR providing 40 Gbits/s.

   **DDR** - Double data rate infiniband. Doubled bandwidth of 4X SDR providing 20 Gbits/s.

   **SDR** - Single data rate infiniband. 2.5 Gbits/s per lane. A 4X link provides 10 Gbits/s of bandwidth.
BCast - A call in MPI which utilizes the multicast capabilities of the interconnect hardware to send data between hosts.

RPC - Remote Procedure Call. A programming pattern involving one computer program telling another computer program what procedure to run without the 2nd computer having to write the procedure in any special way.

JMS - Java Messaging Service. Messaging standard that allows application components based on the Java 2 Platform to create, send, receive, and read messages. It provides distributed communication that is loosely coupled, reliable, and asynchronous.

API - Application Programming Interface. Set of data structures, functions, classes and protocols provided by libraries or operating system to support building applications.

Netbook - A lightweight, low processing-power computer meant mainly for internet access and general purpose applications.

HVR - Hierarchical Volume Renderer. [14][15][16]

TCP - Transport Layer Protocol. One of the core protocols of the Internet protocol suite.

VPN - Virtual Private Network. Allows a user to access a network via an IP address other than the one that actually connects their computer to the Internet.

SWT - Standard Widget Toolkit. A graphical widget toolkit for use with the Java platform.

3. Current Hardware at the LCSE

In order to promote greater use of interactive scientific visualization, the Laboratory of Computational Sciences and Engineering at the University of Minnesota (http://www.lcse.umn.edu), better known as the LCSE, in the past decade under the directorship of Professor Paul R. Woodward, has developed and modified the rendering cluster to function also as a data pipeline. Fourteen visualization nodes and sixteen data processing nodes are used to operate LCSE’s PowerWall. (http://www.lcse.umn.edu) Figure 1 shows that the visualization nodes are Dell workstations with 8 Gigabytes of RAM, Dual 3.6 GHz Xeon processors, an NVidiaGeForce 8800 graphic board, and a locally attached four terabyte striped disk system. The disk system on these nodes can achieve data rates of 400MB/s reads and 200MB/s writes. The five data processing and visualization nodes are Dell workstations with 32 Gigabytes of RAM, Dual 3.0 GHz Quad core Xeon processors, two NVidia GeForce 8800 graphic boards, and a locally attached 32 terabyte striped disk system. Figure 2 shows a more detailed view of the cluster of super-workstations. The disk system on these sixteen nodes can achieve data rates of 1000MB/s reads and 1000MB/s writes. The data processing nodes receive data directly from the computational resources at the Minnesota supercomputing Institute. This data is transferred via a 4x DDR Infiniband fiber which makes the physical connection from the Infiniband fabric in LCSE to the Infiniband fabric based at the Minnesota Supercomputing Institute (MSI) of the University of Minnesota. Data can be written from any of 2048 SGI cores in MSI, directly from the running simulation code at 2GB/s, to any data processing node in LCSE. Over the past year...
the LCSE has also been building a Cell and multi-core Intel based computational resource. This resource, which will be installed by spring 2009, consists of six IBM Cell "Tri-Blades" each one containing two QS22 IBM Dual Cell blades and one LS21 Quad Core AMD host. The Cell hardware is supplemented by eight Intel "super-workstation" nodes, each of which contain 4 quadro-core Nehalem processors, 64 Gbytes of RAM and 48 Tbytes of local disk memory. This entire system of super-workstations can easily provide nearly one teraflop of computational performance in an on-demand fashion. One does not need to wait. This system would be attached to the visualization nodes via a fully interconnected 4x QDR Infiniband fabric. Having this type of dedicated resource would provide for interactive supercomputing without the delays inherent in reserving a shared resource for a specific time. This system would also increase, by a factor of 32, the available bandwidth between the location of the computation and the visualization nodes. This increased bandwidth would allow for more frequent and higher quality samplings of the current state of a given large-scale run involving more than 100 million degrees of freedom, or around 512 cubed for a single-phase fluid flow problem.

Once received, the data is processed into the desired visualization format. This conversion involves transforming the data from raw BoB formatted data into a HVR file [11]. The HVR file is broadcast across the LCSE 4x QDR Infiniband fabric to the visualization nodes at 4GB/s. From here, visualization can be carried out either on the Power Wall (see Figure 3), or through a remote client via a web-portal interface based on our web-client-server paradigm. This subject will be discussed below much more thoroughly.
Figure #1 depicts a detailed view of the current hardware involved in our interactive visualization Power-wall system at the LCSE. The raw data originates from either an external computational resource from MSI or Teragrid centers or the local cluster based on multi-core processor with Nehalem quadro-core or IBM-CELL processors. After having been re-formatted, the resultant HVR files are broadcasted to all the rendering nodes involved in the volume rendering of this particular dataset. This processed image can then be displayed on the PowerWall or sent via the internet to a remote display.
Fig. 2# A more detailed view of the super-workstation cluster to be used for on-demand interactive visualization and computing.
4. Web Access Enabling Java Components

Since the Power-wall only offers interactive visualization access to a limited number of users, in the past year we have been focusing on developing software to allow for a wider audience of interactive visualization. To allow for ubiquitous access of large-scale visualizations we choose to use the Web as our medium. Creating our application to work as a web application allows anyone with Internet access to use it. This access is enabled in our system through three modular components, one of which is the front-end web application. These three components together use combinations of Java, JavaScript, AJAX, XML, SQL, and CSS to allow for a robust and secure system.

4.1 Web Service Overview

The visual exploration of large three dimensional datasets requires the use of a large cluster of machines in order to explore the data in real time. This is mainly due to the time required to volume render a single frame of the dataset for a specific timestep. The volume rendering technique mainly employed in the lab is a texture mapping technique in which the dataset is split up into slices and each slice is then rendered in a back to front fashion. This technique gives realistic results and can be hardware accelerated using common off-the-shelf graphics accelerators such Nvidia Geforce based graphics cards. High quality visualization requires many slices of the dataset to be rendered for each frame.
To allow for real time exploration of the data, a rendering cluster is employed to render a single frame of the dataset. A single frame is rendered by splitting up the image and requesting each node in the cluster to only render a specific portion of the image. The rendering nodes should each have the entire dataset on its local disk as the system expects the data to be pre-replicated locally to facilitate the rendering rates needed for smooth exploration. The results are then collected and stitched back together for display.

Custom built software is exclusively employed in the LCSE mainly for performance reasons. Each node of the rendering cluster runs a server application called lcse_serv. Lcse_serv is a modular, single threaded server application that accepts TCP connections and routes them to specific handler modules that implement the specific protocols needed for each connection. Along with providing an entry point in the system, lcse_serv also provides facilities to start, stop, and communicate with processes on a cluster node that individual handler modules require for their specific operations. Due to the module nature of the system, future applications can be very easily incorporated into services which the cluster can provide.

This system was initially developed for exclusive use inside of the laboratory with a high speed Infiniband connected internal network. With such an environment driving the initial design, it was natural to have the controlling application divide up the rendering work, send the request to each rendering node individually, collect the resultant images from each node and stitch the image back together for display. Performance being the main issue, this design benefited from not requiring a layer of redirection as the controlling application issued request directly to individual nodes in the cluster. This data flow provides very high performance for exploring dataset inside of the laboratory infrastructure.

Although the design of the system was not built with the expectation of off-site access to rendering servers being desirable, it was soon discovered that such access would be beneficial in certain situation. To enable such off-site visualization sessions to occur within the limitation of the previous system, a virtual private network (VPN) was set up into the lab and one connection per rendering nodes was tunneled through it, as shown below in figure 4.
This contrived configuration does work with great strain but is very fragile because of the large number of connections needed to be maintained for each user utilizing the rendering cluster. This method would also never scale to more than just several users. A more stable and scalable system has been implemented by creating a rendering service that contains all the functionality of communicating with the rendering servers that previously only existed within the controlling application. This service can then be used as a back-end rendering service for a variety of different applications, from a graphical user interface to an AJAX web interface and can also be easily incorporated in exciting future application.

The rendering service which we have built allows for a variety of applications to be built that enables a user to utilize the entire cluster of rendering nodes and internal applications we have in the lab. Such applications can be very lightweight on the user’s system as most of the processing and communication overheads are occurring between the rendering service and the cluster. We have envisioned a volume rendering application that is capable of rendering large amounts of volume data in real time and can be displayed very easily on many different types of hardware, from very powerful desktop machines to web enabled handheld devices.

Our prototype service has been implemented using Java and XML technologies. The rendering service is implemented in Java and can be run under a Java Servlet technology enable web server, such as Apache Tomcat. Two prototype AJAX web applications which make use of the rendering service have also been implemented, using two different frameworks; Echo2 and GWT.
By making use of an intermediary rendering service, the total number of connections needed by each instance of the application is essentially reduced to one. The rendering service will only make one connection to each rendering server to be multiplexed by the multiple users of the service. When the rendering service receives a request for an image to be rendered, it makes all the necessary connections needed to the individual rendering machines, splits up the job into separate chunks to be rendered in parallel, collects the results, and stitches the finished image back together for delivery. Requests are made on the service utilizing the standard web protocols – XML, HTTP and TCP/IP. The resultant image is base64 encoded and embedded in a standard XML response.

![Diagram](image)

Figure 5. Schematic diagram showing the hierarchy of our visualization rendering web service under an Ajax interface using Java Technologies. We stress here the simplicity of the implementation as compared to the use of the VPN protocol shown in figure 3.

The diagram in figure 5 shows the interaction between the web server running the web interface, the web server running the rendering service and the rendering cluster. The rendering service should be physically located on the same local area network as the rendering servers to eliminate the need for multiple connections to be tunneled into the private network for multiple destinations. The only internet accessible device required is now the server running the rendering service. This system can also allow developers the freedom to develop their own application in house that makes use of the rendering cluster and technologies developed at the LCSE laboratory.

4.2 Overview of Web Client-Server Paradigm
We have employed the web client-server paradigm for carrying out our web-services activities. The Client-Server combination comprises the other two components in our system. The user must access the web client. We currently have two working Client-Server models that come from two different web frameworks: Echo2 and Google Web Toolkit (GWT) (http://code.google.com/webtoolkit/). Similar in design and features, these two models differ in the mode of client-server communication, among other areas. Both frameworks are built around the idea of being coded in Java but then translated to JavaScript so dual progression of each application is not difficult. Release of the new framework Echo3 (http://code.google.com/webtoolkit/) may allow for new and different things not available right now to either Echo2 or GWT but a stable release is not yet in sight, so we will continue with these two aforementioned tools.

An example that applies to both programming models is as follows: The GUI allows the user to adjust various visualization options such as their viewing angle and the positioning of the data being viewed. These interactions are captured by the browser and sent back to the server for processing. Based on the user's interactions, the server modifies an XML document full of properties of how to render the image. This document is sent via HTTP to the web service where it is parsed and information is extracted that tells the service which render servers to access and the image properties to be used when visualizing the data. The service handles splitting up the parts of the image to be rendered and making the calls to the servers to be rendered in parallel. Upon completion of the rendering, the service pieces the image parts back together and encodes the byte array into a base64 string. This string is embedded in an XML document that is sent back to the server that passed the initial XML document. The server parses this XML and decodes the base64 string so that it can be created as an image. Once the server has the image, the client requests it and it is sent to be displayed for the user. This process repeats many times during a visualization session until the user wants to stop, at which time they can save their current set of image properties for upload into later visualization sessions. This sequence is summarized in Figure 6.
4.2.1 Echo2 based Web Application

The Echo Java web framework was our first choice for implementing a user interface. This library is an open-source software distributed under Mozilla Public License. Echo2 - the current stable version of Echo, offers a well documented and rich API for building AJAX-enabled, robust web applications. Echo stands out from the rest of web frameworks as an easy to use and component-oriented platform, following MVC (Model View Controller) architectural pattern. The web development process in Echo looks like a stand-alone application development, using Swing or SWT, and is therefore well-suited for building prototypes, as the implementation can progress really fast. Nevertheless, our main concern in using Echo was that it allows us to write thin-client applications and take advantage of its rich set of components – the vital feature in the creation of a modern-looking, interactive web user interface. The thin client approach was chosen because of the limited computational and storage resources on mobile devices. However, as we discovered later, some part of processing can be done on the client-side, resulting in less bandwidth demands.

Figure #6. Shows the connections and program flow for our web application. Generically are shown both Echo2 and GWT, with the only difference resulting in Client-Server communication.
To provide the user with a convenient edition of all parameters of image rendering request, we designed and implemented two custom components for color map selection and alpha selection (see Figure 1). Echo supports the creation of custom components either by combining available components or by using special low-level API (JS coding required). The implemented GUI works well on many types of web browsers (Firefox 2.x.x, Firefox 3.x.x, Opera 9.5+, IE 7.0, Maemo Web Browser – Nokia N800). We benefit from the high-level approach of Echo, not bothering with XML or HTML but at the same time we lose some part of client-server communication control, as it is embedded into the platform.

After creating the web GUI which allows for adjusting all image parameters and user authentication, we then move into performance issues to provide appropriate response times, the crucial issue of soft-real time tasks such as interactive visualization. Using Echo GUI we tested response times in the following scenario. Tomcat web container serving MantleVis application was set up at the UoM (connected to rendering service via Fast Ethernet) while the client was making requests from Poland (AGH UST). For JPEG encoded image of size 1024x768 , we obtained average response time around 1 second, which is acceptable. However, our aim is to decrease this response time as much as possible to provide the user with more smooth, movie-like feeling for the interactions.

During the tests we discovered some framework-related issues that pushed us into testing another platform (GWT). First, in browser-container AJAX HTTP communication, StyleSheet information is sent multiple times increasing bandwidth requirements. We also learned that a part of processing (e.g. operations on image request data) can be done client-side decreasing the number of client-server synchronization messages and without the need of much processor power. The next release of Echo (version 3) will provide thick client implementation mode and also resolve performance issues. Still, the stable version has not been published yet, therefore we decided to test the framework from different vendor.
Figure #7 This shows the color bar component as added to the Echo2 web application. A similar version is present in the GWT web application. The top bar is a preview of the transparency and the bottom bar shows the color. The table allows for adding of new "color sticks" and adjusting previously added color sticks. When the user clicks on the mini color swatches, a pop up window displays a full color picker. A separate page controls the adding and removing of "alpha sticks" that control the transparency.
4.2.2 Google Web Toolkit based Web Application

Here we lay out the details of the Google Web Toolkit (GWT) that are pertinent to our visualization system.

This web framework departs from the traditional fat server approach by taking Java code and compiling it into JavaScript through a special compiler. This allows us to do two things. The first is the use of efficient JavaScript that would be difficult to code manually; doing the actual coding, debugging, and maintenance in a higher level language. The second is using this JavaScript to harness the processing power of the browser to do simple tasks; reducing server requests and application slowdowns related to those requests. However, this is a double-edged sword because it means Client-Server communication is developer driven and less robust than in typical Fat-Server web frameworks (Echo2 for example).

The special compiler, created by Google, has a list of supported Java types, classes, and functions that it knows how to convert into JavaScript equivalents. It is only used to compile JavaScript for code to be run client-side; server side can be any back-end you want. We have opted to use a Java back-end with classes provided by the GWT project to facilitate communication with the client. This compiler can also optimize the JavaScript to run in an efficient manner client side. The default mode is called "Obfuscate" and it creates JavaScript
used for production use; hardly readable by common coding standards. However, there are also two other modes that do create the JavaScript in such a way that it could be examined and debugged easily. With each new update of GWT, the compiler is updated with more efficient forms of JavaScript conversion.

Although it does offer support for XML and JSON, the communication mechanism we used through GWT is called Remote Procedure Call (RPC), which has been around for awhile. This method involves invoking procedures on the server from the client and vice-versa. A typical call involves the client calling a procedure server side, along with what is known as a callback; a special procedure client side. The server processes the procedure indicated and then looks at the callback and makes an RPC call to the client, telling the client to execute whichever procedure was picked at the callback. However, all of this occurs asynchronously, meaning the client makes an RPC call to the server, but then continues execution. This allows for the client to make an RPC call to the server, and then both concurrently work on whatever tasks they have.

To deal with the different quirks associated with different browsers running the JavaScript, the compiler creates a version of the JavaScript file tailored specifically for each browser. However, each version is not sent to the client-side when the application is run. GWT uses a concept known as "Deferred Binding" where in only the version of code to be used is sent client side and loaded. This allows for cross-browser compatibility but doesn't sacrifice it through the bloated JavaScript. GWT itself generates the rules and code to determine which browser a client is using and which version to send them. As a developer, you don't handle any of this. However, if you create a widget and discover that it handles differently on various browsers, you can then write a version of that widget for each browser and then the code to allow for Deferred Binding to take place.
Figure #9. A more detailed view of the GWT web application. The "Server-Session History" is used by observers to view the images (see figure 9 below showing observer mode). This is currently done through custom implementations. We plan to implement ActiveMQ (JMS API) in the future.

```java
public void getNextImage()
{
    Date newDate = new Date();
    String imageURL = GWT.getModuleBaseUrl() + "ImageServer";
    String query = "?" + "image" + "+" + newDate.getTime();
    String url = imageURL + query;

    image.setUrl(url);
}
```

Figure #10 Snippet of a code taken from the GWT Application. This procedure extracts the next image from the image server.

4.3 Observer Mode for collaboration

To allow for more than one person to view interactive sessions and to facilitate group collaboration, we have created an observer mode. Once a user has started a session, the history of all the images they have viewed is stored on a separate server. During this session, another user can connect to our visualization system and view this history of images. This mode is currently in development and only allows for chronological viewing of the images (i.e., you cannot go backward). If an observer catches up with the controller, who started the session, their web application must wait for more images to be sent to the session history server. Therefore, if an observer logged into our system just after a controller started a session, they would be able to have close to the same view as the controller, the only other variable being the network latency.
5. Benchmark and Examples

With the development of two different web applications, we have created some simple benchmarks to test the capabilities of each application. These only serve to give us ideas for the future when more of the system is developed and completed; at which time we will conduct more thorough benchmarks and tests.

Table 1. Results of applications running through Firefox and Internet Explorer. One client was residing at the Minnesota Supercomputing Institute in U.S.A. (Laptop 1) and the other connected from Krakow, Poland (Laptop 2). Shows elapsed time between a button clicked and the resulting image being loaded.
Current trends in technology are leading to more hand-held devices capable of browsing and using web based systems. Older technology, such as the Nokia N800 and OQO, were bulkier, had problems with short battery life, and the touch screen interface was not as well developed. Netbooks because of their low costs are fast becoming very popular. Both the Nokia and the OQO have been discussed by us already [2][3]. The newest generation of devices includes the iPhone which has 3G capabilities and makes mobile-web feasible and T-Mobile G1 that incorporate more advanced technology than previous mobile devices such as their own internet connection and fully functional web browsers operated by touch screens. In addition to these things, the T-Mobile G1 phone implementing the Google Android OS also has a full QWERTY keyboard that flips out of the phone. The disadvantages to the iPhone are that the screen is still small compared to some other handheld devices, the internet connection is slow compared to high speed connections at home, and the interface is slightly different. The G1 phone is at a disadvantage to the iPhone in that its touch screen is not as sophisticated.

Both the iPhone and the G1 interfaces allow for unique ways to interact with data. Both of these devices use a touch screen for the interface method, while only the iPhone allows for a multi-touch interface. With an iPhone users access the web portal by using the Safari web browser provided with the device. The G1 uses a web browser known as Web-Kit which is a simplified version of Google's browser Chrome, although it has been mentioned that future versions of the phone will use the Google browser. Both phones have a small screen, but the touch interface allows for users to more easily zoom in and control the web portal. Both phones also allow for both a vertical and horizontal view of web sites. When viewed horizontally it is easier to see both the interface and the visualized image, and the vertical layout allows for an easier interface to work with the web portal options then scroll over viewing a larger visualized image. It is in horizontal mode that the QWERTY keyboard becomes usable for the G1. Certain aspects of our web application depend on the screen size, which changes as the user flips the phone between horizontal and vertical mode. Measures to account for this change neither have yet to be implemented nor tests to see if such changes are necessary. They will be carried out as the application reaches a more stable release.

Using handheld devices such as the iPhone to access the web sites can be difficult in terms of the innate technical capabilities of such devices. Our GWT version of the application is

<table>
<thead>
<tr>
<th>Browser</th>
<th>Framework</th>
<th>Laptop 1</th>
<th>Laptop 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firefox (3.0)</td>
<td>Echo2</td>
<td>0.96</td>
<td>1.19</td>
</tr>
<tr>
<td></td>
<td>GWT</td>
<td>0.86</td>
<td>1.19</td>
</tr>
<tr>
<td>Internet Explorer (7.0)</td>
<td>Echo2</td>
<td>0.95</td>
<td>2.08</td>
</tr>
<tr>
<td></td>
<td>GWT</td>
<td>0.89</td>
<td>1.51</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Browser</th>
<th>Framework</th>
<th>Average (sec.)</th>
<th>Standard Deviation (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firefox (3.0)</td>
<td>Echo2</td>
<td>0.96</td>
<td>0.10</td>
</tr>
<tr>
<td></td>
<td>GWT</td>
<td>0.86</td>
<td>0.07</td>
</tr>
<tr>
<td>Internet Explorer (7.0)</td>
<td>Echo2</td>
<td>0.95</td>
<td>0.07</td>
</tr>
<tr>
<td></td>
<td>GWT</td>
<td>0.89</td>
<td>0.06</td>
</tr>
</tbody>
</table>
a fat client and currently is too demanding for real use except in the case of a netbook, such as the Samsung NC10 (see Figure 12), which has 2 Gbytes and a 1.6 GHz CPU. Attempting to create our own iPhone native application would not solve this issue. A native iPhone application would only allow us more access to certain features of the iPhone such as the camera or the GPS, neither of which is helpful to our situation. The only viable options are to decrease the memory strains of the GWT application or to focus our efforts on the newest Echo framework once it is released. Using the newest Echo framework would allow for a thin client approach that should be more usable for handheld devices. The GWT would then only be used by more powerful computers, such as a net-book.

Figure #12 The GWT version of our web application, on a 3G iPhone in horizontal mode, running through Safari web browser. The user can use their fingers to zoom in or out as needed. This allows for easily manipulation of the controls along with good viewing of the visualization.
Figure #13 The GWT version of our web application on a T-Mobile Android phone in horizontal mode. The browser used by the phone is Web-Kit.
6. Discussions and Future Perspectives

We plan to develop an extension to the work described in this paper by applying web-based, interactive visualization approach to other types of simulations. In particular, molecular dynamics simulations are an immense/inexhaustible source of information-rich, high-volume data that requires robust rendering frameworks. Recently, highly effective and powerful scheme for atomistic visualization of disordered systems was proposed [17][18]. It provides a rich set of space and time varying data visualization modes ranging from coordinate-based atom pathlines to advanced methods aware of additional information extracted from simulation results such as radial distribution functions. The OpenGL standalone Windows application, presented in [18], is
capable of bringing deep insight into data by rendering coordination environments, bonding, atom clustering and structural stability over time. In close cooperation with team led by Professor B. Karki from Louisiana State University, we are going to develop new, distributed approach for atomistic visualization. Our motivation is to make MD rendering engine accessible for a wider audience and interoperable. The present, single machine – single user, on-fly data analysis is sufficient for a small number of users but in order to avoid the need of distribution of multiple standalone application copies (multiple users scenario) and get rid of rendering engine platform dependence, we adopt the scheme derived from interactive mantle visualization. Providing usability in case of limited storage and computational resources (mobile devices) is also under our consideration. The diagram showing the essential steps planned is depicted in Figure 14. First, the existing standalone application has to be adjusted to support off screen batch rendering, invoked via a Web Service interface. This interface should provide the whole set of visualization modes available at existing application. We are going to take advantage of gSOAP and Axis Web Service libraries to implement MD Rendering Service (C++) – Web Application (Java) communication. As in case of interactive mantle visualization, we are going to use Echo or GWT to build end-user management GUI. The database will be used to store and load simulation results. In the presented three-layer system, the user can interactively manipulate simulation data from different platforms including mobile devices. The web-service-published rendering engine can be used by applications written in different programming languages.
Figure #16 The overview of Interactive Atomistic Visualization system. The first layer contains off-screen rendering engine whose functionality is exposed using Web Service (gSOAP). The second layer is a web application that enables user to interact with simulation data. The front-end of the system is a web browser.

This application provides high levels of both usability as well as portability. While these features have not yet been fully implemented into the program, we are already capable of using a script to take a standard BOB file, converting it to a HVR compatible file, and then connecting the file with the application. This streamlines the procedures as well as limits the amount of post
processing necessary to visualize a dataset. When the system is finished, this application will be able to notice when a run has completed and notify the researcher. The data will have been post processed so that it is accessible by the application. The user will log in to the application, interact with the visualized data and either choose to save the data as a good run or discard it immediately.

We have also taken into consideration that different problems will have different solutions. To this end, our application will be modular. Our visualization engines will only require that the understand functions from the service that control camera position and object position. In addition, they only need to be able to return an image. From there, any visualization software that can be wrapped with these requirements can be used with our system.

Figure 17. Large-aspect ratio two-dimensional run with over 500,000 unknowns. This work was done by Elena Sizova [19]. Two dimensional equations do not require the same amount of visualization and computing power as big 3-D runs. Therefore, it may be wasteful to use the same rendering engines for 3-D data as for 2-D.

To demonstrate the usefulness of this notion, we will use the example of a simulation of a two dimensional subducting slab equation (see Figure 17). The governing equations for mass, energy and momentum can be solved and be visualized using Matlab. There is no need to wait in queue for a run. A user will be able to choose Matlab as the visualization engine, enter the equation that needs to be solved, and the application will return the resulting figures. The only storage used in this process is that used by Matlab itself, with no unnecessary disk space used on preprocessing, processing, or post processing. By being able to choose the visualization engines, via a module system, users will able to optimize the application for their specific datasets.
Figure #18  With a 3-G phone, or a net-book, the user (girl in red) connects to a web client, wherein they can choose a particular rendering package. Currently, we are developing interactive rendering with Amira or Avizo visualization package, Matlab, and HVR (Hierarchical volume rendering). After selecting a rendering engine and a dataset, the user's choices for the different visualization packages are passed over to the Web Server. The Web Server load balances the incoming requests and distributes each request to a computational node. At the computational node, a dataset is generated with specified input parameters. This dataset is passed to the rendering nodes, which uses package specified by the user to generate a figure. This process will be repeated many times each minute. Thus a movie can be made in this way.
In this article we have demonstrated that the web-client-server paradigm can significantly facilitate the user to carry out interactive visualization and analysis of large-scale numerical simulations effortlessly over the internet. This scenario is aptly shown in Figure 19, which shows that with extensive wireless coverage one can carry out interactive visualization almost anywhere. As absolutely every computer in our lives, such as a television or an in-dash computer in our car, is being connected at every step of the way to the internet, this relentless trend will further allow the user to have ubiquitous access for monitoring computational results. Our work, based on the web-client server paradigm, will pave the way to interactive supercomputing over the internet. The system is still a bit crude, but we are working constantly to improve it and hope others will join us in this open venture.
Acknowledgments
-----------------------

We thank discussions with Paul R. Woodward, David H. Porter, Megan R. Damon, M. Charley Kameyama, B. Karki, Boris Kaus and Gordon Erlebacher. We are grateful to Marina Shukh for helping us with the Google Android device. This research has been supported by the VLAB grant from the National Science Foundation and an equipment grant from the National Science Foundation.

References
----------


