

Automatic Distribution of Rendering Workloads in a Grid Enabled Collaborative Visualization Environment

Ian J. Grimstead, Nick J. Avis & David W. Walker
[I.J.Grimstead|N.J.Avis|D.W.Walker]@cs.cardiff.ac.uk
School of Computer Science, Cardiff University
Queen's Buildings, Newport Road, PO Box 916
Cardiff, CF24 3XF, United Kingdom

ABSTRACT

This paper presents a distributed, collaborative grid enabled visualization environment that supports automated resource discovery across heterogeneous machines. Our Resource-Aware Visualization Environment (RAVE) runs as a background process using Grid/Web services, enabling us to share resources with other users rather than commandeering an entire machine. RAVE supports a wide range of machines, from hand-held PDAs to high-end servers with large-scale stereo, tracked displays. The local display device may render all, some or none of the data set remotely, depending on its available resources. This enables scientists and engineers to collaborate from their desks, in the field or in front of specialised immersive displays. We present initial results of our implementation, showing how we distribute complete datasets across multiple machines as required, using a central data service to distribute data updates from collaborating users. We will demonstrate RAVE at SC2004, utilising available heterogeneous resources.

Categories and Subject Descriptors

C.2.4 [Computer-Communication Networks]: Distributed Systems—*Client/server, Distributed applications*; H.5.3 [Information Interfaces and Presentation]: Group and Organization Interfaces—*Collaborative computing, Computer-supported cooperative work*; I.3.2 [Computer Graphics]: Graphics Systems—*Distributed/network graphics*

General Terms

Design, Experimentation, Performance

Keywords

Visualization, Grid/Web services, heterogeneous systems

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SC2004 November 2004, Pittsburgh, Pennsylvania, USA
0-7695-2153-3/04 \$20.00 (c) 2004 IEEE.

1. INTRODUCTION

The increased availability and diversity of tomographic scanning technologies and other data collection systems is creating massive amounts of data. To derive research or diagnostic value from these datasets requires increasingly sophisticated computational infrastructures to store, query, process and visualize the data. Although processing speeds increase at Moore's Law rate (doubling every 18 months) and storage densities and raw graphics processing rates increase even more rapidly, such datasets can quickly overwhelm local capacity. At present to afford the user interactive data visualization and navigation requires the data to be shipped to large graphics supercomputers, which exploit parallelism on a number of levels to achieve the required levels of performance. The user can then view the visualizations using a variety of different display devices.

Increases in network speed and connectivity are allowing more co-operation between geographically remote teams and resources. Such developments allow remote processing (including visualization) and the novel applications of these technologies to support new ways of working.

Grid computing, based on service-oriented architectures such as the Open Grid Services Architecture (OGSA) [9], permit 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). The motivation for this work 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 (Grid infrastructure is discussed further in Section 4.3).

We briefly review current Grid-based and related visualization systems, and introduce RAVE, our Resource-Aware Visualization Environment. We present an overview of its architecture, discuss our implementation and show initial test results of remote visualization using various clients to observe remotely-rendered complex objects. We finish with a discussion of our findings and future work.

2. PREVIOUS WORK

Remote access to rendering hardware enables expensive, specialist hardware to be used without having to move from the user's current place of work to a specific site. This is especially important when the user cannot move to the ma-

chine (for instance, a surgeon performing an operation)—the capabilities of the machine must be delivered remotely to the user. We now review various remote visualization applications, paying particular attention to Grid-enabled applications.

OpenGL VizServer 3.1 [17] 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. One user starts a VizServer session on their machine, and can then invite remote collaborators to share the session. OpenGL applications will now display on all collaborators machines, with one user at a time being in control of the application. Note however, that in current implementations all users share the same visualization—a client cannot have a unique view position.

COVISE [19] is a modular visualization environment (MVE), where one user is the master with control over the entire environment, whilst other users are slaves that can only observe, but may have a unique local view position. COVISE takes advantage of local processing on the slaves by only transmitting changes in data. For instance, the master sends changes to an object, and the slaves then render the new view using a local copy of the data. COVISE also supports running slaves without a GUI—these machines become remote compute servers, enabling the Master to distribute processes over multiple machines, but final rendering must be carried out at the local client machine.

The RealityGrid [12] project is investigating the visualization, modelling and simulation of complex condensed matter structures. A collaborative Problem-Solving Environment (PSE) [18] has been implemented, where a central server coordinates the available resources. Visualization clients connect to the server and request that data output from a remote application be rendered. The rendering is carried out remotely (receiving raw data from the remote application), with the final frame buffer being sent back to the client. A token mechanism controls access to steerable parameters and data.

The e-Demand [8] project is implementing a PSE on the Grid, where each module in the environment is represented by an OGSA service (with each service having a back channel to the previous service in the network). Multiple rendering services or multiple presentation services can be deployed, to form a collaborative environment. e-Demand is specifically targeting autostereo displays, where the rendering service depends on the specific autostereo display, so each display uses the back channel to configure the render service.

IRIS Explorer (an MVE) continues to be augmented and used for collaborative research [24], including the gViz [1] project which aims to Grid-enable the COVISA collaborative visualization tools.

The SuperVise [15] 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. The user supplies a data file, and uses the SuperVise system to select appropriate resources to form a pipeline to visualize the data.

The Distributed Visualization System [13] uses frameless rendering to distribute pixels amongst multiple machines for rendering. Each machine renders a subset of pixels propor-

tional to the machine’s rendering speed, but must contain a complete copy of the dataset.

Visapult [7] is a volume visualization system for massive datasets (of the order of 1-5 Tb). The data are distributed amongst many parallel nodes for volume rendering, using Cactus [3]. The rendered data subset is then sent to the visualization client as a 2D image, which uses local rendering hardware to merge the images into the complete visualization.

Current visualization systems often make assumptions about the available resources; for instance, COVISE assumes local rendering support, whilst OpenGL VizServer assumes the client has modest or very little rendering capability and relies totally on remote resources, other than to unpack and paint pixels onto the display device. We also note a disjointness between collaborative and large scale rendering systems in that collaborative systems do not use rendering distribution such as used in massive dataset visualization (such as Visapult).

We wish to produce a collaborative system that can automatically discover and make use of available resources (either local or remote) through distribution of workload, and react to changes in these resources. To this end we have created the Resource Aware Visualization Environment (RAVE); the architecture underlying RAVE is presented in the next section.

3. RAVE ARCHITECTURE

We propose a novel and unique visualization system that will respond to available heterogeneous resources, provide a collaborative environment, and be persistent (enabling users to collaborate asynchronously with previously recorded sessions). The system must not only react to changes in resources, but also make best use of them by sharing resources between users, and distributing workload amongst resources. We describe our architecture and its underlying components, followed by the important features available through our architecture.

3.1 Architecture Components

We now describe the main components that make up the RAVE architecture; a data service, a render service and a thin client. The architecture is presented in Figure 1, and is followed by a description of each component.

3.1.1 Data Service

The data service imports data from either a static file or a live feed from an external program, either of which may be local or remotely hosted. The data service forms a persistent, central distribution point for the data to be visualized. Multiple sessions may be managed by the same data service, sharing resources between users. The data are intermittently streamed to disk, recording any changes that are made in the form of an audit trail. A recorded session may be played back at a later date; this enables users to append to a recorded session, collaborating asynchronously with previous users who may then later continue to work with the amended session.

The data are stored in the form of a scene tree; nodes of the tree may contain various types of data, such as voxels, point clouds or polygons. This enables us to support different data formats for visualization.



Figure 1: Diagram of basic RAVE architecture

3.1.2 Render Service

Render services connect to the data service, and request a copy of the latest data. The data service informs the render service of any changes, using network bandwidth-saving techniques such as multicasting. This enables multiple render services to simultaneously visualize the same data.

A render service can be exposed to the local console, so a user can interact with the shared data through the render service by modifying their viewpoint or the actual scene itself. This permits large scale immersive devices such as an Immersadesk R2 to be used, along with commodity display hardware. Changes made locally are transmitted back to the data service, propagating to other members of this collaborative session.

If a local user does not have the facility to install a render service on their machine, an active client can be installed instead—this is a stand-alone copy of the render service that can only render to the screen and does not support off-screen rendering (as it does not have a Grid/Web service interface to advertise to other clients). This enables a user to connect to a data service without installing any system software (such as a Grid/Web service container).

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 are stored in the render service to save resources.

Render service may be requested to render a subset of the scene tree or frame buffer; this enables several render services to render a dataset that would otherwise overwhelm

an individual service. The subset is rendered locally and the resulting framebuffer is then transmitted to another render service, where the results are composited. The collaborating render services share the same camera view point, so the framebuffer aligns exactly. Compositing is currently restricted to opaque solids, as this does not require any specific ordering of frame buffers at composition.

3.1.3 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 no or very modest local rendering resources. The thin client must make use of remote rendering, so it connects to the render service and requests rendered copies of the data. The local user can still manipulate the camera view point and the underlying data, but the actual data processing and rendering transformations are carried out remotely whilst the local client only deals with information presentation. This is akin to a render service compositing multiple sections of a complex dataset, only the thin client accepts a single framebuffer for visualization. OpenGL VizServer works in a similar manner, only our implementation is totally heterogeneous.

3.2 Architecture Features

We now describe the important and unique features of the RAVE architecture.

3.2.1 Heterogeneous Support

Once Grid/Web resources (with a known API) are exposed as services, they can be inter-connected without knowledge of the contents of the modules. This enables differ-

ent hardware implementations to communicate, without any consideration of the underlying operating system, processor endianness, etc. For instance, in our testbed we have Linux (RedHat, Fedora and embedded variants), SGI IRIX and Sun Solaris working together. Also, the resources vary from a hand-held PDA to a large SGI Onyx server.

3.2.2 Resource Discovery

To open our system to any resource, we chose to implement Grid/Web services and advertise our resources through Web Service Definition Language (WSDL) documents. WSDL can be registered with a UDDI server, enabling remote users to find our publicly-available resources and connect automatically (no configuration is required by the client, although resources may need to have access permissions modified to permit new users). This is discussed in more detail in Section 4.3.

3.2.3 Efficient Usage of Resources

For our system to use resources as efficiently as possible, we must not interfere with the local user on the console. This enables our services to run on a machine whilst it is being used for another task, unlike (for instance) OpenGL VizServer, which takes over a complete graphics pipe (graphical display).

This concept fits in with our architecture where a service can support many simultaneous clients—now, the host machines can support many simultaneous users, as we are not taking over the machine.

3.2.4 Collaboration

Clients are represented in the dataset by an avatar—a simple graphical object to indicate the position and view of the client. Clients can manipulate items in the dataset, with scene updates being sent to the central data service for reflection to other clients/services.

3.2.5 Workload Distribution

When a dataset would overwhelm the resources on a particular render service, the data may be distributed amongst multiple services instead. When a client requests a dataset to be rendered, it must select which render service to use. The data service interrogates the render service for its capacity (available polygons per second, texture memory, support for hardware assisted volume rendering, etc.). If a render service cannot support the entire dataset, then the data service recruits available render services to assist. Within our present testbed if insufficient resources are available, the request is refused with an explanatory error message.

There are two approaches to workload distribution: dataset distribution and framebuffer distribution. To distribute the dataset, the data server requires sections of the dataset to be marked as being of interest to a render service—this render service must be updated if the data service receives any changes to this subset of the data. The render service itself is thus given a subset of the scene tree, including the parent nodes to orientate the scene subset in the world, along with the client's camera. The render service then renders using the client's camera position and orientation, and sends the resulting frame (and depth) buffer to the client's selected render service.

To distribute the framebuffer, the render service divides its target frame buffer into tiles. A single tile is rendered lo-

cally, whilst the remaining tiles are rendered remotely. The render service requests tiled rendering assistance from the data service, which then forwards the request to the most appropriate render service that is already connected to the scene. The assisting render service renders to an off-screen buffer, which it then forwards it directly to the requesting render service.

3.2.6 Usability

As stated in the previous sections the heterogeneous resource discovery, marshalling and orchestration will be conducted in a manner that is entirely transparent to the end user(s). Our RAVE therefore embraces the spirit of grid computing and provides an effective environment for end users to easily discover and perform high end visualization services and transforms. Furthermore the visualizations can be delivered wherever convenient to the user for display and interaction, both individually and within collaborative and geographically remote sessions. Through this combination of features our RAVE system proves highly user friendly, intuitive and usable for a variety of end-users.

3.2.7 Workload Migration

When a render service becomes overloaded (i.e. its rendering rate drops below a given threshold), it informs the data server. The data server then examines available render services to find which service has spare capacity. The data service then uses the workload distribution scheme to distribute the scene across render services, removing nodes or tiles from the overloaded service and adding them to an alternate service. If there is insufficient spare capacity, then the data server uses UDDI to discover additional render services that are not connected to the data service. These underutilised services can then be recruited to join the session hosted on the data service and contribute to the rendering resources.

When a render service is significantly underloaded (for a given amount of time, to smooth out spikes of usage), the data service again redistributes data. This time nodes are added to the underloaded service, and removed from the most loaded render service. If no more nodes can be added, the service is marked as available to support other overloaded services.

Nodes must carefully selected to perform a fine-grain movement of work. If an underloaded service has capacity for another 5k polygons/sec and still maintain its current interactive frame rate, we do not want to add 100k polygons by mistake—this service will then become overloaded and need its work redistributing. To avoid this, we will use metrics to define how much capacity is remaining on a service (texture memory, polygons/voxels/points per second), and how much data are contained in a given set of nodes (in terms of texture memory and number of polygons/voxels/points). We can then select an appropriate set of nodes or tiles to move in order to load balance the system. Loadings due to user interaction and navigation will have to be analysed to determine these usage profiles and the workload migration trigger thresholds.

4. IMPLEMENTATION

We now discuss our implementation and reasons behind our selection of various tools.

4.1 Language Used

Java was selected as this runs cross-platform without modification, and the majority of the Grid tools are implemented in Java. C/C++ APIs are becoming available (such as the Apache Axis-C project [5]), so we may move to C/C++ for performance gains later. As we are using Grid/Web services, we can change the underlying implementation of an advertised service without affecting the rest of RAVE.

4.2 Java3D Usage

We require a graphic API that is cross-platform, and will support off-screen rendering. We investigated various other APIs (including VTK [22], Java OpenInventor [20], JOGL [11]), but Java3D was the only solution that was truly cross-platform, supported off-screen rendering (in hardware), and was not restricted by a license fee for each instance.

Code is designed around interfaces, so the underlying rendering API is hidden. Once JOGL is stable, we will implement a rendering engine for this API as a comparison. Currently, JOGL has incomplete support for off-screen rendering.

As we use Grid/Web services, we can change our underlying implementation. Apache Axis-C is now released, so an option for the future is to implement a render service in C++ using OpenGL or higher-level library (such as VTK).

4.3 Grid Infrastructure

The current OGSA model [9] of Grid services is changing rapidly, converging with the Web Services Resource Framework (WSRF) [2]. This means that service implementations must not become tied to any particular infrastructure, as this is subject to change. We wrap our serving technology inside OGSA, Web Services and a (multi-threaded) test environment, so our service “engine” is unaffected by changes in infrastructure, only the wrapper needs to be modified.

Grid and Web services both implement remote procedure calls by sending the procedure arguments and results in XML format (using the Simple Object Access Protocol (SOAP) [23]). They are hence not tied to any particular architecture (such as little or big endian format), as they are transmitted as plain text. This also means that they are not suited to large data transmission or low latency, due to the size of the SOAP packets related to the size of the data, and the time required to marshal/demarshal the data. Note that Grid services in OGSA are implemented as factories, where a service can create instances; Web services are usually stateless function calls. If static variables are stored within Web services, then they can behave in a similar manner as Grid services by implementing a factory as a design pattern (such as passing the name of an instance as the first argument to all instance related methods). The differences between Grid and Web services will be removed once they converge with the WSRF.

Grid and Web services can be both be advertised through standard directory services, such as LDAP [26] or UDDI [14] (often used to register personal contact details, such as telephone numbers, or web pages). We selected UDDI as it has good support with Java, and UDDI version 3 has additional support for Web services and WSDL documents. A Grid/Web service can have its API described in a WSDL document, which is then advertised as a “Technical Model” in UDDI. If any services are advertised as adhering to this technical model, then we know they will have the same API

and underlying behaviour. Hence we have two technical models, one for the data service and one for the render service. For testing, we use jUDDI [6] as a local UDDI server and the IBM test registry [10] for additional testing. Once our services are stable, we publish them to our production UDDI server (hosted by the Welsh e-Science Centre [25]).

With the issues of SOAP taken into account, we only use Grid/Web services for initial service discovery (via UDDI), status interrogation and subsequent subscription. We then back off from SOAP and use direct socket communication to send binary information. The sockets are specified during the initial SOAP-based service subscription by the client, when the service instance is selected.

At present, we are using Apache Axis [4], with Apache Tomcat as a container. The build process is simpler and faster than Globus Toolkit 3 (GT3) [16], so is more suited to our needs during this research and development phase. GT3 is too complex to compile for a PDA (various issues with system headers), so at present we must use Apache Axis with our PDA. We may switch back to using GT3 when we wish to use Grid security certificates to authorise users. However, the UDDI service can store multiple technical models, so we can support both variations simultaneously, as long as all clients and services contain support for both Grid and Web services.

4.4 Testbed Implementation

Our resources are an SGI Onyx 3000 with 32 CPUs and three Infinite Reality graphics pipelines, a Sun Microsystems Inc. V880z with XVR4000, an Intel Centrino 1.6GHz laptop with nVidia GeForce2 420 Go graphics, a dual 2.4GHz Xeon desktop with nVidia FX3000G graphics, an AMD Athlon 1.2GHz desktop with nVidia GeForce2 GTS, and a Sharp Zaurus PDA.

The actual demands on the PDA are for a network connection, enough memory to import and process a frame buffer (120kB for a 200x200 image) and a simple GUI. We are device-agnostic, so any PDA could be selected. The Zaurus was selected as it runs Linux, so all our development machines are running a variant of UNIX (to simplify maintenance).

5. INITIAL TEST RESULTS

Two test models (a skeletal hand and a skeleton) were obtained from the Large Geometric Models Archive at Georgia Institute of Technology [21]. The models were in PLY format, converted to Wavefront OBJ and then imported into our data service. The skeletal hand is from the Stereolithography Archive at Clemson University. The skeleton is taken from the Visible Man project (from the National Library of Medicine), where the skeleton was processed by marching cubes and a polygon decimation algorithm. The two models are shown in the screenshots from the PDA in Figure 2.

Table 1: Models used in benchmarks

Model Name	Number of Polygons	Size of Data File
Skeletal Hand	0.83 million	20MB
Skeleton	2.8 million	75MB

5.1 PDA

The Zaurus supports Java Micro Edition (J2ME), Personal Profile / Connected Device Configuration. This is a subset of Java 2 Standard Edition (J2SE, as used for our data and render services). J2ME has a cut-down image support library, so we could not use supplied J2SE routines to send the image. Sending an image “manually” (by sending each pixel one at a time, converting to a series of bytes for the network) took over two minutes to send a single frame. Hence, we could not use J2ME for high-speed data transmission.

As we use WSDL to advertise our data and render services, we are not restricted to a particular platform or language. This enabled us to use a C/C++ client with the PDA, where an array of raw data bytes is downloaded and the data pointer is directly cast to the appropriate image format, involving minimal overhead; note that this is not possible in Java. This can be seen from our results, where it takes approximately 0.2s to receive and blit an image—compared to over two minutes with Java.

Note that the thin client running in J2SE can use the fast image conversion routines (from byte array to image), so we only need C/C++ with the PDA, all other platforms use J2SE.

Our timings for visualising the test models are presented in Table 2, the render service used was an Intel Centrino 1.6GHz machine equipped with a GeForce2 420 Go graphics device, sending an uncompressed 200x200 pixel image over an 11Mbit/sec wireless network.

Note that the image transfer time is significant; for a 200x200 24 bits-per-pixel image, we receive a maximum of 5 frames per second; this equates to a bandwidth of around 580Kb/sec. For a 640x480 24 bits-per-pixel image (920Kb in size), this would result in around 0.6 frames per second. Hence we need to investigate image compression, as our bottleneck is the available network bandwidth. Wireless network bandwidth is shared between other network users, and is proportional to signal quality (bandwidth is reduced when the user moves away from an access point, or when walls, etc. attenuate the signal transmission). We need a compression algorithm that can adapt on the fly to changing network conditions.

The render service pixel rendering times are highly dependant on the number of polygons on-screen; the views were arranged to have the maximum possible number of visible polygons to present the worst case render service load, and are presented in Figure 2. Note that the Zaurus has a display resolution of 640x480 pixels, so the 200x200 pixel images are small relative to the display.

5.2 Collaboration

We present two users visualising the same dataset in Figure 3, where the local user can see the remote user (who’s host machine is called “Desktop”) navigating around the dataset. Each user is represented by a simple avatar—in this case, a cone pointing in the direction of the user’s view, and the name of the user or host. Our current GUI enables users to carry out actions with specific objects (such as the user’s camera), with selected objects or relative to selected objects (such as rotate the camera around a selected object).

The GUI interrogates objects for any supported interactions, and reflects this in the drop-down menus; all interactions are based on clicking to select/deselect an object,

and dragging. This simple interface then maps neatly onto a PDA. The interrogation approach was selected as this permits alterations of the supported interactions without affecting any part of the GUI or underlying message transport. We will later create additional interactions for special objects, such as bridging objects into remote processes. An example would be to exert a force on a molecule, which is displayed via RAVE but the molecule’s behaviour is computed remotely via a third-party simulator; RAVE is used as the display and collaboration mechanism.

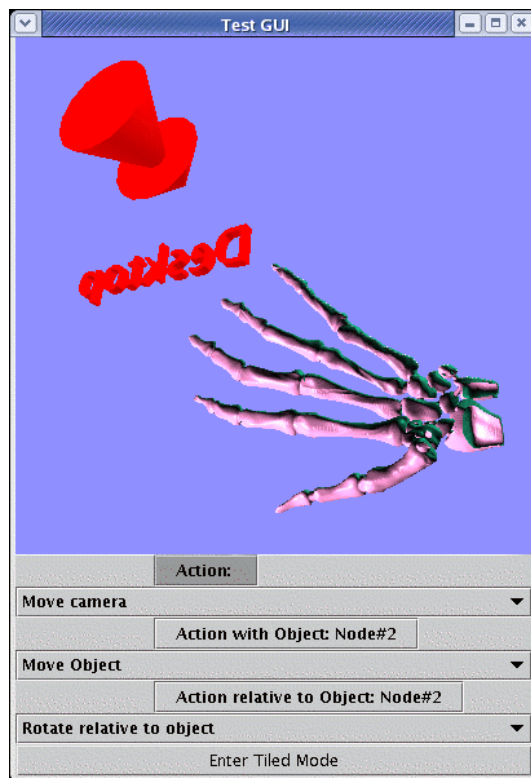


Figure 3: Two users visualising the same scene collaboratively—skeleton hand scene

5.3 Heterogeneous Resources

We have used various client displays, from mobile devices (such as the PDA and Laptop) to large-format immersive displays (such as a FakeSpace Portico rear-projection active stereo Workwall). We use a simple client GUI to examine a UDDI registry, which then reports on what instances are available at each resource. This is shown in Figure 4.

In this example, two machines have registered with the UDDI server, with several service instances on these machine. Each data service instance represents a data session that is available for users connect a render service or render capable client. Each render service instance is a render service that is already connected to a data service and is ready for a thin client to connect to receive rendered images. Note that in our example, machine “tower” has a render service running “Skull-internal”—this dataset was obtained from machine “adrenochrome”, from data service “Skull”. The render services on adrenochrome were using local instances of the data server.

Table 2: Visualization Timings Using a PDA

Model Name	Number of Polygons	Frames per Second	Total Latency	Image Receipt	Render Time	Other Overheads
Skeletal Hand	0.83 million	2.9	0.339s	0.201s	0.091s	0.047s
Skeleton	2.8 million	1.6	0.598s	0.194s	0.355s	0.049s

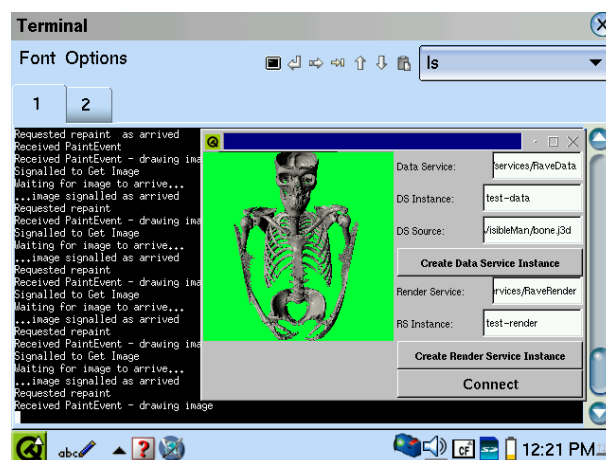
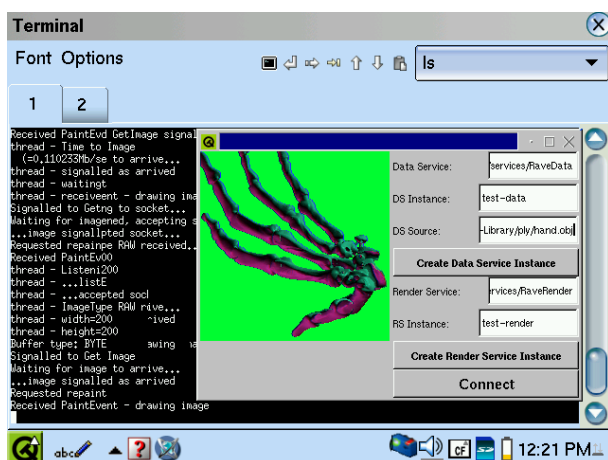


Figure 2: Screen dumps from a Zaurus PDA running the RAVE thin client

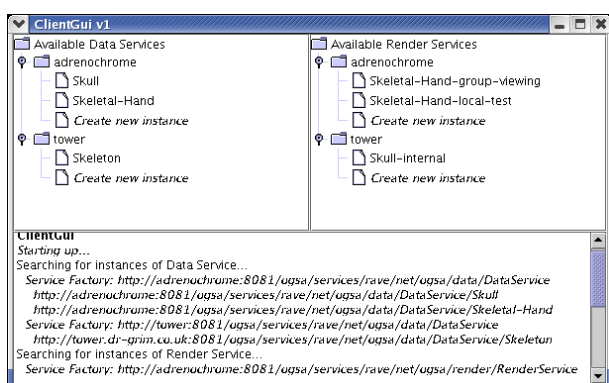


Figure 4: Simple UDDI registry GUI

The GUI also has the option of creating new instances, by clicking on the “Create new instance” service instance, in italics at the bottom of each service instance listing. This permits the entry of a data URL to create a data service, or the URL of the data service instance to create a new render service (as a render service needs a data service to bootstrap from).

5.4 Off-Screen Rendering

Java3D renders off-screen scenes significantly slower than on-screen; taking 100% to be on-screen rendering speed, Table 3 presents our findings with several machines and datasets for off-screen rendering (rendering a 400x400 pixel image). Dataset “Elle” is a Blaxxun VRML Benchmark, whilst “Galleon” is a Sun Microsystems Inc. example file supplied with Java3D; both datasets are untextured polygons.

We noted that the CPU usage during these tests was much lower for off-screen rendering compared to on-screen rendering (such as around 20% usage). This implies that Java3D is rendering off-screen slower than perhaps it could, as to render off-screen initiates a request for an image to be rendered, and then test if it has completed there is no direct control over the rendering. To investigate this, we rendered off-screen four 200x200 images simultaneously. In one test, we sequentially requested an image to be rendered, waited for it to complete and then moved to the next image. In the second test, we interleaved our requests so that we requested all 4 images to render, and waited in a round-robin fashion for each to complete, when it was requested to render again. This should overlap the rendering as much as possible, with our results presented in Table 4 where 100% is on-screen rendering of four 200x200 images. We use “int” to imply interleaved, and “seq” to imply sequential rendering.

These results show that with a Linux workstation, the on-screen rendering speed is available if multiple images are rendered. Our results for the V880z with the Elle model are surprising, and possibly indicate off-screen rendering is carried out in software rather than hardware. Further investigation of this is necessary, as all tests were carried out using identical Java code, and versions of Java and Java3D (1.4.2 and 1.3.1 respectively).

5.5 Workload Distribution

If insufficient resources are currently subscribed to the session (e.g. more render services required), then the data service will attempt to recruit additional resources via UDDI. Our initial timings are presented in Table 5. The UDDI timings are on a clear network (100Mb ethernet); the first time given is for an already initialised UDDI session to scan for access points of RAVE render services. The second time covers a full UDDI bootstrap (including proxy creation, scan busi-

Table 3: Off-screen render timings

400x400 image Dataset	GeForce2 420 Go Centrino 1.6GHz	GeForce2 GTS Athlon 1.2GHz	XVR-4000 Sun Fire V880z UltraSPARC III 900MHz
“Elle” (50kpoly)	35%	40%	3%
“Galleon” (5.5kpoly)	9%	9%	16%

Table 4: Off-screen render timings

200x200 image Dataset	GeForce2 420 Go Centrino 1.6GHz	GeForce2 GTS Athlon 1.2GHz	XVR-4000 Sun Fire V880z UltraSPARC III 900MHz
“Elle” (50kpoly)	seq:55% int:90%	seq:51% int:90%	seq:3% int:4%
“Galleon” (5.5kpoly)	seq:9% int:33%	seq:11% int:41%	seq:30% int:48%

ness representing the RAVE project, scan for render services under the RAVE project, and finally scan for access points of these services). Once this initial “bootstrap” is carried out, the UDDI proxy can be kept live and the instead use the simpler check of scanning the access points (to check for service removal or insertion).

The service bootstrap was carried out locally on 100Mb ethernet, including the time spent to contact the Axis Web Service, request the creation of a new render service instance and bootstrap from the data service.

Table 5: Timings of UDDI recruitment and subsequent service bootstrap

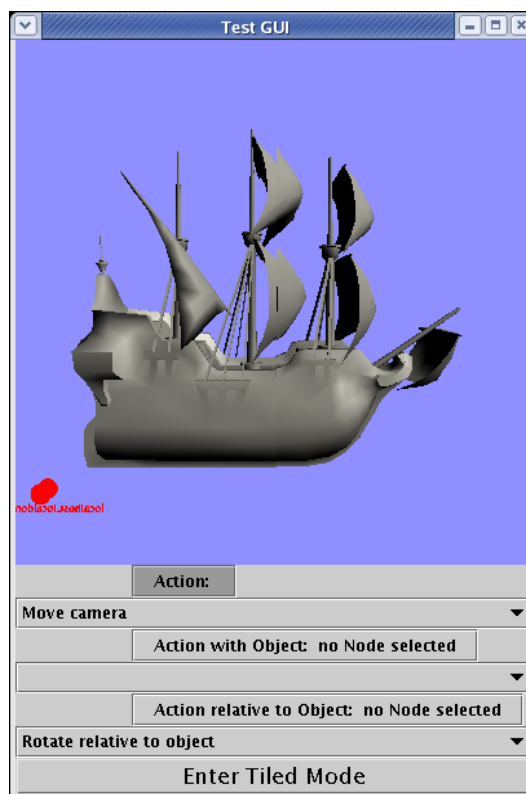
	Size of	UDDI	Service
Model	Data File	scan	bootstrap
Galleon	0.3Mb	0.73s (4.8s full)	10.5s
Skeletal Hand	20Mb	0.70s (4.2s full)	68.2s

With our tiled rendering approach, the bootstrap is not a major issue as we continue to render locally until we receive a tile from our remote render service. We overlap update messages with the initial bootstrap messages, so the remote resource does not miss any updates during initialisation. Hence if the user modifies the scene or moves the camera, as soon as the remote resource is bootstrapped it will be pre-synchronised with data service and reflect the scene or camera changes.

Once the remotely rendered tile is received, we resize our local rendering and present the remotely rendered data to the user. Our bootstrap is slower than (for instance) directly sending a native Java3D stream, and is presently bottlenecking on Java’s marshalling/demarshalling of data types from the network. We are using introspection, where each node in the scene graph is examined for implemented interfaces, and the appropriate interface is used to extract the data and publish it on the network. With this method, we reduce our maintenance overhead with code sharing (e.g. many items have a “Position” field, so this is an interface we check for). However, it is likely that this is slowing up the transfer of data to and from the network. We will investigate this further to speed up our network communication.

At present we are not using any synchronisation between frame buffers, local and remote simply rendering “best effort” and continuously stream images to the user. In practise, this can result in visual artifacts such as tearing (for example, our test GUI in Figure 5 shows a tear in the region of the middle mast of the galleon, where the seam is

between the tiles). With 100Mb ethernet the delay between local changes to the scene graph and receiving an updated tile is very small, as long as the render time is insignificant. With the galleon model, the delay was approximately 0.05s and hence quite acceptable. It is too small to be recorded using a human-operated screen capture, so the image was produced by artificially stalling the remote render service. However, with a complex model (such as the skeleton), the latency increases when the render time becomes significantly longer than the transport time. The skeletal hand produces roughly 0.3s latency from mouse drag to updating the tile, so we will need to implement synchronisation with complex scenes.

**Figure 5: Tearing artifact from 2 tiles**

6. FUTURE WORK

We will investigate larger datasets (in the order of several GB) to test our workload distribution algorithm, which we aim to demonstrate at SC2004 working across a heterogeneous selection of machines.

At present, we are only using polygonal datasets. We will extend our support and rendering services to include voxel and point based methods; these will distribute across multiple render services. Subset blocks of the volume can be blended, even though they contain transparency, by considering their relative distance from the view in the order of blending (such as Visapult).

We will finish our implementation of our workload redistribution strategy, which extends our work on workload distribution. This will enable us to use any available resource, as we can stop using a machine once it becomes loaded by (for instance) a local user logging on and using the machine interactively.

Image compression methods are presently being investigated; these are required for the render work distribution and for transmission to thin clients. Special attention is required for the thin client, as it may use a wireless network whose bandwidth is both low and highly variable.

Finally, we will consider the distribution of the data across several data servers, to match our render service workload distribution. This will alleviate any bottleneck in our system, and also support a fail-safe mechanism, where data servers could mirror each other.

7. CONCLUSIONS

We have presented a novel and highly capable visualization framework that supports heterogeneous resources, in terms of system architecture and system resources. This has enabled us to support a wide and diverse range of devices ranging from a PDA (which has minimal local resources) to an SGI Onyx server, along with various workstations and desktops that fit between these two extremes.

Heterogeneity has been used to enable a C++ based client to communicate to a Java-based server alongside Java-based clients without any consideration being made of the source/target implementation language.

Our framework supports collaboration, enabling a handheld device to interact with a user in an immersive environment (such as a tracked, stereo Immersadesk). This enables a user to have a private view of the dataset whilst viewing a large-scale projection (such as a Portico Workwall) which is presented to a group of users.

We have found issues with off-screen rendering with Java3D, but as we are not tied to any particular implementation, we will migrate to an alternative solution once one becomes available. This will not affect any part of the system, as implementation details are hidden behind Grid/Web services.

We consider this system to be of widespread interest due to its heterogeneous nature, enabling scientists to visualize large datasets using whatever computing resources they have to hand. Scientists are also able to collaborate with colleagues without restriction on their host platform or local resources.

8. REFERENCES

- [1] gViz - Visualization Middleware for the Grid. <http://www.visualization.leeds.ac.uk/gViz>.
- [2] Grid and Web Services Standards to Converge. Press release at http://www.marketwire.com/mw/release_html_b1?release_id=61977, GLOBUSWORLD, January 2004.
- [3] G. Allen, W. Benger, T. Goodale, H.-C. Hege, G. Lanfermann, A. Merzky, T. Radke, E. Seidel, and J. Shalf. The Cactus Code: A Problem Solving Environment for the Grid. In *Proceedings of the Ninth International Symposium on High Performance Distributed Computing (HPDC'00)*, pages 253–262. IEEE, August 2000.
- [4] Apache. Apache Web Services Project—Axis. <http://ws.apache.org/axis/index.html>, 2004.
- [5] Apache. Apache Web Services Project—Axis C++. <http://ws.apache.org/axis/cpp/index.html>, 2004.
- [6] Apache. Apache Web Services Project—jUDDI. <http://ws.apache.org/juddi/>, 2004.
- [7] E. Bethel and J. Shalf. Grid-distributed visualizations using connectionless protocols. *IEEE Computer Graphics & Applications*, pages 51–59, March/April 2003.
- [8] S. M. Charters, N. S. Holliman, and M. Munro. Visualisation in e-Demand: A Grid Service Architecture for Stereoscopic Visualisation. In *Proceedings of UK e-Science All Hands Meeting 2003*, September 2003.
- [9] I. Foster, C. Kesselman, K. M. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Technical report, Globus, February 2002.
- [10] IBM. IBM Test UDDI Registry. <https://uddi.ibm.com/testregistry/registry.html>, 2004.
- [11] JOGL. Reference Implementation of the Java Bindings for OpenGL. <https://jogl.dev.java.net/>, 2004.
- [12] LeSC. RealityGrid. Project flyer, London e-Science Centre, September 2002.
- [13] J. Mahovsky and L. Benedicenti. An Architecture for Java-Based Real-Time Distributed Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 9(4):570 – 579, October–December 2003.
- [14] OASIS. Universal Description, Discovery and Integration (UDDI)—Version 2 Specifications. <http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm>, July 2002.
- [15] J. Osborne and H. 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.
- [16] T. Sandholm and J. Gawor. Globus Toolkit 3 Core—A Grid Service Container Framework. Globus Toolkit 3 Core White Paper, July 2003.
- [17] SGI. SGI OpenGL VizServer 3.1. Data sheet, SGI, March 2003.
- [18] 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.
- [19] H. P. C. C. Stuttgart. COVISE Features. <http://www.hlrs.de/organization/vis/covise/features/>,

HLRS, September 2000.

- [20] TGS. TGS Java Open Inventor. http://www.tgs.com/index.htm?pro_div/3dms_j_main.htm~main, 2004.
- [21] G. Turk and B. Mullins. The Large Geometric Models Archive. http://www.cc.gatech.edu/projects/large_models/, Georgia Institute of Technology, 2003.
- [22] VTK. VTK—The Visualization Toolkit. <http://www.vtk.org/>, 2004.
- [23] W3C. Simple Object Access Protocol (SOAP)—W3C Recommendation Version 1.2—Primer. <http://www.w3.org/TR/soap12-part0/>, June 2003.
- [24] J. Walton. *Visualization Handbook*, chapter NAG's IRIS Explorer. Academic Press, 2003.
- [25] WeSC. Welsh e-Science Centre. <http://www.wesc.ac.uk>, 2004.
- [26] W. Yeong, T. Howes, and S. Kille. RFC 1777—Lightweight Directory Access Protocol (LDAP). <http://www.faqs.org/rfcs/rfc1777.html>, March 1995.