

Visualization Across the Pond: How a Wireless PDA can Collaborate with Million-Polygon Datasets via 9,000km of Cable

Ian J. Grimstead*, Nick J. Avis and David W. Walker
Cardiff School of Computer Science
Cardiff University

Abstract

We present an initial report on using our distributed, collaborative grid enabled visualization environment to link SuperComputing 2004 (Pittsburgh, PA, USA) with the Cardiff School of Computer Science (Cardiff, Wales, UK). A PDA was used to visualize large, shared datasets (in the range of 0.5-4.5 million polygons) stored in Cardiff, interacting with a laptop (rendering the data locally). The system used was the Resource-Aware Visualization Environment (RAVE), deployed as Web Services running in the background on remote machines. This enables us to use a wide range of heterogeneous machines without being concerned with the underlying implementation of RAVE, or the architecture of the machine.

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

Keywords: Visualization, Grid/Web services, heterogeneous systems

1 Introduction

Increases in network speed and connectivity are promoting the use of remote resources via grid computing, based on service-oriented architectures such as the Open Grid Services Architecture (OGSA) [Foster et al. 2002]. These 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).

With datasets rapidly increasing in size, the visualization of such datasets can quickly overwhelm local computing power. The availability of Grid computing enables users to recruit resources to supply datasets or to assist in their rendering. Grid computing (and high speed networking in general) enables co-operation between remote teams at interactive rates and hence becomes more desirable as the network technology improves.

With this in mind, we present an experiment carried out using the Resource-Aware Visualization Environment (RAVE), us-

ing Grid/Web Services to advertise data sources and recruit rendering assistance from remote resources. We describe an experiment carried out at SuperComputing 2004, where RAVE was presented [Grimstead et al. 2004]; a PDA interacted collaboratively with a laptop at SC2004 (Pittsburgh, PA, USA) via services located remotely in the Cardiff School of Computer Science (Cardiff, Wales, UK).

We briefly review current Grid-based and related visualization systems and introduce RAVE. We present an overview of its architecture and discuss our implementation in detail (including such factors as choice of language and rendering technology). We then report on our experiment of using RAVE across the Atlantic, followed with a discussion on future work and our final conclusions.

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. Although basic desktop machines have graphics capabilities that we could only dream of a few years ago, demand for rendering power will often outstrip the available resources. Hence we wish to make specialised, high-end resources available for sharing with remote users. This is especially important when the user cannot move to the machine (for instance, a surgeon performing an operation)—the capabilities of the machine must be delivered remotely to the user.

OpenGL VizServer 3.1 [SGI 2003b] 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. All users share the same visualization—a client cannot have a unique view position. However, this is still a very useable approach, and has been used to good effect with medical applications (such as [John 2003]).

An alternative approach is taken by COVISE [Wössner et al. 2002], which is a Modular Visualization Environment (MVE). 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 (unlike VizServer, where all participants share a single view). COVISE takes advantage of local processing on the slaves by only transmitting changes in data, where each machine renders the data locally (requiring graphics hardware on each machine).

The ARTE environment [Martin 2002] presents a hybrid approach whereby a full bitmap may be transmitted (akin to VizServer), the geometry may be transmitted (akin to COVISE) or a combination of the two. In combination mode, partial geometry is sent with a depth-buffered bitmap rendering of the remainder, for the client to combine with the local rendering. The hybrid rendering approach fits well with the requirements of RAVE, but ARTE appears to use a single render server (supported on a single operating system)

*I.J.Grimstead@cs.cardiff.ac.uk

whereas we are interested in a cross-platform collaborative environment that can use multiple resources.

gIX [Womack and Leech 1998] and PEX [Rost et al. 1989] are both methods for rendering data stored remotely over X11. The program runs on a remote machine, while the local client receives a stream of prepared primitives for localised rendering. This approach is tied to the underlying windowing system (namely, X11) and is orientated towards remote rendering of a single application. The continuous streaming of prepared primitives is an approach somewhere between sending the whole scene graph (unprocessed) and sending a rendered frame buffer. However, with this approach the data is sent repeatedly, yet the local client still has to perform the rendering. This results in the continuous usage of both network bandwidth and GPU resources (the worst of both approaches), and could explain why current research is biased towards streaming of frame buffers or initial bootstrap of the scene graph.

MVEs and Problem Solving Environments (PSEs) are popular tools with visualization, with several projects using this approach. The e-Demand project [Charters et al. 2004] is implementing a PSE on the Grid, where each module in the environment is represented by an OGSA service, whilst the gViz project [Brodie et al. 2004b] is extending IRIS Explorer [Walton 2003] to be grid-enabled and collaborative (where users can independently control each module of the MVE).

For a fuller review of remote visualization applications refer to [Grimstead et al. 2004] and [Brodie et al. 2004a].

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 rendering resources. 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), which was initially presented at SC2004 [Grimstead et al. 2004]; 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 now describe the main components that make up the RAVE architecture; a data service, an active client, a render service and a thin client, with the architecture presented in Figure 1.

3.1 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.

3.2 Active Client

An active client is a machine that has a graphics processor and is capable of rendering the dataset. The active client connects to the data service and requests a copy of the latest data, or a subset of the latest data, bootstrapping the client. The client then maintains a connection to the data service, receiving any changes made to the data by remote users and also sending any local changes made by the local user. Note that only changes are sent—this reduces the bandwidth required, akin to COVISE [Wössner et al. 2002].

The selection of a subset of the data can enable a client with insufficient rendering power to render a lower resolution Level Of Detail (LOD), or to render just the area of interest. LOD approaches are used in various rendering architectures, such as ARTE [Martin 2002] and MUVEES [Chen et al. 2003].

3.3 Render Service

Render services operate in a similar manner as an active client, except these do not render to the local console, instead operating in the background. If a local client does not have sufficient resources to render the data, a render service can perform the rendering instead and send the rendered frame over the network to the client for display. 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 single copies of the data are stored in the render service to save resources.

The host console is not visually affected when running a render service (the only side effect being a slow-down in on-screen rendering as the GPU is being shared). A specialised graphics card can then be used simultaneously by multiple users (local and remote), promoting the sharing of graphical resources. This is in contrast to other systems, such as VizServer [SGI 2003b], which may take over the host console to perform rendering.

A 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 or active client, 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.

3.4 Thin Client

Whereas an active client 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

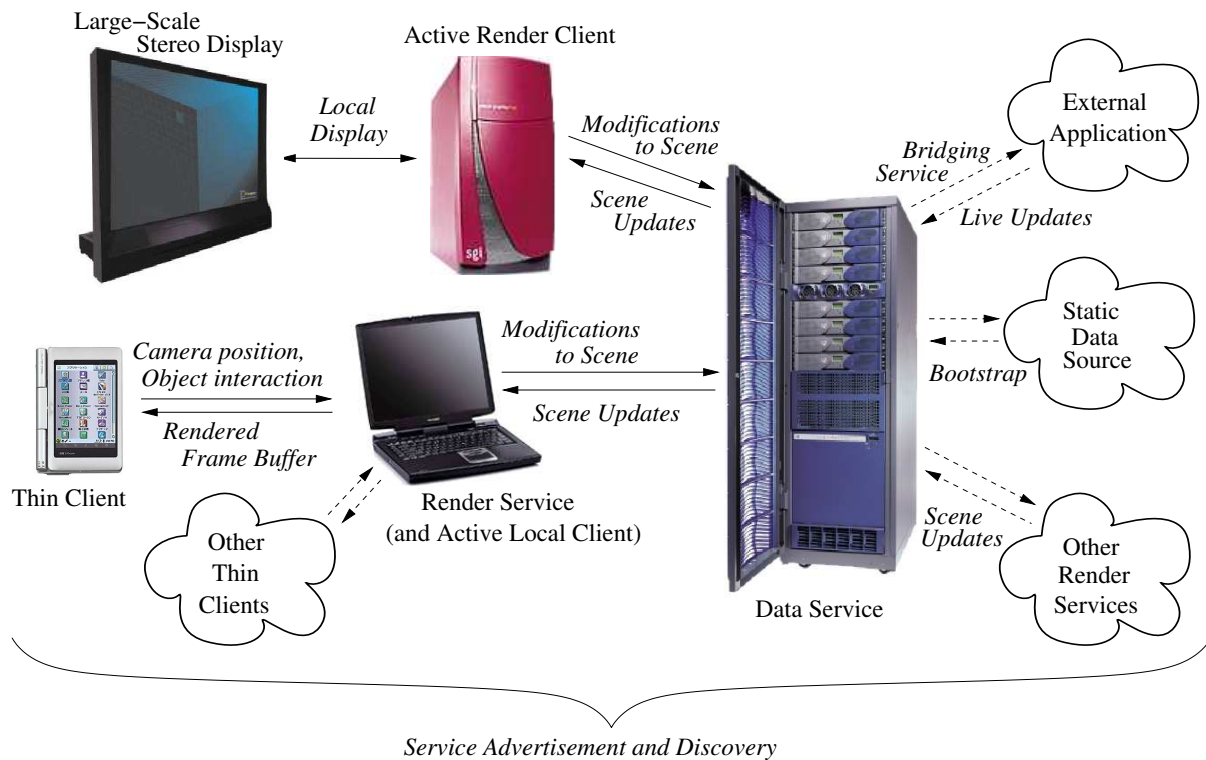


Figure 1: Diagram of basic RAVE architecture

actual processing and rendering transformations are carried out remotely whilst the local client only deals with information presentation. OpenGL VizServer [SGI 2003b] works in a similar manner, only our implementation is totally heterogeneous.

4 Implementation

We now discuss our implementation and reasons behind our selection of various tools. Further information regarding the PDA implementation and workload distribution may be found in [Grimstead et al. 2004].

4.1 Grid/Web Services

The OGSA model [Foster et al. 2002] of Grid services is converging with the Web Services Resource Framework (WSRF) [IBM 2004a]. 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) [W3C 2003]). The specification of the procedure calls is defined using Web Services Definition Language (WSDL) [W3C 2001], again based on XML. They are hence not tied to any particular architecture (such as little or big endian format), as they are transmitted as plain text. Services can then be inter-connected or connected to a client without knowledge of the underlying implementation, machine architecture

or operating system. For example, when working with RAVE we use (and inter-operate with) Linux (RedHat, Fedora and embedded variants), SGI IRIX, Sun Solaris and Microsoft Windows.

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.

The usage of XML means that SOAP messages are not suited to low latency or large data transmissions, due to the size of the SOAP packets compared to the size of the original data, and the time required to marshal/demarshal the data. With the issues of SOAP taken into account, we only use Grid/Web services for initial service discovery (via UDDI—refer to the next section), 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.

Globus Toolkit 3 (GT3) [Sandholm and Gawor 2003] hosts Grid Services and has inbuilt security, job submission, certificate handling, etc. The basic installation of Apache Axis [Apache 2004a] as a Web Services host (with Apache Tomcat as a container) is just an insecure WS implementation. However, the lack of features in Axis is an advantage at this stage of development, as we do not need the additional features and hence do not wish to pay the penalty of maintaining such a heavyweight system. Additionally, we have had problems compiling GT3 to work with a PDA (such as system header conflicts, requirements of 100’s of Mbs of storage whilst

building the libraries); hence we selected Apache Axis.

4.2 Service Discovery

Grid and Web services can both be advertised through standard directory services, such as LDAP [Yeong et al. 1995] or UDDI [OASIS 2002] (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 [Apache 2004c] as a local UDDI server and the IBM test registry [IBM 2004b] for additional testing. Once our services are stable, we publish them to our production UDDI server (hosted by the Welsh e-Science Centre [WeSC 2004]).

To produce WSDL documents, we used `java2wsdl` and `wsdl2java` from the Apache Axis project. These first generate a WSDL document from a compiled Java interface, and then generate the Java service skeletons (to contain the Web Service), and the related client stubs (to call the remote service). For the C/C++ client, we used gSOAP [van Engelen 2004]; this produced the C++ client stubs for the PDA.

4.3 Language Used

Java was selected as this runs cross-platform without modification, and the majority of the Grid tools are implemented in Java. As we wished to reuse as much 3rd party code as possible with the minimum of effort, Java was a natural choice. There is also a cut-down version of Java available for PDAs (J2ME), so this would enable some code to be used even on embedded devices.

C/C++ APIs are becoming available (such as the Apache Axis-C project [Apache 2004b]), 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. However, a C/C++ implementation would require a cross-platform build system and more thorough testing.

For our PDA client, our real-time display (as a thin client) required rapid access to the frame buffer, this was not possible from J2ME as the image support libraries are only present in J2SE. Hence we had to write a C++/QTopia client for the PDA, which also runs under Linux. Web Services hide the underlying implementation of the service, and hence also of the client requesting the service. This has enabled the C++ client to interact with the services without any special knowledge.

All other services and clients are written in Java, enabling us to run RAVE under many environments, including as an Applet in a Web Browser.

4.4 Java3D Usage

We require a graphic API that is cross-platform, and will support off-screen rendering. We investigated various other APIs (including VTK [VTK 2004], Java OpenInventor [TGS 2004], JOGL [JOGL

2004]), 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. Java3D uses OpenGL as its underlying rendering mechanism, and is hence fairly efficient if Java3D is requested to use display lists (as these then run directly on the graphics hardware, and are not affected by our choice of implementation language or environment).

Java3D has been released by Sun as an open-source project [Sun Microsystems Inc. 2004], whilst SGI and Sun announced a joint initiative to produce Java bindings for OpenGL [SGI 2003a]. This could indicate that Sun is moving its focus away from Java3D and towards OpenGL, but the community around Java3D is active, with a new version of Java3D (1.3.2) due for release. Hence we feel that Java3D is a good solution for the present, but we will also create a Java OpenGL (JOGL) implementation for comparison and future-proofing. At the time of writing, JOGL does not have a binding for SGI IRIX, so does not yet support all of our platforms, and the off-screen rendering is still under development.

4.5 Importing External Data

Java3D has a lot of community support for it, including plug-in modules for importing geometry from 3rd party applications. As we wish to interact with as many datasets as possible, this is a major plus. There are several VRML importers, and the X3D format is supported with the Xj3D project [Xj3D 2004]. We have used Xj3D to import various VRML files, including output from MolScript [Kraulis 1999] (where Protein DataBank files are converted to VRML, and then imported into RAVE using the Xj3D libraries; for an example, see Colour Plate 2).

Note that we can continue to use the Java3D as an importer if we stop using it as a renderer, as we can extract the data from the Java3D structures and copy into an alternative implementation. In RAVE, we have black-boxed our underlying renderer so we can simply import into a Java3D based RAVE data service, and then copy the data into a JOGL based RAVE render service/active client, hence making use of pre-existing importers.

To import other data types, we have also written custom importers. This has enabled visualization of Diffusion Tensor Imaging datasets [Connell and Bastin 2004] (see Colour Plate 1) and the ETOPO datasets [National Geophysical Data Center 2004] (see Colour Plate 3).

4.6 Inter-Service Communication

We required a protocol that would bootstrap a machine (active client or render service) and then send incremental changes to reduce network usage. As we are using Web Services (WS), any remote method calls use SOAP—which is based on XML. XML is text-based and wraps numerical data in such a manner as to increase its size by an order of magnitude; for instance, a four byte floating point value could be stored as:

```
<float>5.213132</float>
```

This simple example requires 24 bytes, compared to the four binary bytes of the original floating point value. For short, occasional data transmission (e.g. less than ten values), the overhead of sending as XML is acceptable. For streaming a large dataset (such as our examples, containing a million or more polygons), the time to encode and decode to/from XML would be too slow, without considering the network overhead.

This meant that we could use WS for simple interactions by the client, but not for bootstrapping or incremental updates. We investigated X3D, but this is also XML-based, so would be too slow. We also considered OpenHSF [OpenHSF 2004] (our investigation was based on HOOPS version 9.0), which would supply an incremental stream for bootstrapping, but we also wanted incremental updates.

To give maximum control over data transmission, we opted for a simple custom protocol, where each Java class implemented a known interface for set/get methods (such as `getColour`, `setPosition`, etc.). Our initial RAVE implementation was for a single user, so our methods simply mapped to a given implementation (currently Java3D or a “Nil” implementation that can only store data, unable to render). We then created client and server sets of classes that wrapped RAVE objects, exposing an identical API. Once the wrapped objects were in use, the code simply delegated the set/get method to the wrapped implementation, but any set methods also sent the data over the network. To send data, Java introspection was used to discover what interfaces were implemented; from this, the network code can request certain data types to be sent (e.g. if an object implemented the `Colour` interface, then the `getColour` method would be called and the colour sent over the network). This enabled us to rapidly add new object types by implementing known interfaces, so new scene tree nodes were instantly supported with network transmission.

The set/get approach is sufficient for our needs at present, but we are free to extend this as required, such as an add method to introduce new vertices, or specific methods for new data types, such as voxels. Volumetric data will spawn new requirements, for instance changing the mapping from density to colour, which is not appropriate with polygonal data.

Note that the client and server classes effectively implement a shared scene graph—if any client makes a modification to its local copy of the scene graph (be it a render service or an active client), then the modifications are reflected back to the data service, which will then forward these modifications to all interested clients.

4.7 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. Clients can also control the cameras of other users, enabling RAVE to “remote control” other devices; an example of this is using a PDA to steer a large-scale display device (such as a projector or a Portico WorkWall).

To interact with the scene, objects are selected by clicking on them; an available interaction is selected from a menu, and launched by dragging the pointer. This simple interface was selected as it easily maps between a windowed environment (mouse click, mouse drag), a PDA (stylus “pick” and drag) and a VR environment (wand-based selection). There are three forms of interaction available to the clients:

1. Interact with unselected object
2. Interact with selected object
3. Interact relative to selected object

Unselected objects are those that cannot be seen to be clicked on—such as the user’s camera. Examples of interactions include:

Roll camera Select your camera in the “Interact with unselected object” menu and drag on-screen to roll the camera around the X or Y axis.

Move object Select the object on-screen and then select “Move object” in the “Interact with selected object” menu. Dragging the pointer will then move the object parallel to the viewing plane.

Rotate around object Select the object to rotate your camera around, then select “Rotate around object” in the “Interact relative to selected object” menu. Dragging the pointer will orbit the camera around the selected object.

On an active client, the interactions are discovered through Java introspection—data objects implement the “interaction” interfaces that match the three different types outlined above. A thin client instead requests interactions from the render service using Web Services; the render service then performs the introspection and returns the results as a series of menu options. Note that scene interactions are automatically reflected to all users as we implement a shared scene graph.

These interactions are sufficient to navigate around the scene, and to update/modify it. We have yet to implement more advanced collaboration, such as announcement of interesting viewpoints, attaching to other user’s cameras (enabling a guided tour), audio communication, etc. Security is not implemented in the system, so any user can modify any object at any time. Eventually we will implement a per-object locking approach, rather than a more restrictive master-slave approach (where one user may modify anything whilst all other users are locked out).

5 Experimental Results

Our experiment consisted of a live demonstration of a wireless PDA interacting with a laptop, both located at SuperComputing 2004 (Pittsburgh, PA, USA). However, the remote services used were located in Cardiff, Wales, UK. We describe the equipment used, how RAVE was operated and the resulting experience of collaboration.

5.1 Configuration

A Toshiba Satellite Pro M10 laptop was used as an active client for local rendering, whilst a Sharp Zaurus C760 PDA was used as a thin client. Both these devices were located in Pittsburgh at SC2004, and utilised resources in WeSC and the Cardiff School of Computer Science. An SGI Onyx 300 at WeSC was running a data service and an AMD-based Fedora 2 Linux server at the Cardiff School of Computer Science was running a render service. The Welsh Service-Orientated grid was also available in WeSC, but we selected the above machines to test heterogeneity and machines separation (rather than all the machines connected to the same network switch). The machines used are summarised in Table 1.

The data service supplied geometry to the laptop in SC2004, whilst the render service distributed a stream of rendered frames to the PDA; both of these links were trans-Atlantic. Finally, the WeSC UDDI server was used by the laptop to discover the available services. The configuration is presented in Figure 2. The network between the laptop in SC2004 and the machines in Cardiff, UK had 15 hops and a latency of approximately 0.1s. The wireless network (with the PDA) incurred an additional penalty of approximately 4ms, so is insignificant compared to the trans-Atlantic connection.

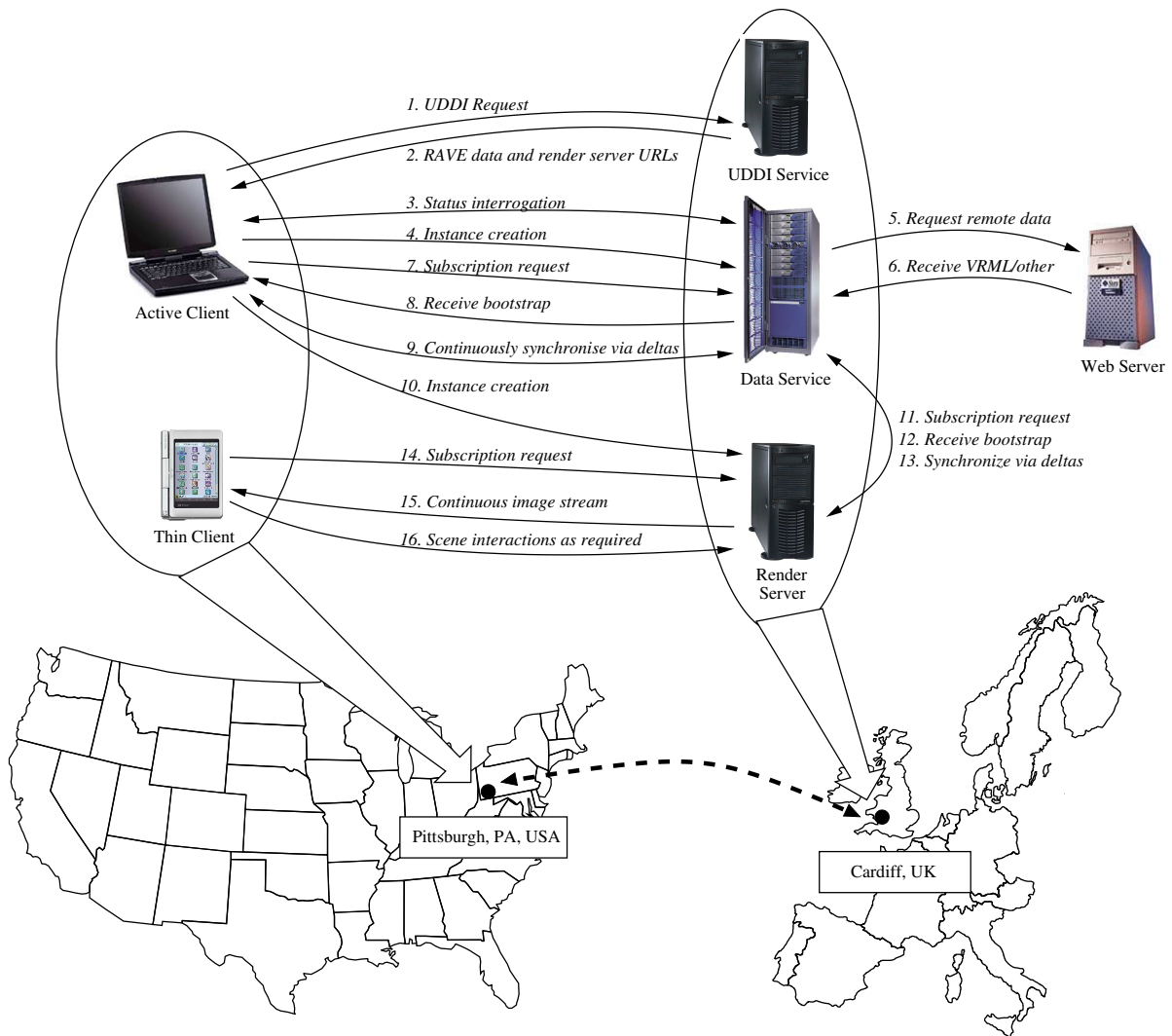


Figure 2: Test configuration as used at SC2004

Table 1: Equipment used at SC2004

Machine	Operating System	Location	CPU	Memory	GPU
Toshiba Satellite Pro M10	Windows XP	SC2004	1.6GHz Intel Centrino	512Mb	GeForce Go! 420
Sharp Zaurus C760	Embedded Linux	SC2004	400MHz Intel XScale	32Mb	None
Linux server	Fedora 2 Linux	Cardiff	2×1.8GHz AMD Opteron	2Gb	GeForce FX600
SGI Onyx 300	IRIX	Cardiff	32×500MHz MIPS	32Gb	InfiniteReality

5.2 Usage Sequence

We now describe the use sequence of the RAVE system, with each step covered in a separate section.

5.2.1 Service Discovery

Firstly, the laptop was informed of the location of a UDDI server (via a Java .properties file); the RAVE service manager then contacted the UDDI server to discover implementations of data and render services (Figure 2 steps 1-2). Each service was then interrogated via Web Service calls, requesting the load on the machine, available memory, number of service instances located at each service and the number of concurrent users (Figure 2 step 3). A WS call was then used to request a network testing socket; this socket was located on the remote service, and was used to reflect packets sent from the local client. The local client then timed network packets of increasing size, to discover network bandwidth and latency. A screen-shot of the RAVE service manager is presented in Figure 3.

5.2.2 Data Service Instantiation

The SGI Onyx was selected in the list of available data services (refer to (a) in Figure 3), with the available data instances then listed (refer to (b)). We wished to visualize a Diffusion Tensor Imaging dataset [Connell and Bastin 2004], IPRC molecule [Deisenhofer et al. 1995] from the Protein Data Bank [H.M.Berman et al. 2000] and a subset of the ETOPO data [National Geophysical Data Center 2004]. For each of these, the URL of the dataset was entered into the GUI (refer to (c)), the SGI Onyx data service was selected and the service instance created. A comparison of the datasets is presented in Table 2¹.

Table 2: Comparison of datasets

Name of dataset	Memory used	Number of polygons	Number of nodes
Diffusion Tensor Imaging	29.8Mb	948kpoly	2200
IPRC molecule	109.9Mb	4.6Mpoly	3
ETOPO	23.2Mb	546kpoly	29,000

During creation, the instance creation WS call was made (Figure 2 step 4); the remote data service then contacted the remote data source (step 5) and received the data (step 6). Upon completion, the data service was then ready to serve this particular dataset to RAVE clients, and the GUI was automatically updated to list the new data instance.

5.2.3 Active Client Instantiation

The user selected the new data instance in the GUI (refer to (b) in Figure 3), which entered the service details in the client creation section of the GUI (refer to (g)). The user selected "Create Java3D active client" and entered the size of the on-screen image and the name they would like associated with their avatar. A subscription request was sent via a WS call (Figure 2 step 7). The data service verified such a data instance was present, and registered the client with the dataset, returning a unique ID code representing that client.

¹"Memory used" excludes Java3D/OpenGL overhead, such as Display Lists, internal Java3D workspace, etc.

The client then used WS calls to request a TCP/IP socket to receive binary data, and upon a successful port handshake (using the unique ID code) the client was bootstrapped with the data (Figure 2 step 8). Once bootstrap was complete, the data service and active client continuously exchanged messages over TCP/IP to reflect changes made to the dataset (either locally at the client or remotely at the data service, possibly by other clients, as indicated by step 9). A view of the DTI dataset from the laptop is presented in the Colour Plate 1.

5.2.4 Render Service Instantiation

The PDA has insufficient internal memory to store the DTI dataset, without considering how quickly it could render the dataset (as it does not possess a graphics processor). Hence we had to use a render service to perform the rendering. In this case, the user selected the Linux server in the "Available Render Services" (refer to (d) in Figure 3) and the SGI Onyx in the "Available Data Services". The DTI dataset was then selected from the data service, representing that the user wanted to connect the render service to this data service instance. The GUI was updated to offer the user the opportunity of creating a new render service instance (refer to (f) in Figure 3). The user entered a description of the render service instance, and the laptop sent a WS message to the render service to request a new instance (Figure 2 step 10).

The render service then performed a similar series of WS calls as the active client in the previous section, requesting a subscription to the data service instance, receiving bootstrap data and continuously synchronising after bootstrap (Figure 2 steps 11-13).

Upon completion, the laptop's GUI was automatically updated to reveal the newly available render service instance.

5.2.5 Thin Client Instantiation: PDA

The user now entered the render service URL and the render service instance name (as shown in the laptop GUI) into the PDA's GUI, followed by the desired screen buffer size and the name of their avatar. In this instance, our PDA had a high-resolution screen, so an image size of 400x400 was used, with an example screen shot presented in Colour Plate 2 (showing the IPRC molecule dataset).

5.2.6 Thin Client Instantiation: Laptop

Although the laptop had a graphics processor, it could not render very large datasets (due to lack of memory and render speed). For an example, the user carried out the same procedure as detailed in Sections 5.2.2-5.2.4, but instead entered the URL for a copy of the ETOPO dataset. The render service instance was then selected, and a Java thin client was created (refer to (g) in Figure 3) An example screen shot is presented in Colour Plate 3, taken from a PC running Linux.

5.2.7 Selection of Client: Active or Thin

If a client is capable of rendering the dataset, then the user can opt to run an active client and download the scene directly from the data service. However, if the client has a high-speed network connection and a fast render service is available, the user could instead opt to run a thin client and use the render service.

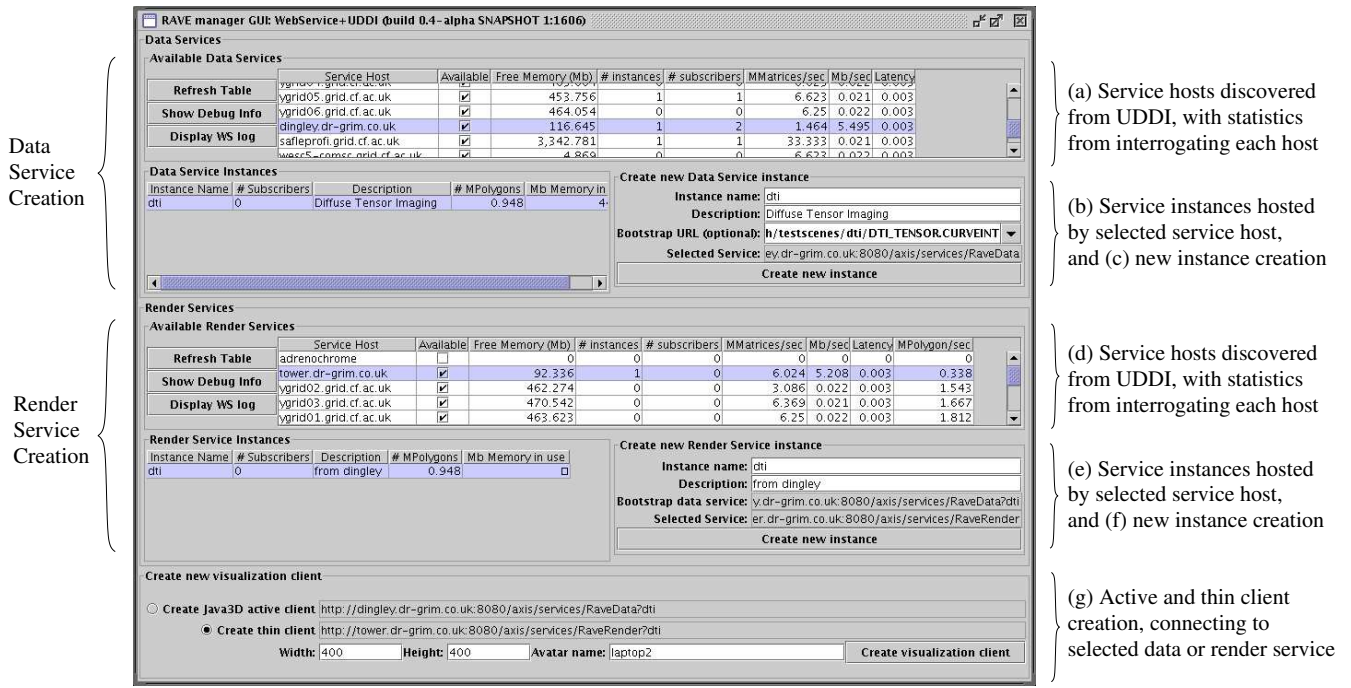


Figure 3: RAVE service manager

The user can determine this by examining the UDDI GUI—if a render service is listed with a higher polygon throughput than that of the client, and it has sufficient network bandwidth, then the user would receive a higher frame-rate with the render server than with local rendering. A simple heuristic can be used to determine this (where FPS=“Frames Per Second”):

$$FPS_{poly} = \frac{\text{polygons per second}}{\text{number of polygons in model}} \quad (1)$$

$$FPS_{frame} = \frac{\text{network bandwidth}}{\text{image size}} \quad (2)$$

$$FPS_{max} = \text{Max}(FPS_{poly}, FPS_{frame}) \quad (3)$$

A render service should be selected if it can deliver a higher FPS than the local client, and that the latency is within the user’s threshold. For instance, a render service may be able to deliver a far higher FPS, but this is unusable if the latency between user interaction and receipt of the resulting frame is not at an interactive rate. This selection approach will be used when we implement a more automated GUI; refer to Section 6.

5.3 Collaboration

When the laptop and PDA were both displaying the DTI dataset, an avatar was visible for each client (the laptop can see the PDA’s avatar, and vice-versa). This enabled the laptop to follow the PDA around the dataset, and share the same view. At this early stage of the project, we have only implemented minimal collaboration support, so we have yet to introduce gesticulation or direct communication (textual, speech or other), so the experience was limited. However, the thin and active clients were designed to continuously send messages when navigating the scene, at a frequency independent

of the local rendering speed. Hence if the local machine slowed down, the actual movement of the avatar in space was still continuous. This was observed by the frame rate of the PDA being around 1-2fps whilst the PDA’s avatar was seen moving smoothly on the laptop.

The limitation of the collaboration (aside from an incomplete feature set) was mainly the speed of update on the PDA, which was hampered by the wireless connection. With exclusive use of an 802.11b wireless connection, the PDA can achieve around 4fps with a 400×400 run-length encoded image (representing 1.8Mb/sec, whereas maximum bandwidth we have achieved through raw socket tests was around 0.6Mb/sec), so we are limited by the network bandwidth. At SC2004, we were obtaining around 1-2fps due to the wireless bandwidth being shared with other show attendees. As the results are from observed rather than measured delays, they have a margin of error, but the observed latency between interacting with the PDA’s GUI and a result arriving on the laptop was around 0.5s, with a total delay of around 0.7s before seeing the rendered result on the laptop.

We are normally limited by available wireless network bandwidth (or render speed with more complex models, such as the ETOPO dataset), so using the PDA via such a high latency link was not a major problem. The link was fast enough for the user to actively steer the laptop (in front of a live audience) with the PDA; the visible latency was between 0.5 and 1.5 seconds, depending on the dataset (such as time for the laptop to re-render the scene, etc.).

We must note that the size of the avatar was an issue; depending on the scale of the dataset, the avatar could be so small as to be lost from view when it receded into the distance. We need to add a radar map or some other method of finding the avatar (such as displaying the avatar’s name over the top of the image).

6 Future Work

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 [Bethel and Shalf 2003]).

We will investigate image compression algorithms that can react to network load, such as lossy, incremental algorithms. Such an algorithm will enable us to maintain a given frame rate, whilst incrementally improving the received image if the scene or camera do not change (forming a so-called “golden thread”—if the algorithm is run for long enough, a perfect image will eventually be received). This will enable us to run RAVE on very low bandwidth devices such as a mobile telephone (using a GPRS or G3 enabled handset).

We will also investigate automatic selection of level-of-detail, to attempt to maintain a given frame rate and to enable a dataset to be rendered on a machine with otherwise insufficient resources.

Initially, we have the capability to send either the geometry (for an active client) or the frame buffer (for a thin client); we will also investigate distributed rendering, where part of the dataset is rendered on a render server and remainder rendered locally. This presents a hybrid between an active client and a thin client, and is akin to the approach taken in the ARTE environment [Martin 2002].

All image and data compression methods must be switchable, in case a user requires perfect detail and is prepared for slow frame rates (an example would be a medical practitioner examining data—they do not want important image detail obscured by a compression artifact).

The current GUI contains too much information for a casual user, and should be considered more of a testing interface. We will implement another GUI that merely requires the user to enter the URL of their dataset and a session name; the GUI can then use heuristics to determine which is the best data service to select (possibly a service with the dataset already loaded). Heuristics can select to run an active client or a thin client, again determining which is the best render service to use if required.

Issues were found when attempting to discover how much memory a given data or render service instance occupied. RAVE services occupy a Java Virtual machine (JVM) that is shared with other web services on a given machine, so the total amount of memory used by the JVM does not reveal how much memory an individual instance is using. We have added a simple guesstimate at present (by enumerating the underlying scene graph and counting vertices, parameters, etc.), but are working on a general purpose approach via introspection.

We will rerun our experiments across Britain, linking Cardiff to remote sites with RAVE. This will enable us to evaluate the user experience, and discover exactly what is required to support the collaborative aspect of RAVE.

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 hand-held device to interact with a user in an large-scale environment (such as a Portico WorkWall or a projected display). This also enables the PDA to be used as a private display by the presenter, and then to remote-control the publicly visible display when required (all from the PDA).

Our work shows that Java3D can be used to visualise highly complex datasets, and also support devices as a background service without disturbing the local user. This promotes the use of rendering hardware to support remote users, rather than just for a user sitting in front of a console.

References

- APACHE. 2004. Apache Web Services Project—Axis. <http://ws.apache.org/axis/index.html>.
- APACHE. 2004. Apache Web Services Project—Axis C++. <http://ws.apache.org/axis/cpp/index.html>.
- APACHE. 2004. Apache Web Services Project—jUDDI. <http://ws.apache.org/juddi/>.
- BETHEL, E., AND SHALF, J. 2003. Grid-distributed visualizations using connectionless protocols. *IEEE Computer Graphics & Applications* (March/April), 51–59.
- BRODLIE, K., BROOKE, J., M.CHEN, CHISNALL, D., FEWINGS, A., HUGHES, C., JOHN, N. W., JONES, M. W., RIDING, M., AND ROARD, N. 2004. Visual Supercomputing—Technologies, Applications and Challenges. In *STAR—State of the Art Report, Eurographics*.
- BRODLIE, K., WOOD, J., DUCE, D., AND SAGAR, M. 2004. gViz: Visualization and Computational Steering on the Grid. In *Proceedings of UK e-Science All Hands Meeting 2004*.
- CHARTERS, S. M., HOLLIMAN, N. S., AND MUNRO, M. 2004. Visualisation on the Grid: A Web Service Approach. In *Proceedings of UK e-Science All Hands Meeting 2004*.
- CHEN, J. X., YANG, Y., AND LOFTIN, B. 2003. MUVEES: a PC-based Multi-User Virtual Environment for Learning. In *Proceedings of the IEEE International Symposium on Virtual Reality 2003 (VR '03)*, IEEE Computer Society, 163–170.
- CONNELL, M., AND BASTIN, M. 2004. Diffusion Tensor Imaging (DTI) dataset. Personal communication, SHEFC Brain Imaging Research Centre for Scotland.
- DEISENHOFER, J., EPP, O., SINNING, I., AND MICHEL, H. 1995. Crystallographic refinement at 2.3 Å resolution and refined model of the photosynthetic reaction centre from rhodospseudomonas viridis. *Journal of Molecular Biology* 246, 429. PDB ID: 1PRC.

- FOSTER, I., KESSELMAN, C., NICK, K. M., AND TUECKE, S. 2002. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Tech. rep., Globus, February.
- GRIMSTEAD, I. J., AVIS, N. J., AND WALKER, D. W. 2004. Automatic Distribution of Rendering Workloads in a Grid Enabled Collaborative Visualization Environment. In *Proceedings of SC2004: SuperComputing 2004*.
- H.M.BERMAN, J.WESTBROOK, Z.FENG, G.GILLILAND, T.N.BHAT, H.WEISSIG, I.N.SHINDYALOV, AND P.E.BOURNE. 2000. The Protein Data Bank. *Nucleic Acids Research* 28, 235–242.
- IBM. 2004. Grid and Web Services Standards to Converge. Press release at http://www.marketwire.com/mw/release_html_b1?release_id=61977, GLOBUSWORLD, January.
- IBM. 2004. IBM Test UDDI Registry. <https://uddi.ibm.com/testregistry/registry.html>.
- JOGL. 2004. Reference Implementation of the Java Bindings for OpenGL. <https://jogl.dev.java.net/>.
- JOHN, N. W. 2003. High Performance Visualization in a Hospital Operating Theatre. In *Proceedings of the Theory and Practice of Computer Graphics (TPCG03)*, IEEE Computer Society, 170–175.
- KRAULIS, P. 1999. MolScript. <http://www.avatar.se/molscript/>.
- MARTIN, I. M. 2002. Hybrid Transcoding for Adaptive Transmission of 3D Content. In *Proceedings of IEEE International Conference on Multimedia and Expo (ICME)*.
- NATIONAL GEOPHYSICAL DATA CENTER. 2004. GEODAS Grid Translator—Design-a-Grid. http://www.ngdc.noaa.gov/mgg/gdas/gd_designagrid.html.
- OASIS. 2002. Universal Description, Discovery and Integration (UDDI)—Version 2 Specifications. <http://www.oasis-open.org/committees/uddi-spec/doc/tcpspecs.htm>, July.
- OPENHSF, 2004. Open HOOPS Stream Format. <http://www.openhsf.org>.
- ROST, R. J., FRIEDBERG, J. D., AND NISHIMOTO, P. L. 1989. PEX: A Network-Transparent 3D Graphics System. *IEEE Computer Graphics & Applications* (July), 14–26.
- SANDHOLM, T., AND GAWOR, J. 2003. Globus Toolkit 3 Core—A Grid Service Container Framework. Globus Toolkit 3 Core White Paper, July.
- SGI. 2003. SGI and Sun Microsystems’ Software Platforms to Work Seamlessly Together with Java Bindings to OpenGL. http://www.sgi.com/company_info/newsroom/press_releases/2003/july/sgisu%20n_opengl.html, July.
- SGI. 2003. SGI OpenGL VizServer 3.1. Data sheet, SGI, March.
- SUN MICROSYSTEMS INC. 2004. Sun Contributes Four Java Breakthroughs to Open Source Community Including “Project Looking Glass” and Java 3D. <http://www.sun.com/smi/Press/sunflash/2004-06/sunflash.20040628.2.html>, June.
- TGS. 2004. TGS Java Open Inventor. http://www.tgs.com/index.htm?pro_div/3dms_j_main.htm.
- VAN ENGELEN, R. 2004. gSOAP. <http://www.cs.fsu.edu/~engelen/soap.html>.
- VTK. 2004. VTK—The Visualization Toolkit. <http://www.vtk.org/>.
- W3C. 2001. Web Services Description Language (WSDL) 1.1. <http://www.w3.org/TR/wsd1>, March.
- W3C. 2003. Simple Object Access Protocol (SOAP)—W3C Recommendation Version 1.2—Primer. <http://www.w3.org/TR/soap12-part0/>, June.
- WALTON, J. 2003. *Visualization Handbook*. Academic Press, ch. NAG’s IRIS Explorer.
- WESC. 2004. Welsh e-Science Centre. <http://www.wesc.ac.uk>.
- WOMACK, P., AND LEECH, J. 1998. OpenGL Graphics with the X Window System. <http://www.opengl.org/documentation/specs/glx/glx1.3.pdf>, October.
- WÖSSNER, U., SCHULZE-DÖBOLD, J., WALZ, S., AND LANG, U. 2002. Evaluation of a Collaborative Volume Rendering Application in a Distributed Virtual Environment. In *Proceedings of the 8th Eurographics Workshop on Virtual Environments (EGVE)*, 113–122.
- XJ3D. 2004. Xj3D—VRML97 and X3D Java Toolkit. <http://www.xj3d.org>.
- YEONG, W., HOWES, T., AND KILLE, S. 1995. RFC 1777—Lightweight Directory Access Protocol (LDAP). <http://www.faqs.org/rfcs/rfc1777.html>, March.