RAVE: Resource-Aware Visualization Environment

Ian J. Grimstead[*†] Nick J. Avis[†] David W. Walker[†]

Abstract

We present an architecture for remote, collaborative visualization of datasets over the Grid. This permits an implementation-agnostic approach, where different systems can be discovered, reserved and orchestrated without being concerned about specific hardware configurations. We illustrate the utility of our approach to deliver high-quality interactive visualizations of datasets (circa 1 million triangles) to physically remote users, whose local physical resources would be otherwise overwhelmed.

1 Introduction

Grid computing, based on service-oriented architectures such as OGSA [7], permits users to remotely access heterogeneous resources without considering their underlying implementations. This simplifies access for users and promotes the sharing of specialised equipment (such as rendering hardware).

Visualization is also an emerging field, taking advantage of more powerful rendering capabilities. Such rendering hardware is highly specialised, enabling users to visualize extremely complex datasets. This hardware is often expensive and placed in a central location to be shared by users.

The motivation behind the Resource Aware Visualization Environment (RAVE) arises from the desire to both investigate how interactive services can be supported within a Grid infrastructure and to break the constraint on physical co-location of the end user with the high capability visualization engines.

2 Previous Work

We now review various remote visualization applications, paying particular attention to Grid-enabled applications.

OpenGL VizServer 3.1 [10] enables X11 and OpenGL applications to be shared remotely, with machines that do not have specialist high-performance graphics hardware. VizServer renders the OpenGL remotely, and transmits the rendered frame buffer to the collaborator. OpenGL applications display on all collaborators machines, with one user at a time being in control of the application.

COVISE [12] is a modular visualization package, where one user is the master with control over the entire environment, whilst other users are slaves that can only observe. COVISE takes advantage of local processing on the slaves by only transmitting changes in data.

The RealityGrid project has implemented a collaborative Problem-Solving Environment (PSE) [11], where a central server coordinates the available resources. Output data is rendered remotely (receiving raw data from the simulation), with the final frame buffer being sent back to the client.

The e-Demand [6] project is implementing a PSE on the Grid, where each module in the environment is represented by an OGSA service. Multiple rendering services or multiple presentation services can be deployed, to form a collaborative environment.

IRIS Explorer (a Modular Visualization Environment) continues to be used for collaborative research, such as the gViz [4] project which aims to Grid-enable the COVISA collaborative visualization tools.

The SuperVise [8] project investigates the use of Grid technology for visualization, where phases of the visualization pipeline (such as data filtering, geometry transformation) are distributed on different hosts for execution.

Current visualization systems often make assumptions about the available resources; for instance, COVISE assumes local rendering support, whilst OpenGL VizServer relies totally on remote resources. We wish to produce a system that can make use of available resources, either local or remote, and react to changes in these resources. To this end we have created the Resource Aware Visualization Environment.

[*]Welsh e-Science Centre (WeSC), Cardiff University, Newport Road, PO Box 916, Cardiff CF24 3XF

[†]School of Computer Science, Cardiff University, Queen's Buildings, Newport Road, PO Box 916, Cardiff CF24 3XF

3 Architecture

To implement our visualization system, we have four components: a data service, a render service, an active client and a thin client; this architecture is presented in Figure 1. Our data and render services are implemented as Grid/Web services, enabling us to advertise the availability of resources through WSDL and UDDI. Users can then automatically search for supporting resources and access them remotely, rather than having to co-locate with the resource.

3.1 Data Service

The data service imports data from a static file or web resource (at present Lightwave, Wavefront or VRML format). This data repository forms a a central distribution point for the data to be visualized. Multiple sessions may be managed by the same data service, sharing resources between users. RAVE supports "bridging" services, forming a direct link to an external application. This enables RAVE to present a collaborative version of any external dataset. It must be noted that there is no commonly agreed standard for visualization data; hence the need for multiple file formats and bridging software. We are currently discussing this with Manchester Materials [13].

3.2 Render Service

Render services connect to the data service, and request a copy of the latest data. The data service keeps the render service up-to-date with any changes, using network bandwidth-saving techniques such as multicasting. This enables multiple render services to simultaneously visualize the same data for little extra overhead.

A user can then interact with the shared data when the render service displays at the local console. This permits large devices such as a FakeSpace Immersadesk R2 to be used, along with commodity rendering hardware. As render services contain a full scene graph, a view may be rendered in stereo or mono as required without additional information from the data service.

Changes made locally are transmitted back to the data service, propagating to other members of this collaborative session. The render service can also render off-screen for remote users, utilising available rendering resources. Multiple render sessions are supported by each render service, so multiple users may share available rendering resources. If multiple users view the same session, then a single copy of the data is stored in the render service to save resources.

3.3 Active Client

Where a user cannot install a local web service, or if the available render services do not support off-screen rendering, an active render client can be used instead. This connects directly to the data service, rather than via a render service. This enables us to support a wider range of users and hardware.

3.4 Thin Client

Whereas the render service is a render-capable client for the data service, a thin client (such as a PDA) represents a client that has very modest local rendering resource. The thin client must make use of remote rendering, so it connects to the render service and requests off-screen rendered copies of the data. The local user can still manipulate the camera and the underlying data, but the actual data processing is carried out remotely whilst the local client only deals with information presentation. This approach can be used by any client whose local resources would otherwise be swamped with data (for instance, a slow desktop PC or a PDA).

4 Implementation

We now discuss the important parts of our implementation as Grid/Web Services, highlighting areas that may be of use or concern to fellow implementors.

4.1 Data Transfer

Grid/Web services are used for the initial handshake and data request, when we then drop down to TCP/IP sockets for fast direct transfer of live data. The Globus Toolkit 3 (GT3) [9] Java API does not support memory-to-memory data transfer, instead using an enhanced FTP protocol to transfer files. As we wish to send a continuous stream of live updates, we must resort to direct TCP/IP socket communication.

4.2 Language and Graphical API

We have selected Java as our implementation language, as there is plentiful support for Grid/Web services and is cross-platform. The only cross-platform rendering API available for Java is (at present) Java3D, so we have to use

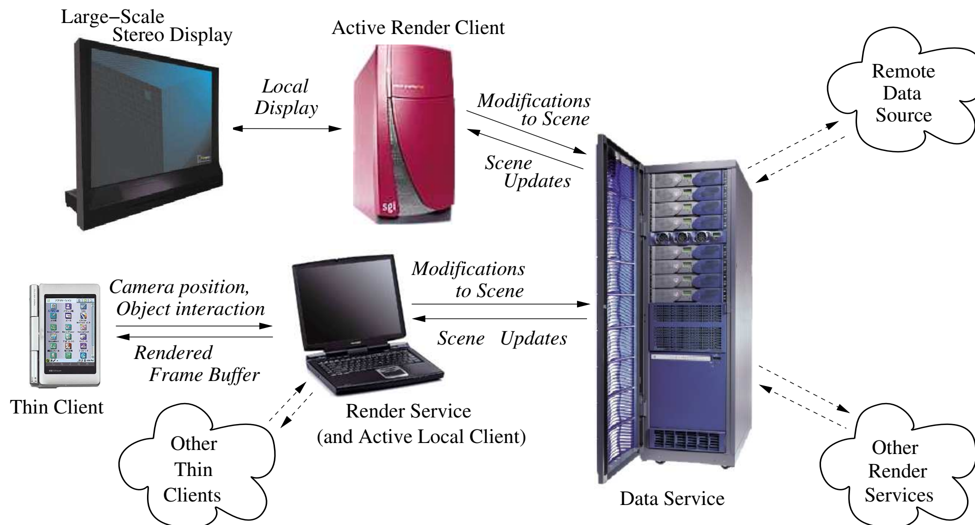

Figure 1: Diagram of RAVE architecture

this. Note that Java3D is reaching end-of-life, with a port of OpenGL to Java having been announced by SGI and Sun. With this in mind, we have created a simple interface and an implementation that wraps Java3D; this hide the actual graphics API we are using, in readyness for a later change (possibly to OpenGL).

4.3 Grid/Web Services

We initially used GT3, but with the announcement that the OGSA model is to converge with the WS-Resource Framework [3], we have refactored our implementation to also support Web Services (WS), by creating implementation-specific wrappers around our core "engine".

We are using Apache Axis [2] in a Jakarta Tomcat [1] container, and make use of the various tools provided to automate as much of the build process as possible (such as java2wsdl to generate the WSDL document).

4.4 PDA Support

We selected a Sharp Zaurus PDA, as it runs Linux and has a high resolution screen (useful for visualization). This meant that all of our devices were running a flavour of UNIX (Solaris, IRIX and Linux), reducing the maintenance overhead.

Our test PDA only supports a subset of J2SE (J2ME), we cannot use our current Java client. As we use WSDL to advertise our data and render services, we are not restricted to a particular platform or language so use Java for the services, and C/C++ for the PDA client.

GT3 was overly complex to compile on a PDA (requiring over 200Mb of intermediate storage), and had various issues with the C compiler on the PDA. Instead, we use the C++ variant of Apache Axis, using tools to create C++ stubs and skeleton directly from the WSDL document.

5 Initial Timings

The render service was run on a Toshiba laptop, with a GeForce 420 Go hardware graphics accelerator. This was selected for ease of testing and portability for demonstration. A 802.11b wireless card was used on the PDA, which contacted the render service on the laptop via a wireless access point. Two test models (a skeletal hand, ∼835kpoly, and a skull, ∼475kpoly) were obtained from the Large Geometric Models Archive at Georgia Institute of Technology [5].

A 200x200 true colour (24 bits per pixel) image was transmitted, with timings presented in Table 1 and a view from the PDA presented in Figure 2. Total latency represents the time between initiating a web services call and an image being shown. The time to receive the image on the PDA is also shown, along with the time taken to render it on the laptop. It can be seen that the render time is dominated by the image transmission (approximately 60%). This initial implementation of the RAVE architecture only sends uncompressed images, so we are limited by network bandwidth.

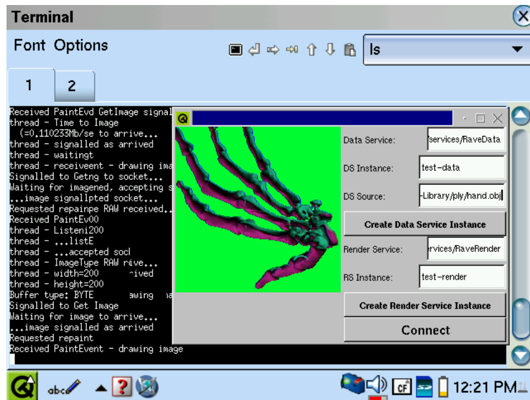

Figure 2: PDA screen dump: Skeletal hand

Table 1: Timings for 200x200 pixel image

Model Name	Total Latency	Image Receipt	Render Time
Skeletal Hand	0.339s	0.201s	0.091s
Skull	0.287s	0.194s	0.062s

6 Discussion

Given the announcement that the OGSA model is to converge with the WS-Resource Framework [3], we have refactored our implementation to also support Web Services (WS), by creating implementation-specific wrappers around our core "engine". This has enabled the implementation of a light-weight PDA client using WS.

Our implementation fits in between COVISE and OpenGL VizServer, where RAVE can make use of local rendering hardware if present, otherwise dropping back to use a thin client where the full frame buffer is transmitted. RAVE is also unusual in the range of hardware it supports (from a PDA to a tracked ImmersaDesk).

References

[1] Apache Jakarta Project—Tomcat. `http://jakarta.apache.org/tomcat/index.html`.

[2] Apache Web Services Project—Axis. `http://ws.apache.org/axis/index.html`.

[3] Grid and Web Services Standards to Converge. Press release at `http://www.marketwire.com/mw/release_html_b1?release_id=61977`, GLOBUSWORLD.

[4] gViz - Visualization Middleware for the Grid. `http://www.visualization.leeds.ac.uk/gViz`.

[5] The Large Geometric Models Archive. `http://www.cc.gatech.edu/projects/large_models/`, Georgia Institute of Technology.

[6] Stuart M. Charters, Nicolas S. Holliman, and Malcolm Munro. Visualisation in e-Demand: A Grid Service Architecture for Stereoscopic Visualisation. In *Proceedings of UK e-Science All Hands Meeting 2003*, September 2003.

[7] Ian Foster, Carl Kesselman, Keffrey M. Nick, and Steven Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Technical report, Globus, February 2002.

[8] James Osborne and Helen Wright. SuperVise: Using Grid Tools to Support Visualization. In *Proceedings of the Fifth International Conference On Parallel Processing and Applied Mathematics (PPAM 2003)*, September 2003.

[9] Thomas Sandholm and Jarek Gawor. Globus Toolkit 3 Core - A Grid Service Container Framework. Globus Toolkit 3 Core White Paper, July 2003.

[10] SGI. SGI OpenGL VizServer 3.1. Data sheet, SGI, March 2003.

[11] J. Stanton, S. Newhouse, and J. Darlington. Implementing a Scientific Visualisation Capability Within a Grid Enabled Component Framework. In *8th International Euro-Par Conference, volume 2400 of Lecture Notes in Computer Science*, August 2002.

[12] High Performance Computing Centre Stuttgart. COVISE Features. `http://www.hlrs.de/organization/vis/covise/features/`, HLRS, September 2000.

[13] Kevin T. W. Tan, Greg Johnson, Nick J. Avis, and Phil J. Withers. Towards a Grid Enabled Engineering Body Scanner. In *Proceedings of UK e-Science All Hands Meeting 2004*, September 2004.